New Virtualization Techniques

Lic. Matías Zabaljáuregui
matiasz@info.unlp.edu.ar

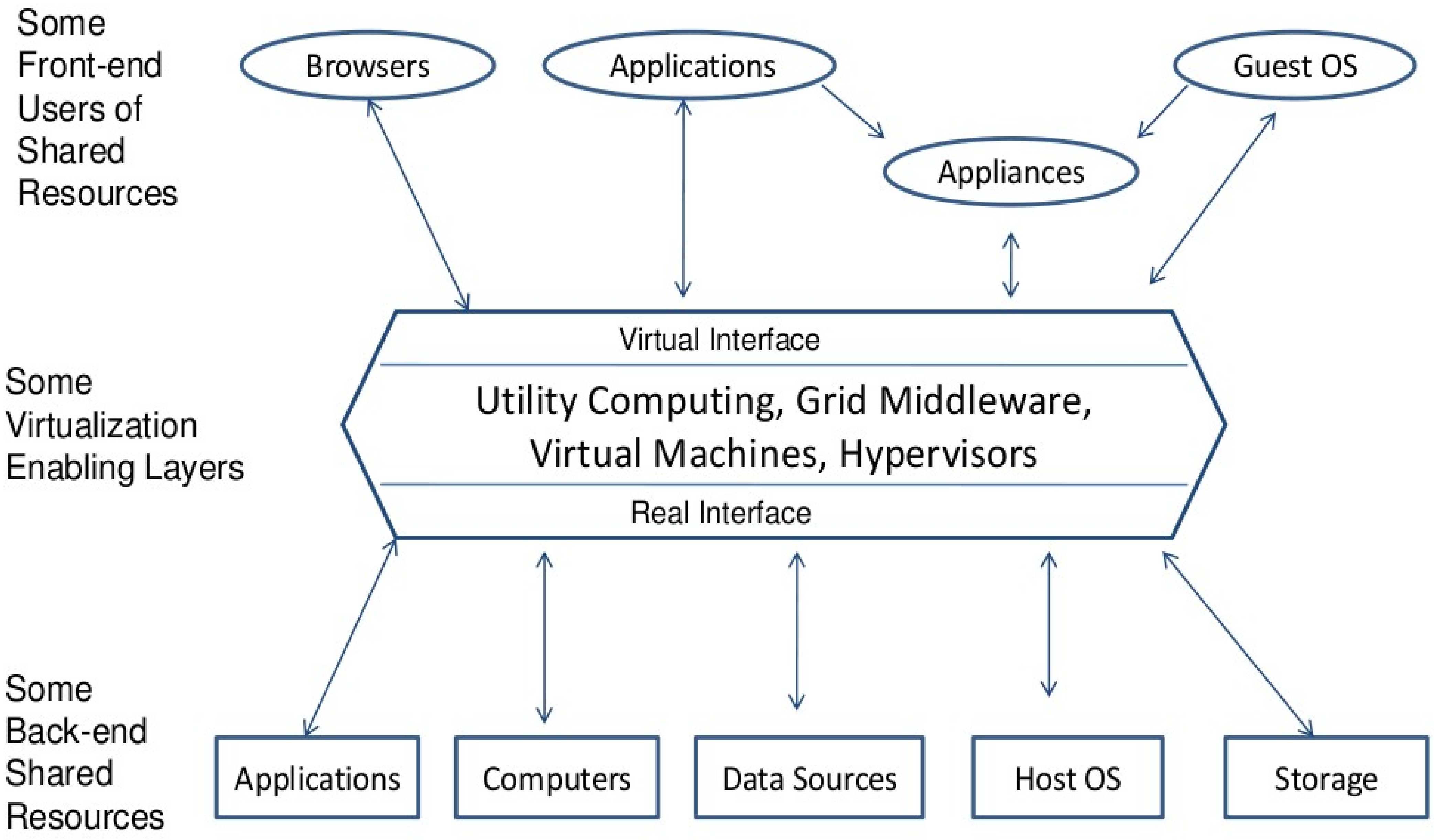
outline

- Introduction to virtualization concepts
- Techniques
  - Binary Translation
  - Paravirtualization
  - Hardware assisted

Introduction to virtualization concepts

- 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.

Generic Virtualization

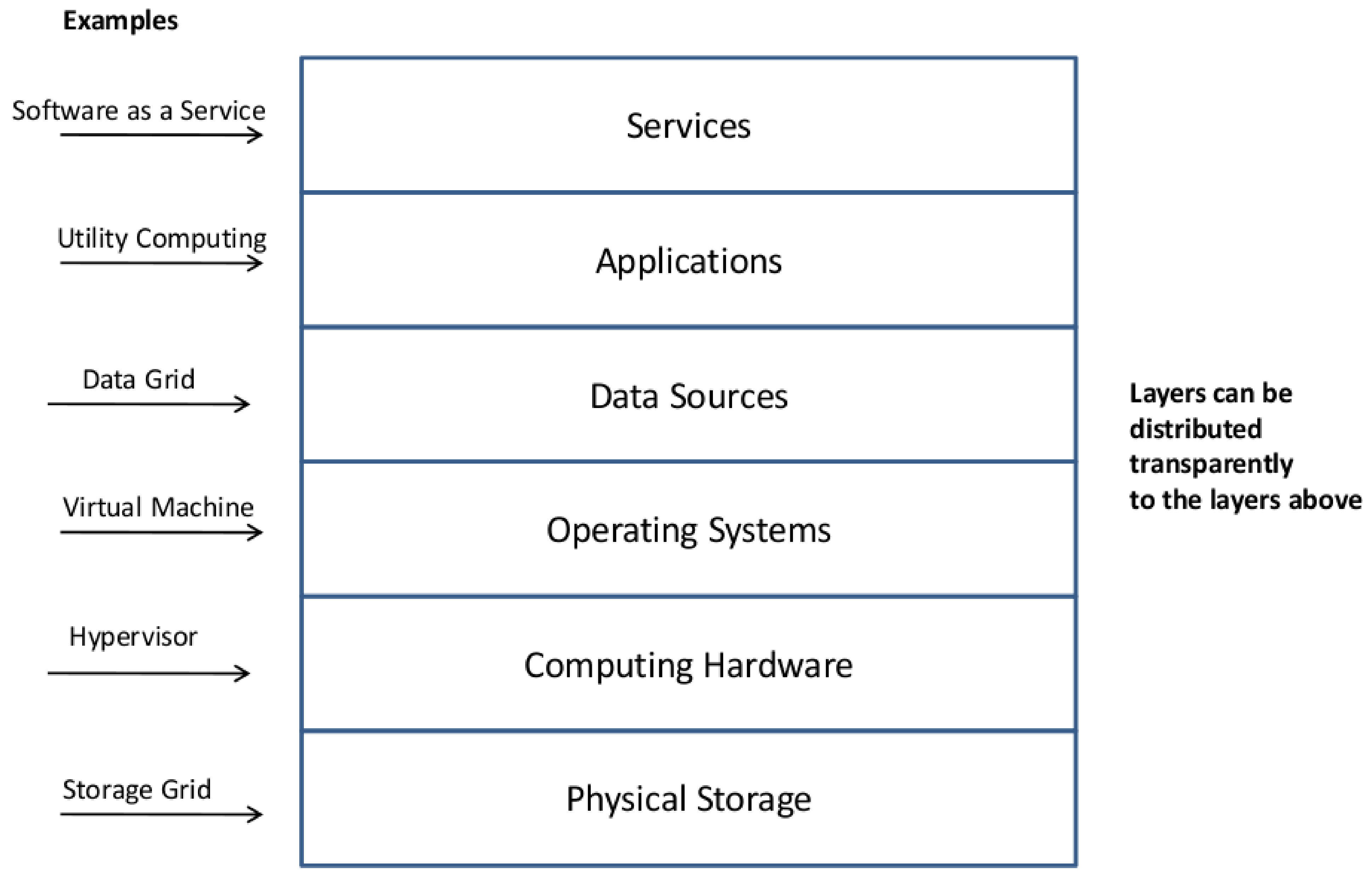

Some Front-end Users of Shared Resources
Browsers
Applications
Guest OS
Appliances
Some Virtualization Enabling Layers
Virtual Interface
Utility Computing, Grid Middleware, Virtual Machines, Hypervisors
Real Interface
Some Back-end Shared Resources
Applications
Computers
Data Sources
Host OS
Storage

generic virtualization

- **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 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).

Virtualization Layers

Virtualization enables access at any layer while hiding the layers below



Horizontal and Vertical Virtualization

- Another way of classifying virtualization techniques:
  - **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 mentioned before. This can also help to understand the difference between vertical and horizontal virtualization.

Horizontal Virtualization Alternatives

a. **Software as a Service (SaaS):** 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):** 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

Distributed Virtualization

- 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.

More on this in ”Grid – Virtualization relationships” (*http://linux.linti.unlp.edu.ar/kernel/wiki/images/1/1f/Grid-virtualization.pdf)*

operating systems and hardware virtualization techniques

- The Renaissance of Virtualization
- x86 virtualization introduction
- Classical virtualization
- The Challenges of x86 Hardware Virtualization
- Refinements to classical virtualization and "new" virtualization techniques

The Renaissance of Virtualization

- 1970s: virtual machines first used
- 1990s:
  - x86 becomes prominent server platform
  - No vertical integration in x86
  - Lack of enterprise features in commodity OSs
- 1999: VMWare first product to virtualize x86
- 2006: AMD and Intel offer hardware support

x86 virtualization introduction

- a virtualization layer is added between the hardware and operating system.

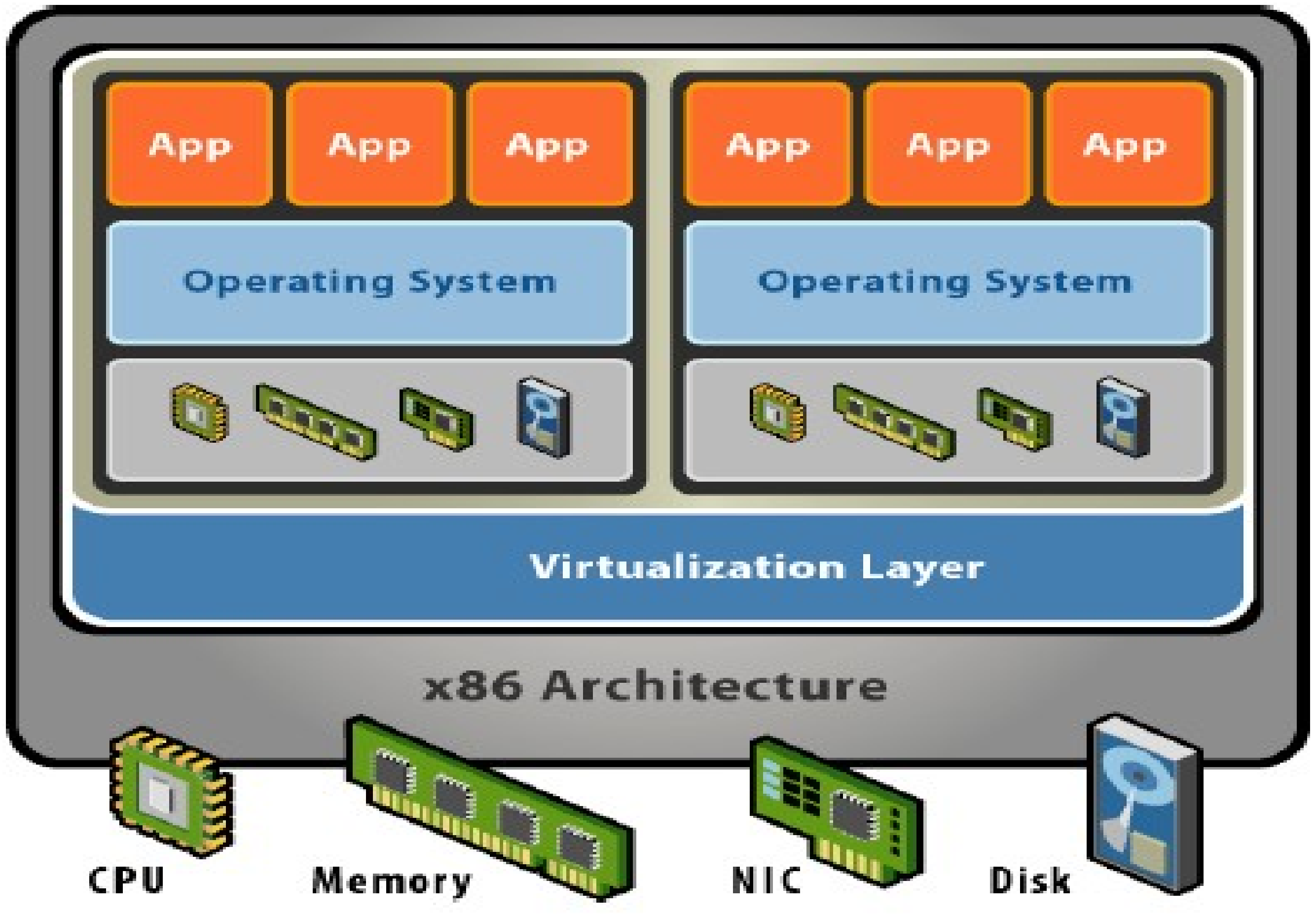


Figure 2 – x86 virtualization layer

- 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.

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.
- 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.**

hypervisor, vmm and guest

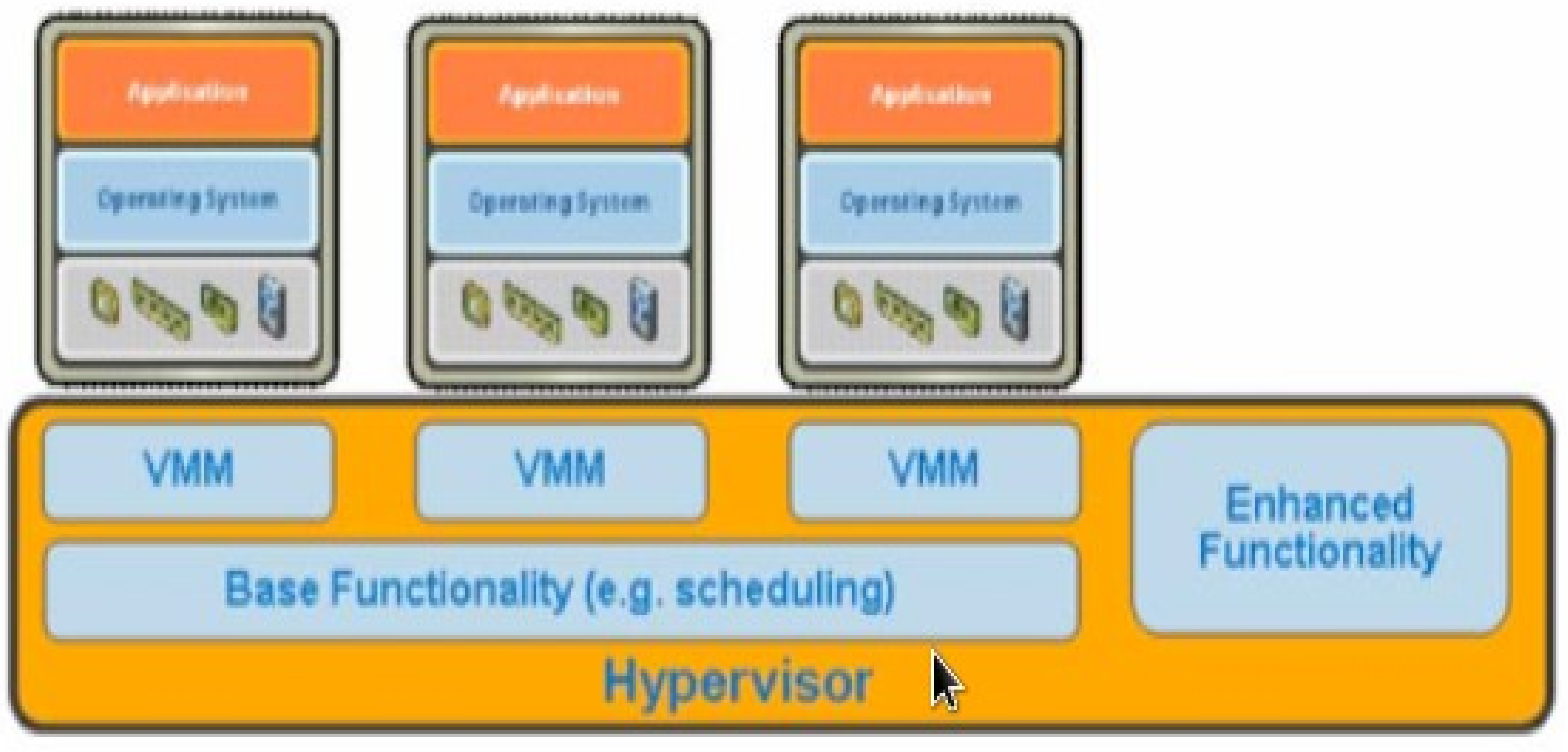


Figure 3 – The hypervisor manages virtual machine monitors that host virtual machines

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"***.
- 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.

Classically virtualizable and trap-and-emulate

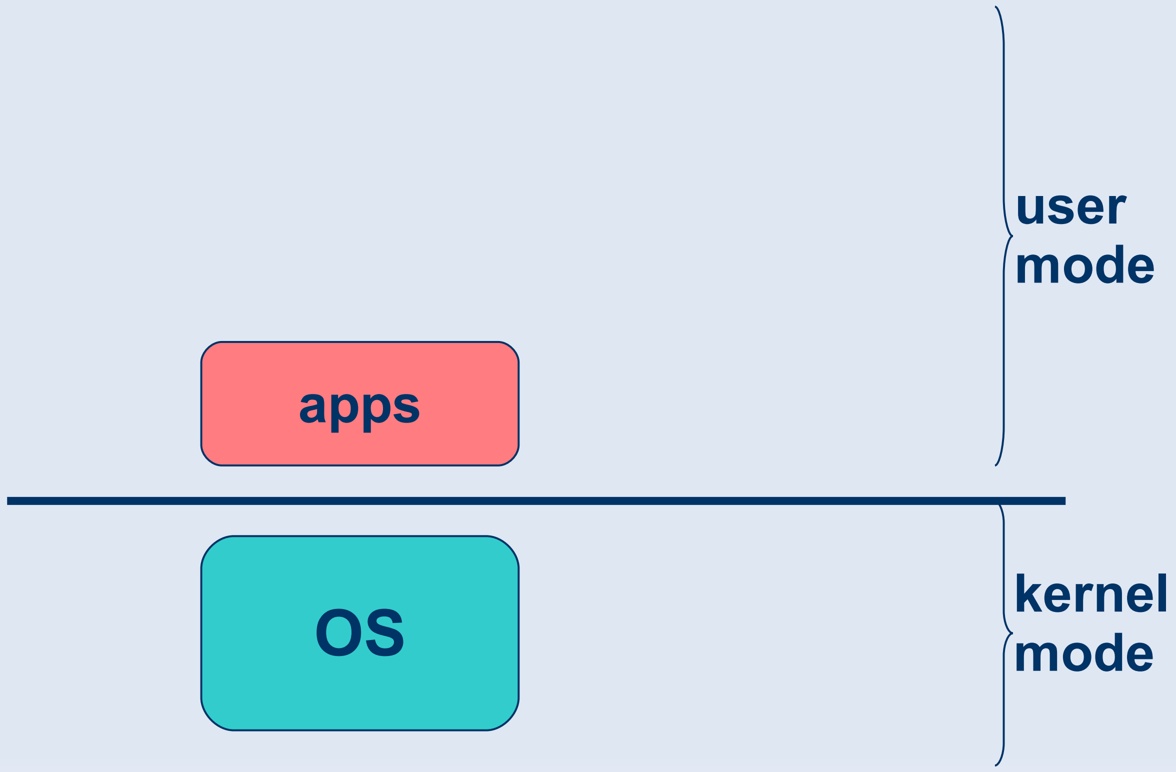
- In 1974, a particular VMM implementation style, trap-and-emulate, was so prevalent as to be considered the only practical method for virtualization.
- I will use the term **classically virtualizable** to describe an architecture that can be virtualized purely with trap-and-emulate.
- In this sense, **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 tracing**

De-privileging

- A classical VMM executes guest operating systems directly, but at a reduced privilege level.
- 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).
- 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.

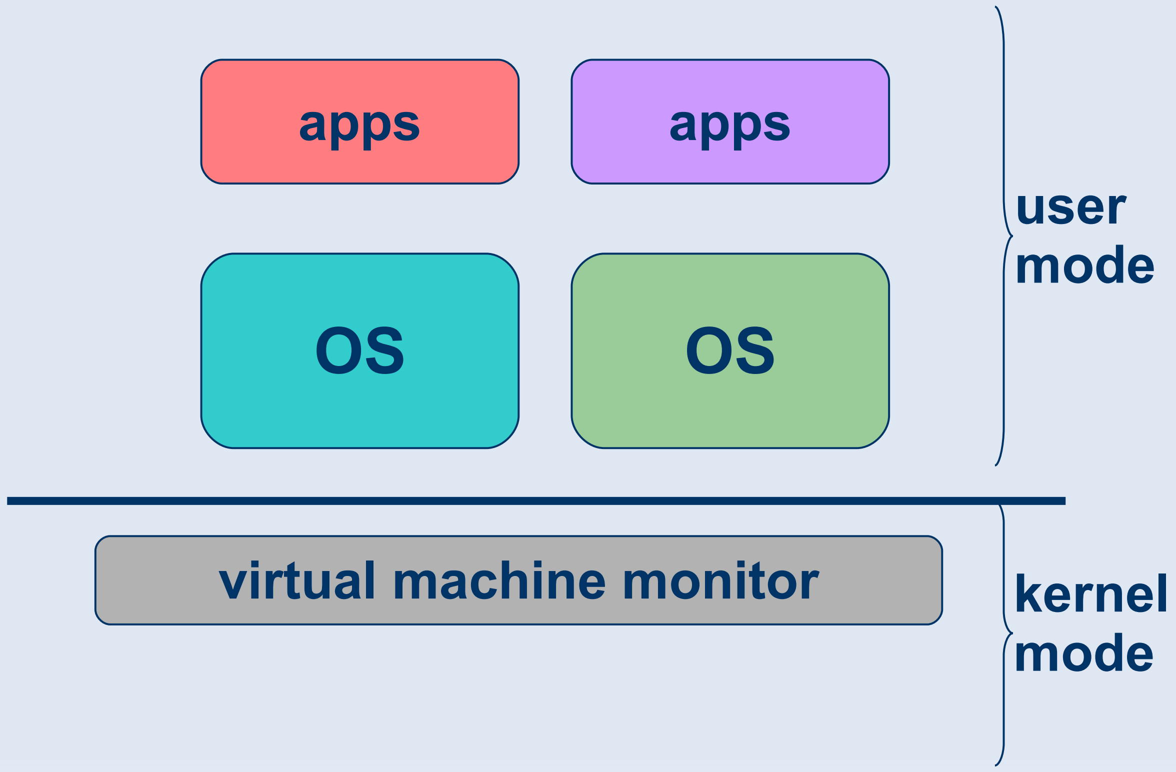
Trap-and-Emulate Virtualization

1. De-Privilege OS



Trap-and-Emulate Virtualization

1. De-Privilege OS



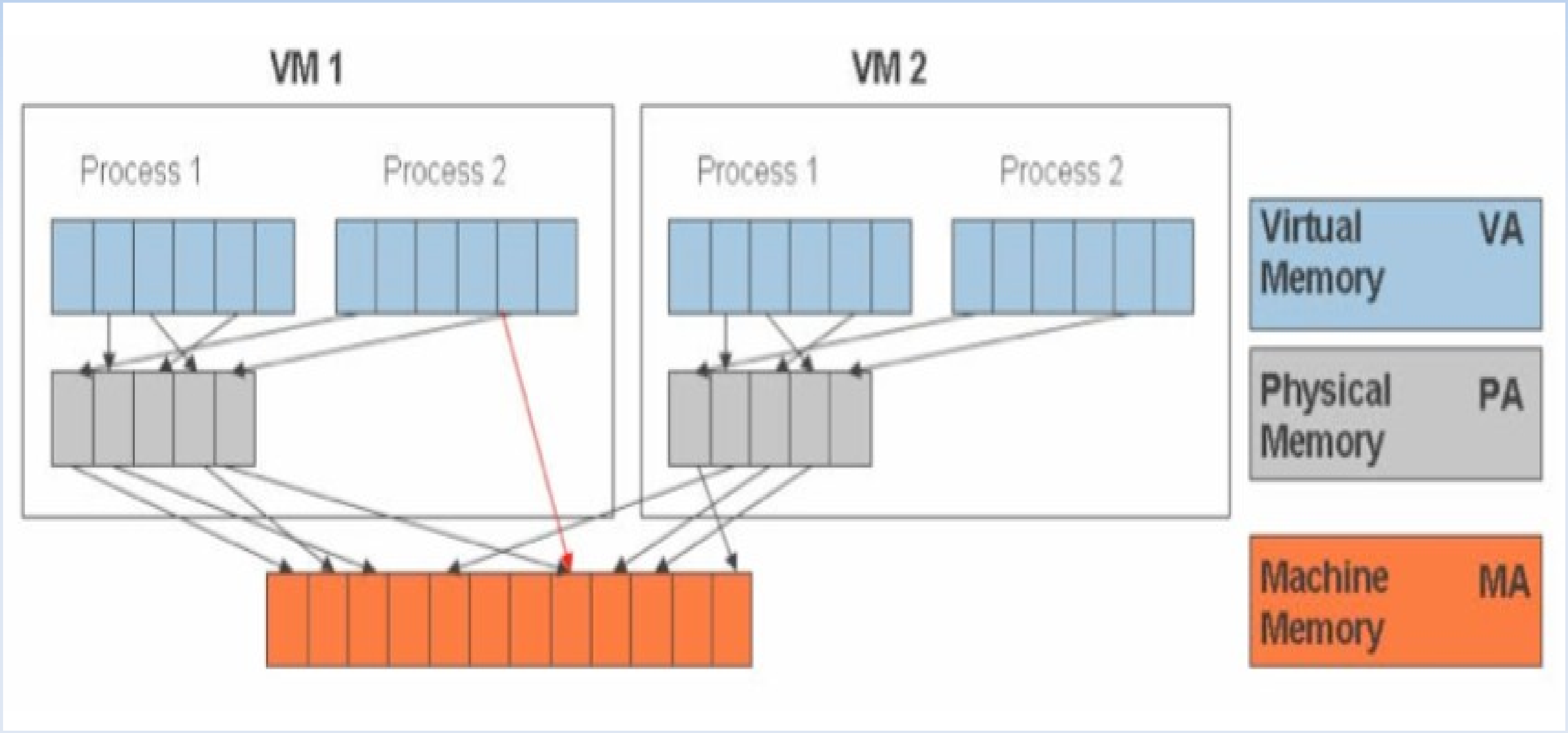
Primary and shadow structures

- 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.

example: memory virtualization

- 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.
- 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.

Shadow page tables

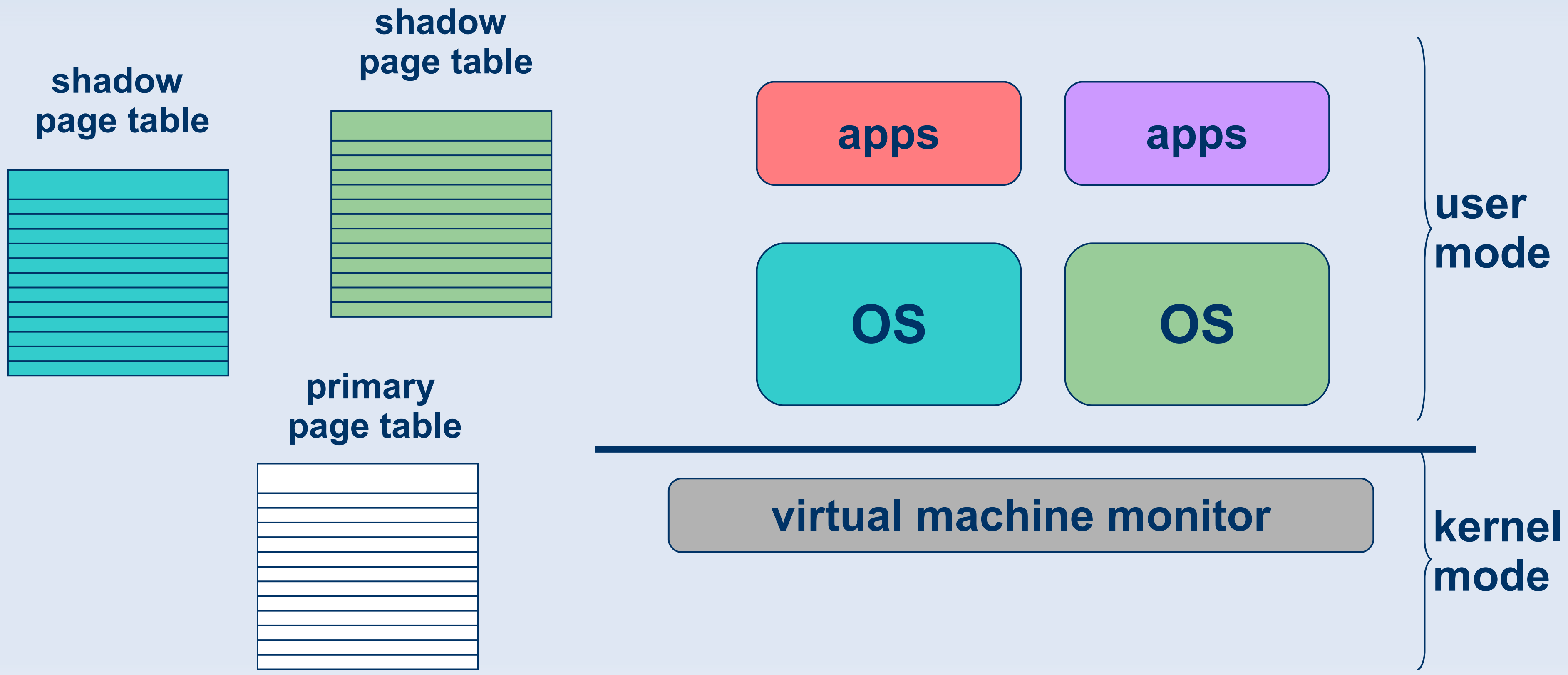


tracing

- 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.

Trap-and-Emulate Virtualization

1. De-Privilege OS
2. Shadow structures and memory tracing



Trap-and-Emulate Virtualization

- **Traps are expensive:** *"I made a performance test comparing direct software interrupt vs invalid opcode fault vs patching....* measuring the total count of cycles with rdtsc and then prints the hypercall average cycle cost. I made 5 runs for every scenario and these are the results:
  - 1) direct software interrupt: 2915, 2789, 2764, 2721, 2898
  - **2) emulation technique: 3410, 3681, 3466, 3392, 3780**
  - 3) patching (rewrite) technique: 2977, 2975, 2891, 2637, 2884
- Many traps unavoidable
  - E.g., page faults
- Important enhancements
  - “Paravirtualization” to reduce traps (e.g., Xen)
  - Hardware VM modes (e.g., IBM s370)

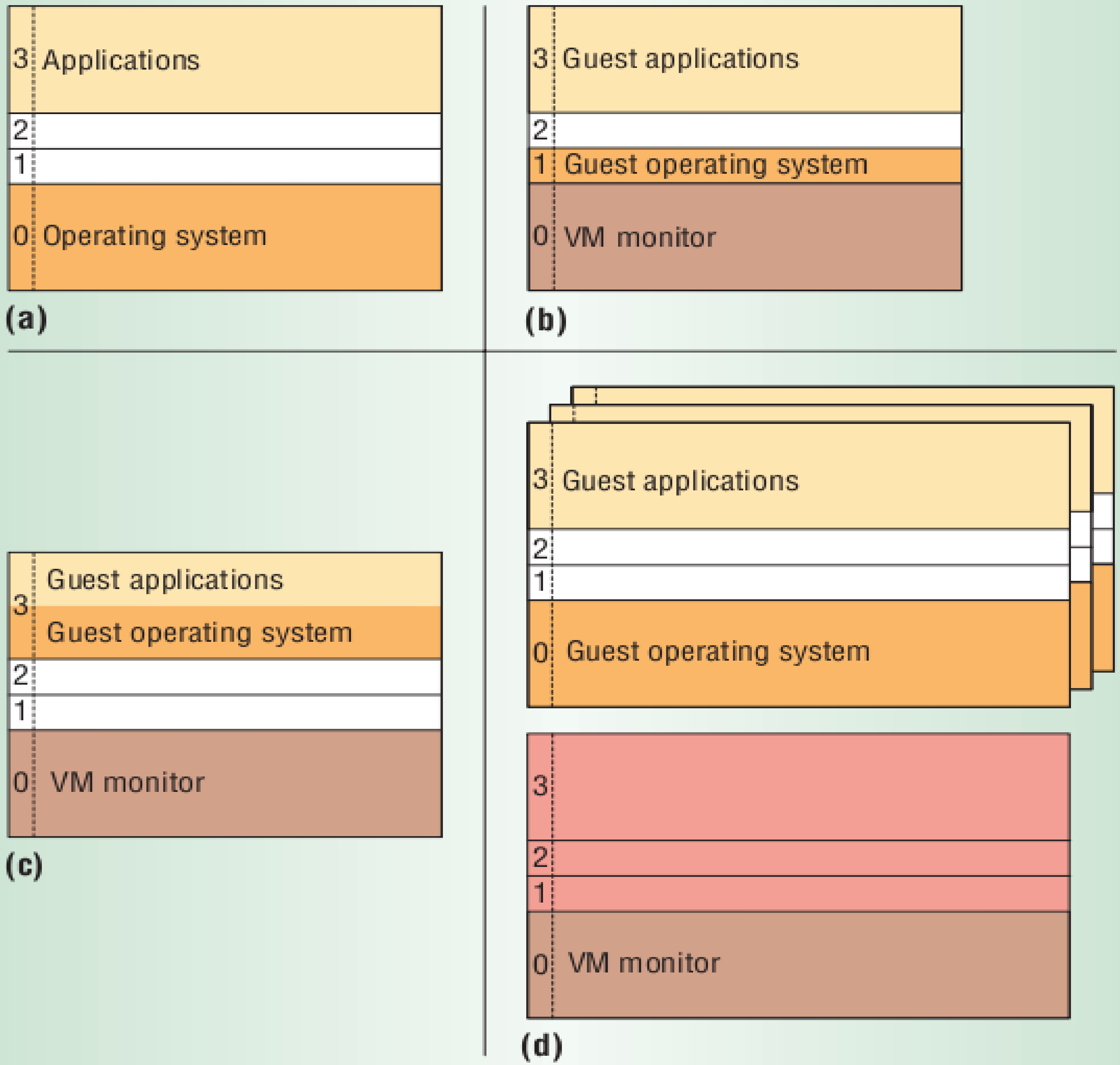
Can x86 Trap and Emulate?

- No
  - Even with 4 execution modes!
  - Key problem: **dual-purpose instructions don't trap**
- Classic Example: ***popf*** instruction
  - Same instruction behaves differently depending on execution mode
  - User Mode: changes ALU flags
  - Kernel Mode: changes ALU **and** system flags
  - <u>**Does not generate a trap in user mode**</u>

Robin, J. and Irvine, C. Analysis of Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. Proceedings of the 9th USENIX Security Symposium, Denver, CO, August 2000.

Deprivileging in x86

- 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)**.


3 Applications
2
1
0 Operating system
(a)
3 Guest applications
2
1 Guest operating system
0 VM monitor
(b)
3 Guest applications
Guest operating system
2
1
0 VM monitor
(c)
3 Guest applications
2
1
0 Guest operating system
3
2
1
0 VM monitor
(d)

The Challenges of x86 Hardware Virtualization

- **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, but **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 manage transitions between guest software and the VMM (For IA-32, these include the IDT and the GDT, which reside in the linear-address space.)

  - 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.**

Challenges of x86...

- **Nonfaulting access to privileged state:** 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.

Challenges of x86...

- **Interrupt virtualization:** 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.**

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. **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.**

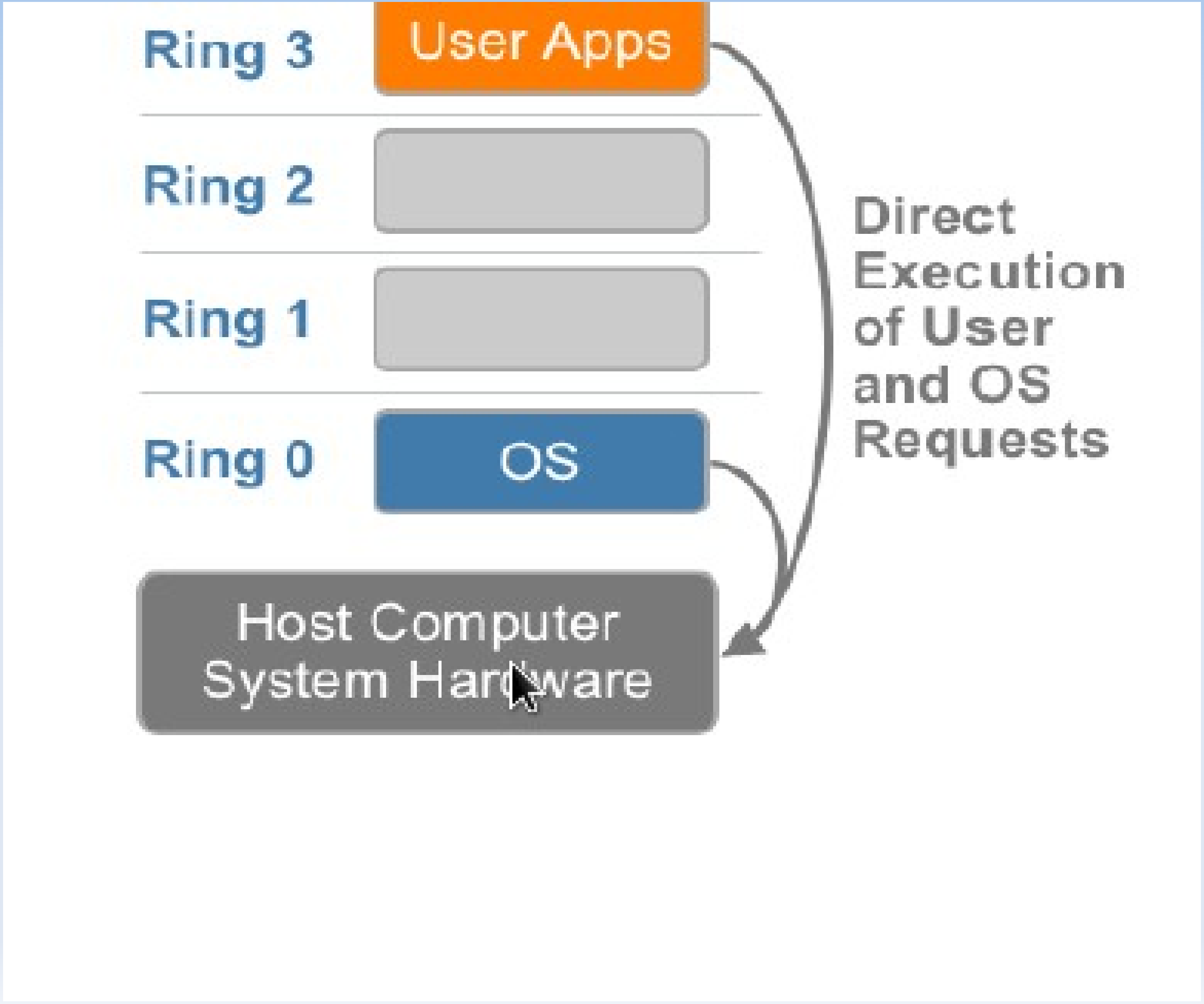
three alternative techniques

- 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 – 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.
- 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.**

OS and apps running on real hardware


Ring 3
User Apps
Ring 2
Ring 1
Ring 0
OS
Host Computer System Hardware
Direct Execution of User and OS Requests

Binary Translation VMM

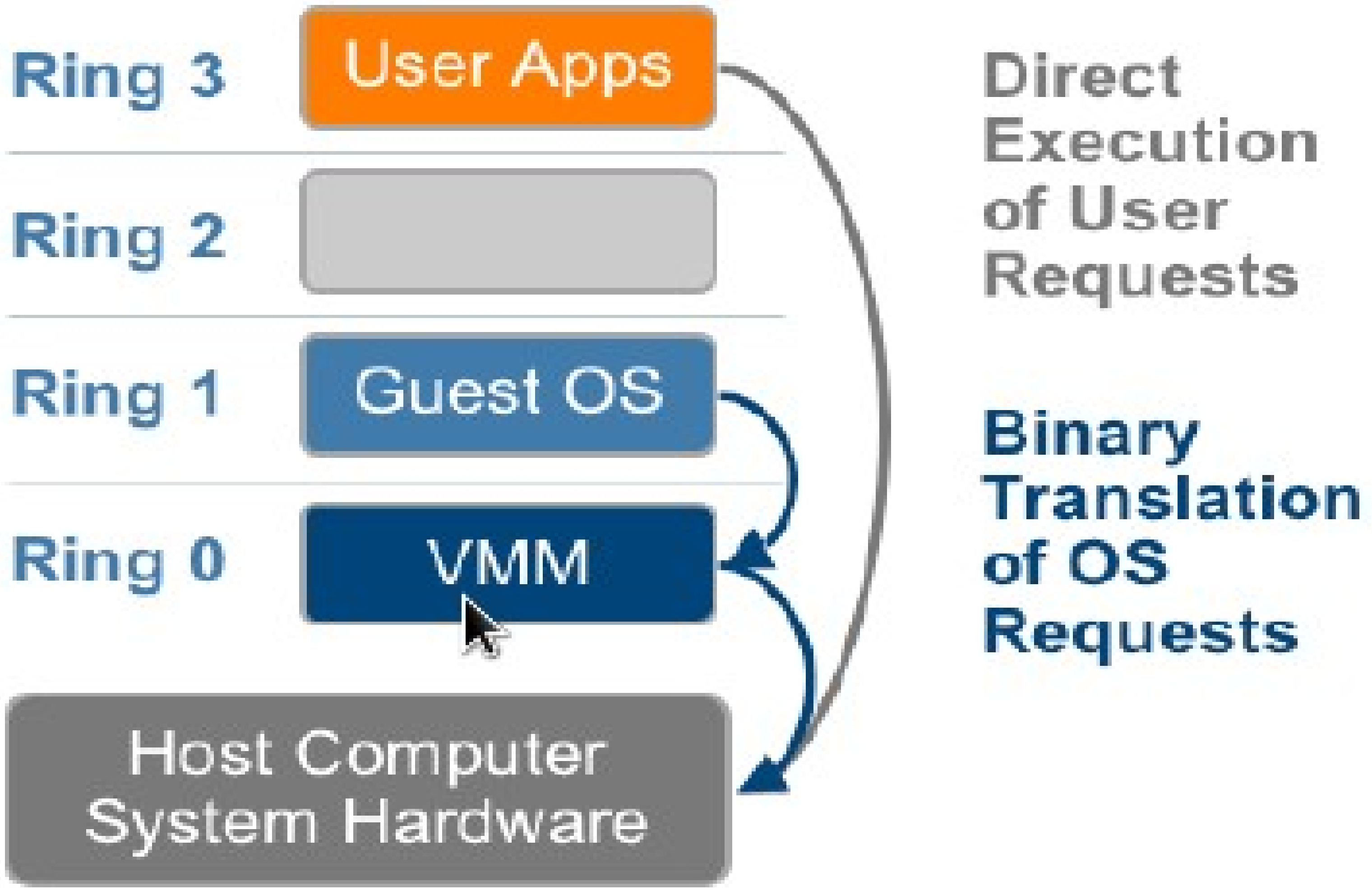

Ring 3
User Apps
Ring 2
Ring 1
Guest OS
Ring 0
VMM
Host Computer System Hardware
Direct Execution of User Requests
Binary Translation of OS Requests

Figure 5 – The binary translation approach to x86 virtualization

Technique 1 – Binary Translation

- 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.

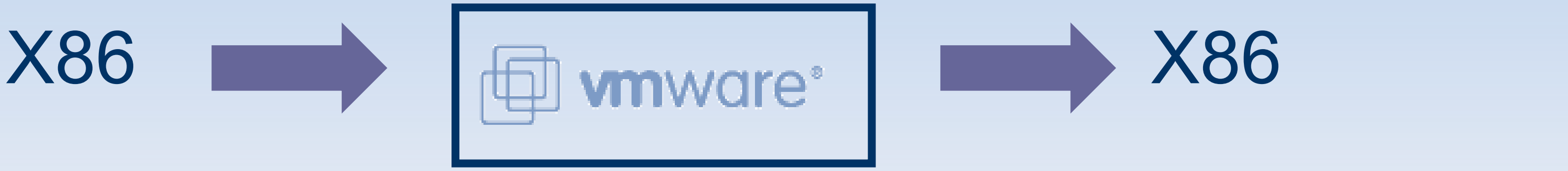
- This combination of binary translation and direct execution provides **Full Virtualization** as the guest OS is fully abstracted (completely decoupled) from the underlying hardware by the virtualization layer.

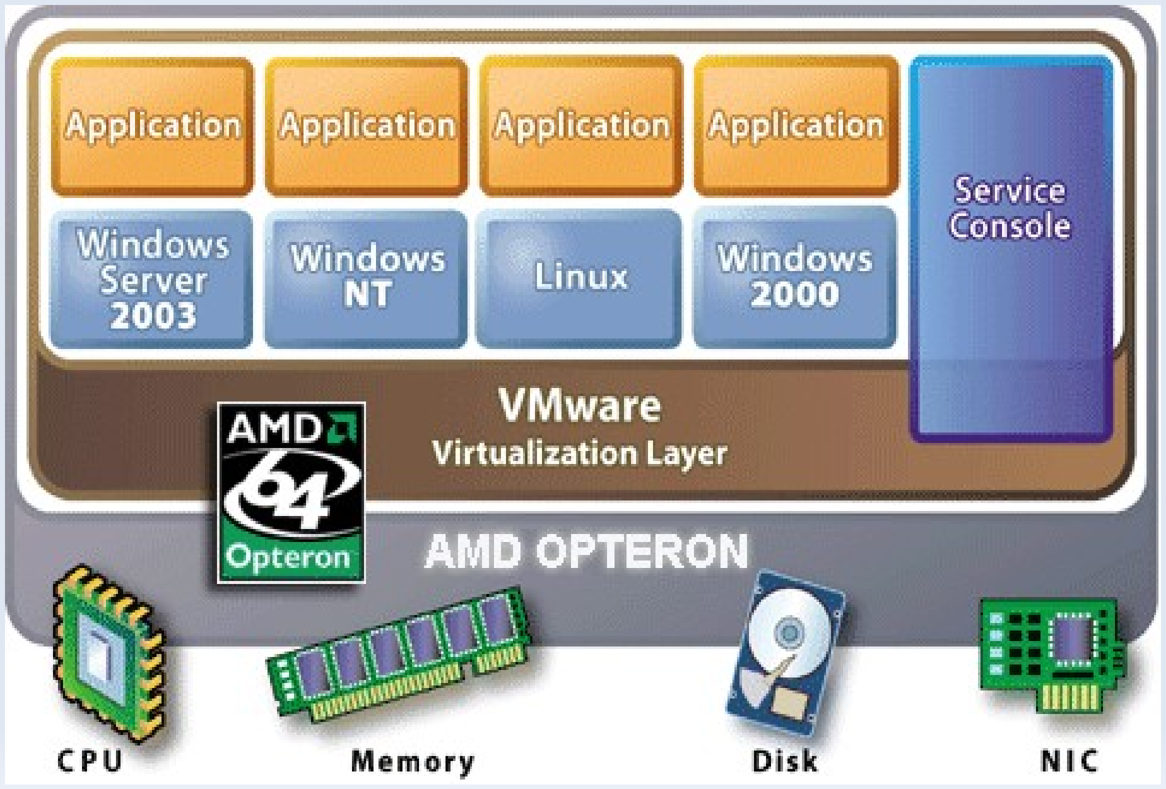
- The guest OS is not aware it is being virtualized and requires no modification. **Full virtualization is the only option that requires no hardware assist or operating system assist to virtualize sensitive and privileged instructions**. The hypervisor translates all operating system instructions on the fly and caches the results for future use, while user level instructions run unmodified at native speed.

Software Virtualization with VMWare

Binary translation

(mostly safe, user-mode)





Software Virtualization with VMWare

- vmware software VMM uses a translator with these properties:
  - **Binary**. Input is binary x86 code, not source code.
  - **Dynamic**. Translation happens at runtime, interleaved with execution of the generated code.
  - **On demand**. Code is translated only when it is about to execute. This laziness side-steps the problem of telling code and data apart.
  - **System level**. The translator makes no assumptions about the guest code. Rules are set by the x86 ISA, not by a higher-level ABI.
  - **Subsetting**. The translator's input is the full x86 instruction set, including all privileged instructions; output is a safe subset (mostly user-mode instructions).
  - **Adaptive**. Translated code is adjusted in response to guest behavior changes to improve overall efficiency.

VMWare’s Binary Translation

- While most instructions can be translated IDENT, there are several noteworthy exceptions:
  - **PC-relative addressing** cannot be translated IDENT since the translator output resides at a different address than the input. The translator inserts compensation code to ensure correct addressing. The net effect is a small code expansion and slowdown.
  - **Direct control flow**. Since code layout changes during translation, control flow must be reconnected in the TC. For direct calls, branches and jumps, the translator can do the mapping from guest address to TC address. The net slowdown is insignificant.
  - **Indirect control flow (jmp, call, ret)** does not go to a fixed target, preventing translation-time binding. Instead, the translated target must be computed dynamically, e.g., with a hash table lookup. The resulting overhead varies by workload but is typically a single-digit percentage.
  - **Privileged instructions.** We use in-TC sequences for simple operations. These may run faster than native: e.g., cli (clear interrupts) on a Pentium 4 takes 60 cycles whereas the translation runs in a handful of cycles (“vcpu.flags.IF:=0”). Complex operations like context switches call out to the runtime, causing measurable overhead due both to the callout and the emulation work.

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**.

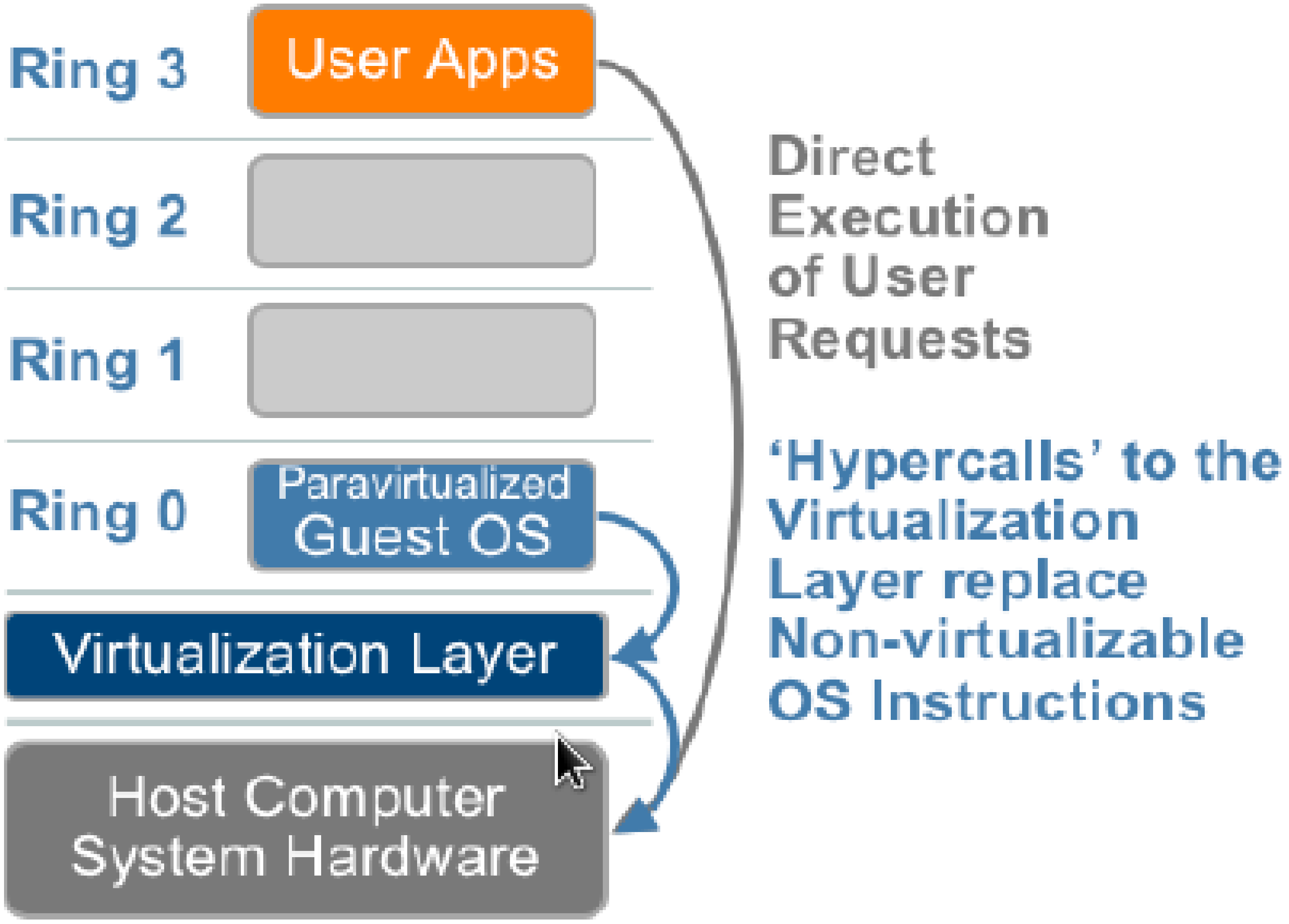

Ring 3
User Apps
Ring 2
Ring 1
Ring 0
Paravirtualized Guest OS
Virtualization Layer
Host Computer System Hardware
Direct Execution of User Requests
'Hypercalls' to the Virtualization Layer replace Non-virtualizable OS Instructions

Figure 6 – The Paravirtualization approach to x86 Virtualization

Lguest: hypervisor example

- lguest
  - paravirtualization within linux kernel
  - proof of concept (paravirt_ops, virtio...)
  - Research / teaching tool

guests and hosts

- A **module** (lg.ko) allows us to **run other Linux kernels** the same way we'd run processes.

- We only run specially **modified Guests.** Setting CONFIG_LGUEST_GUEST, compiles [arch/x86/lguest/boot.c] into the kernel so **it knows how to be a Guest at boot time**. This means that you can use the same kernel you boot normally (ie. as a Host) as a Guest.

- These Guests know that they **cannot do privileged operations**, such as disable interrupts, and that they have to ask the Host to do such things explicitly.

- Some of the replacements for such low-level native hardware operations call the Host through a "**hypercall**". [drivers/lguest/hypercalls.c]

VMI

- In 2005, VMware proposed a transparent paravirtualization interface, the **Virtual Machine Interface (VMI)**, as a standardized communication mechanism between the guest operating system and the hypervisor.
- Virtual Machine Interface is a **layer between the hypervisor and the paravirtualized guest OS**.
- **Transparent paravirtualization** is delivered as the same **guest OS kernel can run either natively on the hardware or virtualized** on any compatible hypervisor.
- It works by design as all VMI calls have two implementations, inline native instructions for bare hardware and indirect calls to a layer between the guest OS and hypervisor in a virtual machine.

VMI experimental support in Vmware products

- VMware has included **experimental support for VMI in Workstation 6 and VMware Player** that demonstrates support for paravirtualized operating systems in a hosted environment.
- The Player and Workstation release provides an implementation of the transparent paravirtualization interface discussed with the Linux community since 2005.
- This experimental support in VMware Player and Workstation is intended for developers who wish to evaluate paravirtualization technology from Vmware.
- Because this experimental support is **based on a hosted virtual machine architecture**, it is not intended to demonstrate I/O performance improvement for paravirtualization.
- **Subsequent implementations on a bare-metal hypervisor architecture**, such as the **VMware ESX Server**, will demonstrate improved CPU and I/O performance due to paravirtualization.

fixed ABI, binary blobs ? Kernel comunity doesn't like them

- VMI works by isolating any operations which may require hypervisor intervention into a special set of function calls.
- The implementation of those functions is **not built into the kernel**; instead, the kernel, **at boot time**, loads a "**hypervisor ROM**" which provides the needed functions.
- The binary interface between the kernel and this loadable segment is set in stone, meaning that kernels built for today's implementations should work equally well on tomorrow's replacement.
- This design also allows the same binary kernel image to run under a variety of hypervisors, or, with the right ROM, in native mode on the bare hardware.
- **The fixed ABI and ability to load "binary blobs" into the kernel does not sit well with all kernel developers**, however. It looks like another way to put proprietary code into the kernel, which is something most kernel hackers would rather support less of.

From VMI to paravirt_ops

- In 2006, **VMware released the VMI specification as an open specification**, and the VMI proposal at the Ottawa Linux Symposium led to the development of the paravirt-ops interface in the Linux community.
- The **paravirt-ops** interface, which was developed by a **joint team from IBM, VMware, Red Hat, and XenSource**, incorporates many of the concepts of VMI including the support of transparent paravirtualization.
- Using this interface, a paravirtualized Linux operating system will be able to run on any hypervisor that supports it.
- **paravirt-ops became a part of the official Linux kernel** starting with version 2.6.20.
- Version 2.6.22 also included the **VMI backend** to supplement the paravirt-ops interface (glue layer connecting paravirt_ops with the VMI binary interface). This design leaves the VMI people solely responsible for maintaining their ABI.

paravirt_ops

- In the end, **paravirt_ops is yet another structure filled with function pointers**. In this case, the operations are the various **machine-specific functions that tend to require a discussion with the hypervisor**.
- They include **things like disabling interrupts, changing processor control registers, changing memory mappings, etc**. As an example, one of the members of paravirt_ops is:

```
        void (fastcall *irq_disable)(void);
```

The patch also defines a little function for use by the kernel:

```
static inline void raw_local_irq_disable(void)
{
        paravirt_ops.irq_disable();
}
```

As long as the kernel always uses this function to disable interrupts, it will use whatever implementation has been provided by the hypervisor which fills in paravirt_ops.

paravirt_ops

- The patch includes a set of **operations for native** (non-virtualized systems) which causes the kernel to behave as it did before .
- That kernel may be a little slower, however, since **many operations which were performed by in-line assembly code are now, instead, done through an indirect function call**.
- **To mitigate the worst performance impacts**, the paravirt_ops patch set includes a **self-patching mechanism to fix up some of the function calls** - the interrupt-related ones, in particular.
- This interface may look a lot like VMI; both interfaces allow the replacement of important low-level operations with hypervisor-specific versions.
- **The difference** is that **paravirt_ops is an inherently source-based interface**, with no binary interface guarantees. It is assumed that this interface will change over time, as most other internal kernel interfaces do.

lguest_init(void)
[arch/x86/lguest/boot.c]

```
/* interrupt-related operations */
pv_irq_ops.init_IRQ = lguest_init_IRQ;
pv_irq_ops.save_fl = save_fl;
pv_irq_ops.restore_fl = restore_fl;
pv_irq_ops.irq_disable = irq_disable;
pv_irq_ops.irq_enable = irq_enable;
pv_irq_ops.safe_halt = lguest_safe_halt;

/* init-time operations */
pv_init_ops.memory_setup = lguest_memory_setup;
pv_init_ops.patch = lguest_patch;

/* Intercepts of various cpu instructions */
pv_cpu_ops.load_gdt = lguest_load_gdt;
pv_cpu_ops.cpuid = lguest_cpuid;
pv_cpu_ops.load_idt = lguest_load_idt;
pv_cpu_ops.iret = lguest_iret;
pv_cpu_ops.load_sp0 = lguest_load_sp0;
pv_cpu_ops.load_tr_desc = lguest_load_tr_desc;
pv_cpu_ops.set_ldt = lguest_set_ldt;

/* pagetable management */
pv_mmu_ops.write_cr3 = lguest_write_cr3;
pv_mmu_ops.flush_tlb_user = lguest_flush_tlb_user;
pv_mmu_ops.flush_tlb_single = lguest_flush_tlb_single;
pv_mmu_ops.flush_tlb_kernel = lguest_flush_tlb_kernel;
pv_mmu_ops.set_pte = lguest_set_pte;
```

hypercalls

- But first, **how does our Guest contact the Host to ask for privileged operations**? **There are two ways**: the direct way is to make a "hypercall", to make requests of the Host Itself. [include/asm-x86/lguest_hcall.h]

- Our hypercall mechanism uses the highest unused trap code (traps 32 and above are used by real hardware interrupts).

```
static inline unsigned long hcall(unsigned long call,  unsigned long arg1,
                                   unsigned long arg2, unsigned long arg3)
{
        asm volatile("int $" __stringify(LGUEST_TRAP_ENTRY)
                /* The call in %eax (aka "a") might be overwritten */
                : "=a"(call)
                  /* The arguments are in %eax, %edx, %ebx & %ecx */
                : "a"(call), "d"(arg1), "b"(arg2), "c"(arg3)
                  /* "memory" means this might write somewhere in memory.
                  * This isn't true for all calls, but it's safe to tell
                  * gcc that it might happen so it doesn't get clever. */
                : "memory");
        return call;
}
```

replace example with hcall: lguest_set_pte_at()

```
static void lguest_set_pte_at(struct mm_struct *mm, unsigned long addr,
                              pte_t *ptep, pte_t pteval)
{
        *ptep = pteval;
        lazy_hcall(LHCALL_SET_PTE, __pa(mm->pgd), addr, pteval.pte_low);
}
```

Devices virtualization

- **When running a virtual machine, the virtual environment has to present devices to the guest OS** – disks and network being the main two (plus video, USB, timers, and others). Effectively, this is the hardware that the VM guest sees.

- **Full virtualization** is a nice feature because it allows you to run any operating system virtualized. Now, if the guest is to be kept entirely ignorant of the fact that it is virtualised, **the host must emulate real hardware. This emulation is both complicated and inefficient.**

  - This is quite slow (**particularly for network devices and disks**), and is the major cause of reduced performance in virtual machines.

- Since in those environments, the driver is not aware that it is virtualized, **all instructions that it generates have to be trapped and emulated**.

What are paravirtual devices?

- However, if you are willing to let **the guest OS know that it's in a virtual environment**, it is possible to avoid the overheads of emulating much of the real hardware, and **use a far more direct path to handle devices inside the VM**.

- In this case, **the guest OS needs a particular driver installed which talks to the paravirtual device**. Under Linux, this interface has been standardised, and is referred to as the "**virtio**" interface.

- Paravirtualized drivers are not only available in paravirtualization, as **also in full (CPU) virtualization paravirtualized drivers can be used**.

- This drivers packs, **available for some operating systems**, usually makes it possible to use paravirtualized drivers for block and network devices. Since these devices cause the highest workload, **performance will increase enormously when these drivers are used**.

general virtual I/O subsystem Rusty's introduction:

- There are many different things which people mean when they talk about I/O.
  - There's guest userspace to guest userspace, guest devices served by the host and guest devices served by another guest.
  - There's device discovery, configuration, serving and guest suspend and resume.
- Aiming for common guest Linux driver code is an achievable short-term aim (ie. a platform-dependent "virtio" layer and common drivers above it).
- There are **four reasonable implementations**. (assuming some method of sending inter-guest interrupts):
  - **A shared page:** This is the simplest: copy in, copy out.
  - **A shared page containing descriptors:** The other end is privileged: it can read/write the memory referred to by the descriptors (eg. guest - host comms)
  - **Shared pages containing descriptors + hypervisor helper:** The other end can use a hypercall to say "copy the memory referred to by that descriptor" to/from itself. This means the descriptor page has to be read-only to the other side so the hypervisor can trust it.
  - **Full Xen-style grant table:** Mapping of arbitrary pages by the other side can be allowed (and revoked), and pages can be "given" to a willing recipient. This is controlled by a separate table, rather than being implied by the descriptors.

Virtio and virtual devices/drivers

- virtio: *Towards a De-Facto Standard For Virtual I/O Devices* (Rusty Russell)
- The **Linux Kernel currently supports at least 8 distinct virtualization systems**: Xen, KVM, VMware's VMI, IBM's System p, IBM's System z, User Mode Linux, lguest and IBM's legacy iSeries. It seems likely that more such systems will appear, and until recently **each of these had its own block, network, console and other drivers** with varying features and optimizations.
- Virtio is a **Linux standard for network and disk device drivers** where just the **guest's device driver "knows" it is running in a virtual environment, and cooperates with the hypervisor.**
- This enables guests to get **high performance** network and disk operations, and gives most of the performance **benefits of paravirtualization**.
- virtio is different, but architecturally similar to, Xen paravirtualized device drivers (such as the ones that you can install in a Windows guest to make it go faster under Xen). Also similar is VMWare's Guest Tools.

Virtio

- virtio **separates the guest Linux side of the driver from the code that is specific to the hypervisor implementation**. Each group of developers can maintain the code on their side of the API without changing the other, unless, of course, the virtio API itself needs to change

- The basic abstraction used by virtio is a "buffer", which consists of a struct scatterlist array. The array contains "out" entries describing data destined for the underlying hypervisor driver, as well as "in" entries for that driver to store data to return to the guest driver.

- This buffer abstraction encapsulates everything needed to communicate data to be written to or read from the hypervisor driver and, eventually, the underlying device. A guest driver, that uses the virtio interface, hands off buffers to the hypervisor driver and awaits their completion.

virtio

- At its core, the virtio API is a set of functions that are provided by the hypervisor driver to be used by the guest:

```
struct virtqueue_ops {
    int (*add_buf)(struct virtqueue *vq,
                   struct scatterlist sg[],
                   unsigned int out_num,
                   unsigned int in_num,
                   void *data);

    void (*sync)(struct virtqueue *vq);

    void *(*get_buf)(struct virtqueue *vq, unsigned int *len);

    int (*detach_buf)(struct virtqueue *vq, void *data);

    bool (*restart)(struct virtqueue *vq);
};
```

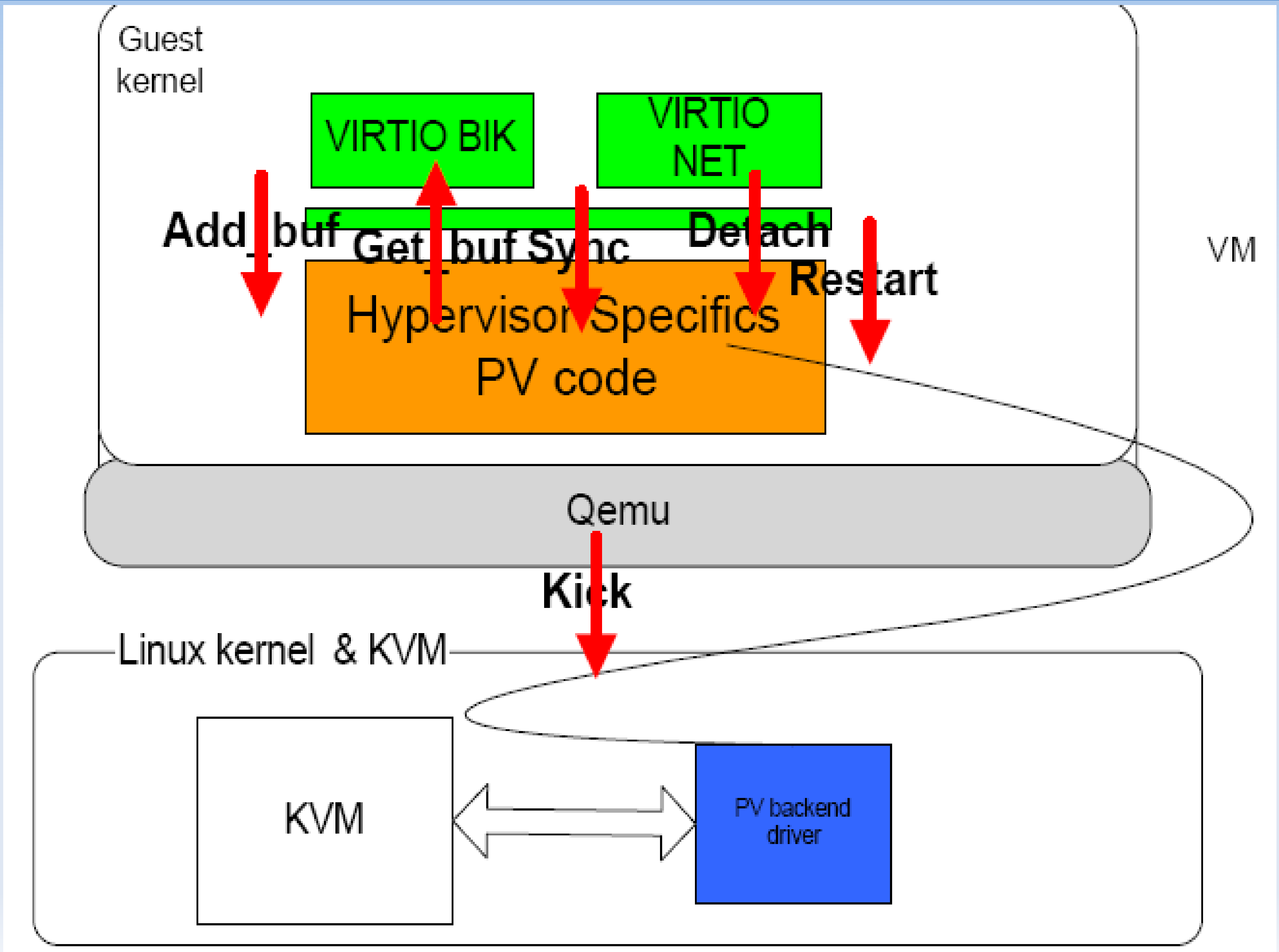
- This operations vector is initialized by the hypervisor and passed to the guest driver using a probe() function.
- The guest then sets up its data structures and registers with its kernel as a block or network device driver.

Virtio basic operation

- The basic operation uses add_buf() to register one or more buffers with the hypervisor driver.
- That driver is kicked via the sync() call to start processing the buffers.
- Each struct virtqueue has a callback associated with it which will be called when some buffers have completed.
- The guest then calls the get_buf() function to retrieve completed buffers. To support polling, which is used by network drivers, get_buf() can be called at any time, returning NULL if none have completed.
- The guest driver can disable further callbacks, at any time, by returning zero from the callback. The restart() routine is then used to re-enable them.
- Finally, the detach_buf() call is used during shutdown to cancel the operation indicated by the buffer and to retrieve it from the hypervisor driver.

Virtio and KVM

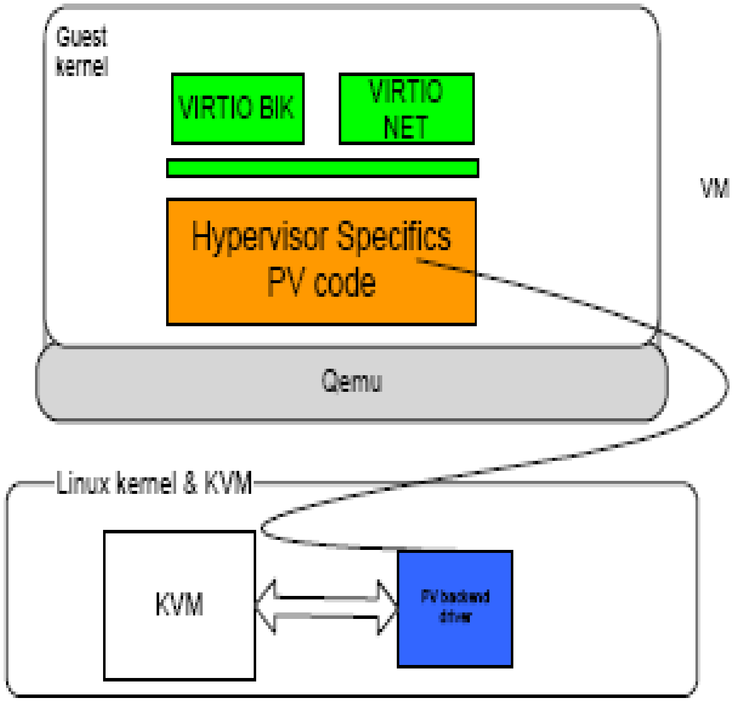
(source: ”KVM PV DEVICES”, dar.laor@qumranet.com)



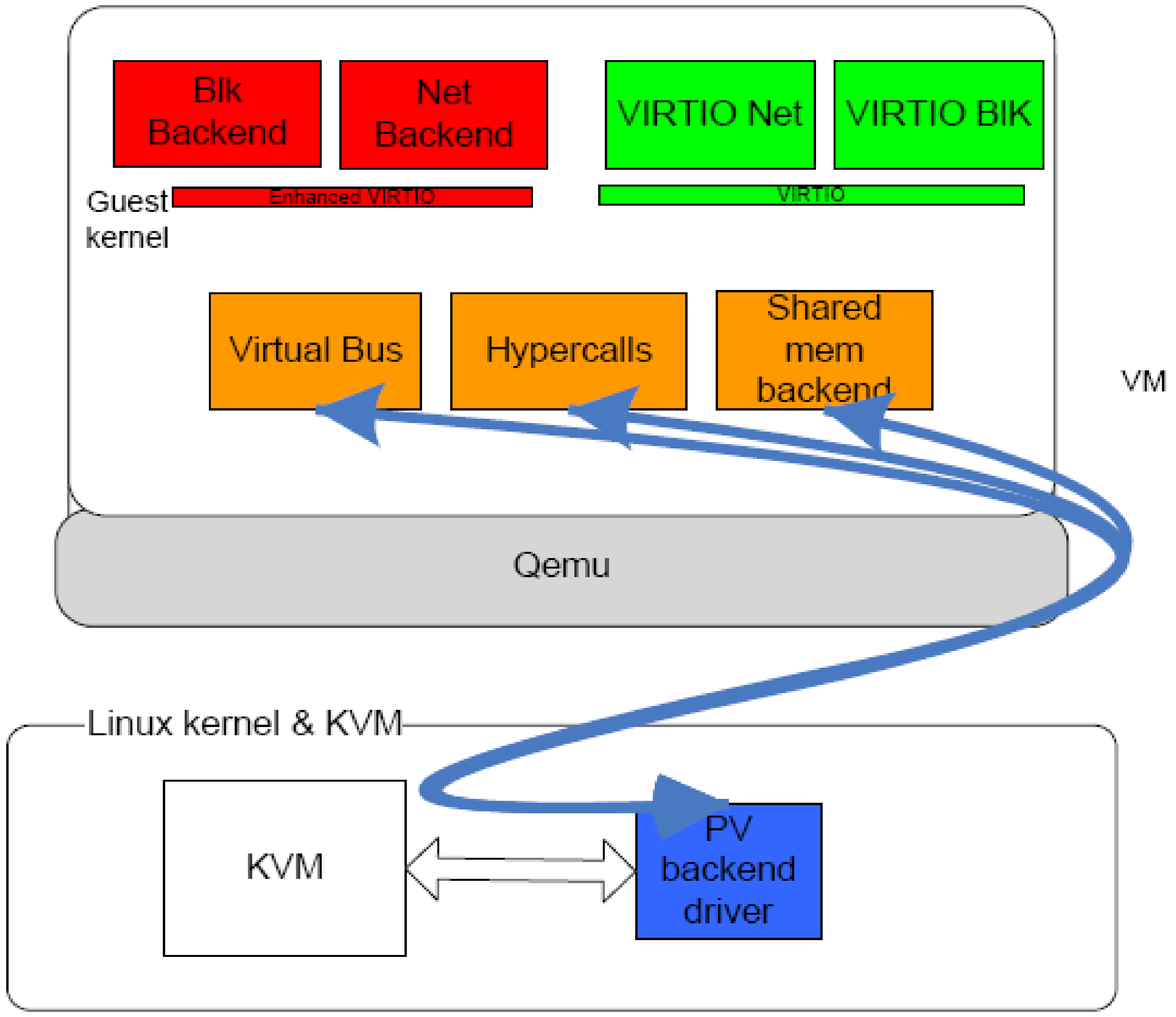
Hypervisor specifics

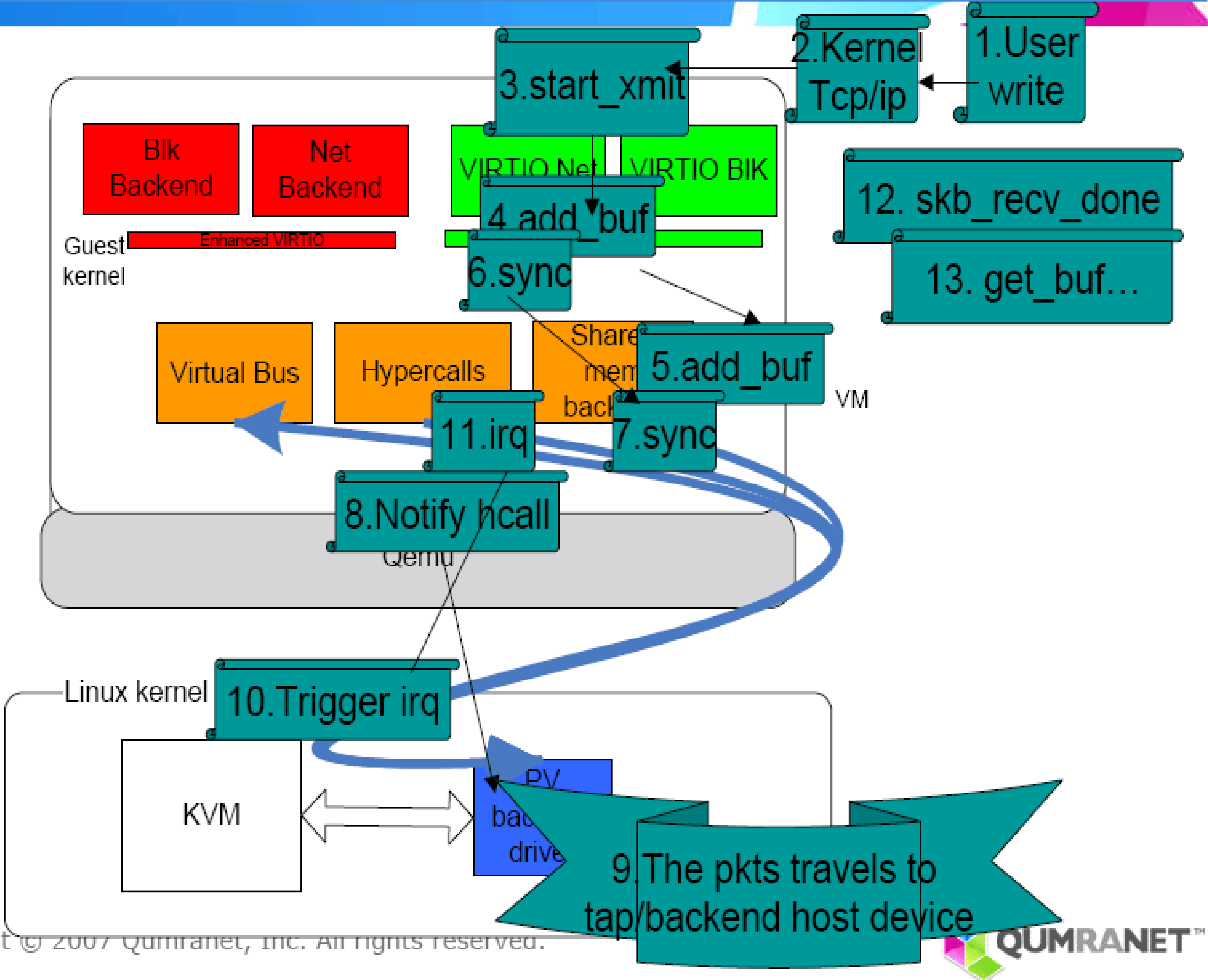
- The front end logic is implemented by VirtIO

- The backend needs
  - Probing & Bus services
    - Enumeration
    - Irq
    - Parameters (mac,..)
  - Shared memory with remote side
  - Hypercalls
  - Host driver/userspace backend



Enhanced virtio




1.User write
2.Kernel Tcp/ip
3.start_xmit
Blk Backend
Net Backend
VIRTIO Net
VIRTIO BlK
4.add_buf
Guest kernel
Enhanced VIRTIO
6.sync
12. skb_recv_done
13. get_buf…
Virtual Bus
Hypercalls
Shared mem back
5.add_buf
VM
11.irq
7.sync
8.Notify hcall
Qemu
Linux kernel
10.Trigger irq
KVM
PV back driver
9.The pkts travels to tap/backend host device

QUMRANET™

Technique 3 –
Hardware Assisted Virtualization

- In this section, we discuss recent **architectural changes that permit classical virtualization of the x86**.
- Hardware vendors are rapidly embracing virtualization and developing new features to simplify virtualization techniques. First generation enhancements include **Intel Virtualization Technology (VTX) and AMD's AMD-V (SVM)** 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.
- Extensions to x86-32 and x86-64
  - Allows classic **trap-and-emulate**
  - Hardware VM modes to reduce traps

Execution modes, VMCB and new virtualization primitives

- 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.
- 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.

AMD SVM and Intel VTX

- When running a protected mode guest, the VMM fills in a VMCB with the current guest state and executes vmrun.
- 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. 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.
- After **emulating the effect of the exiting operation** in the VMCB, the VMM again executes vmrun, returning to guest mode.
- Since most of 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.

Exits and performance

- **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.

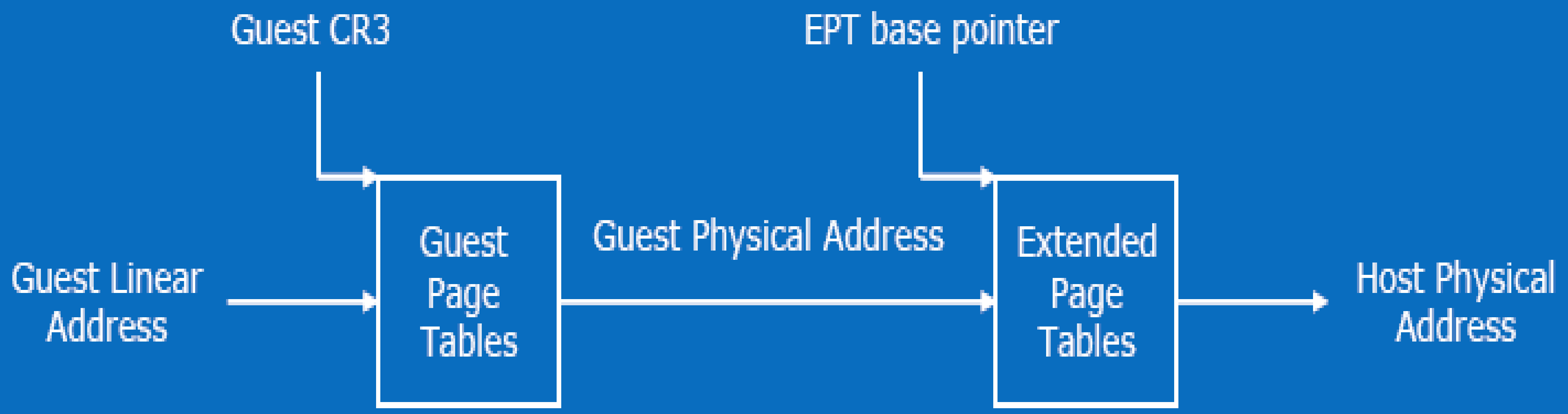
More performance and MMU

- 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.
- Note: AMD is now referring to NPT as “Rapid Virtualization Indexing”.

GLA to HPA

(source: Extending KVM with new IVT, sheng.yang@intel.com) kvm forum 2008

Intel® Extended Page Table



- Guest can have full control over Intel® 64 page tables / events
  - CR3, INVLPG, page fault

NTP / ETP

- 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 next slide figure. 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.**

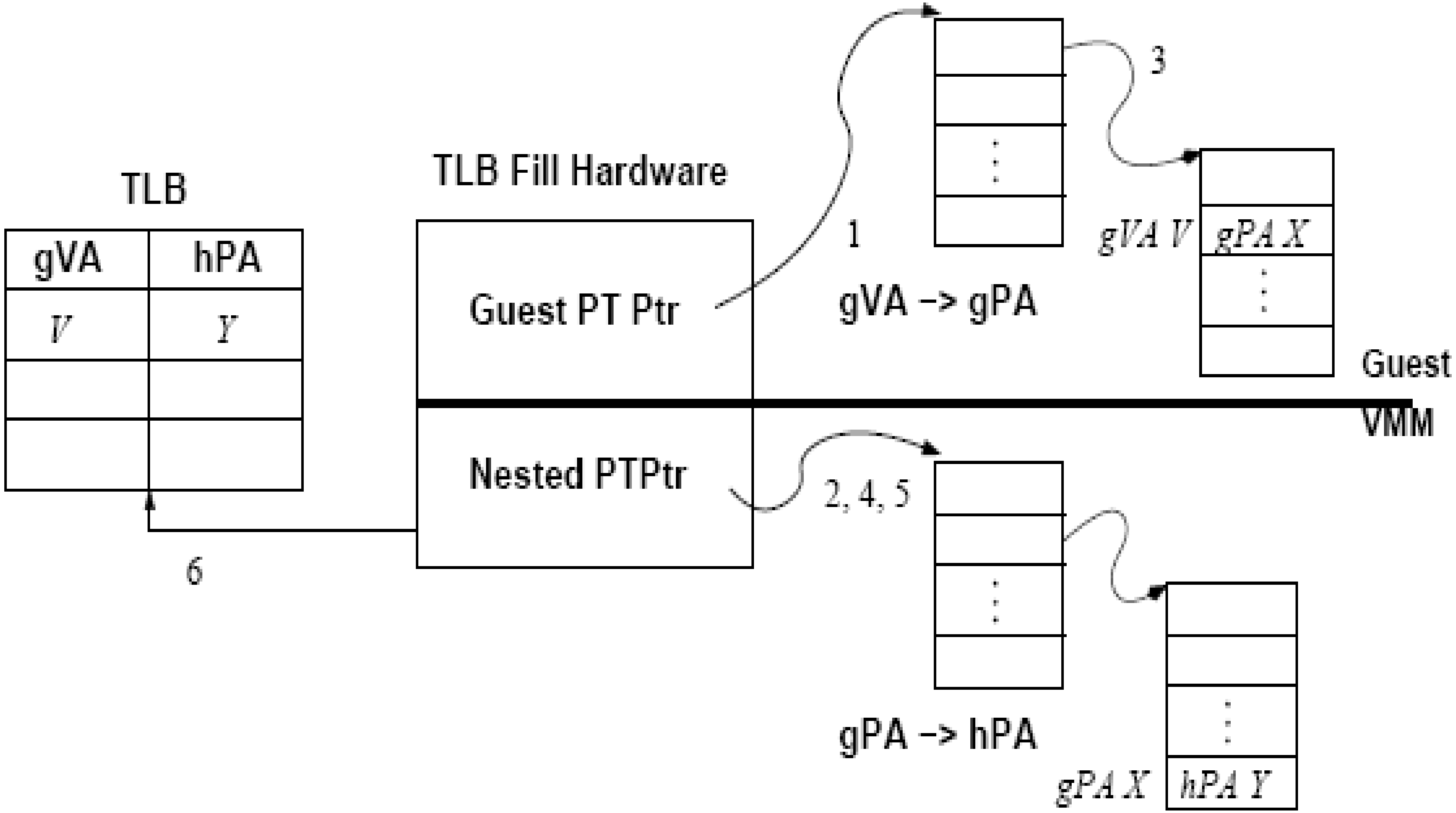
3. The hardware reads the guest page directory entry corresponding to guest virtual address V

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.

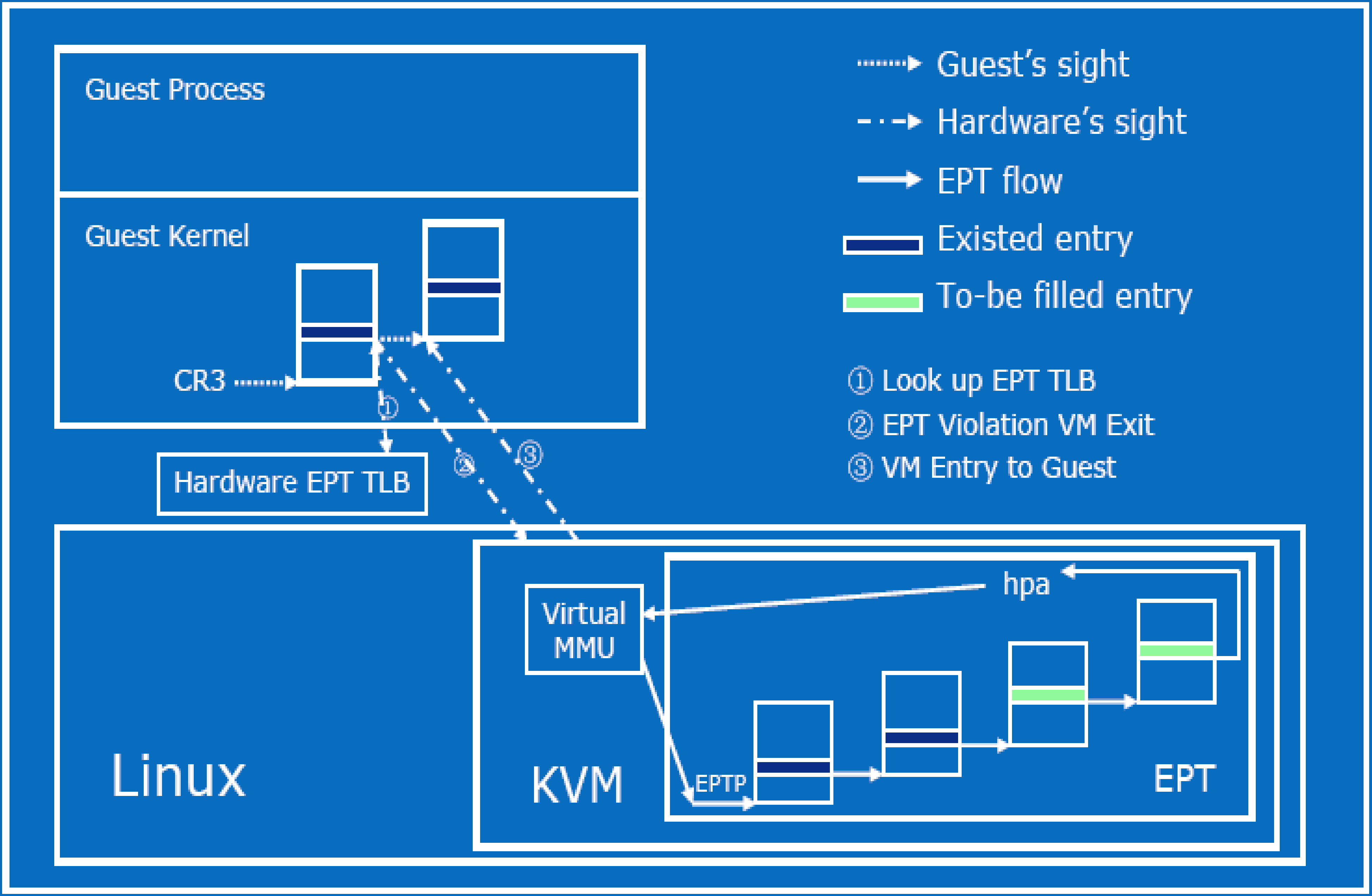
5. Having discovered the host physical address of the final level of the guest page table hierarchy, the hardware reads the guest page table entry corresponding to V . In our example, this PTE points to guest physical address X, which is translated via a third walk of the nested page table, e.g. to host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

Consider the case...



Filling EPT entry when EPT violate



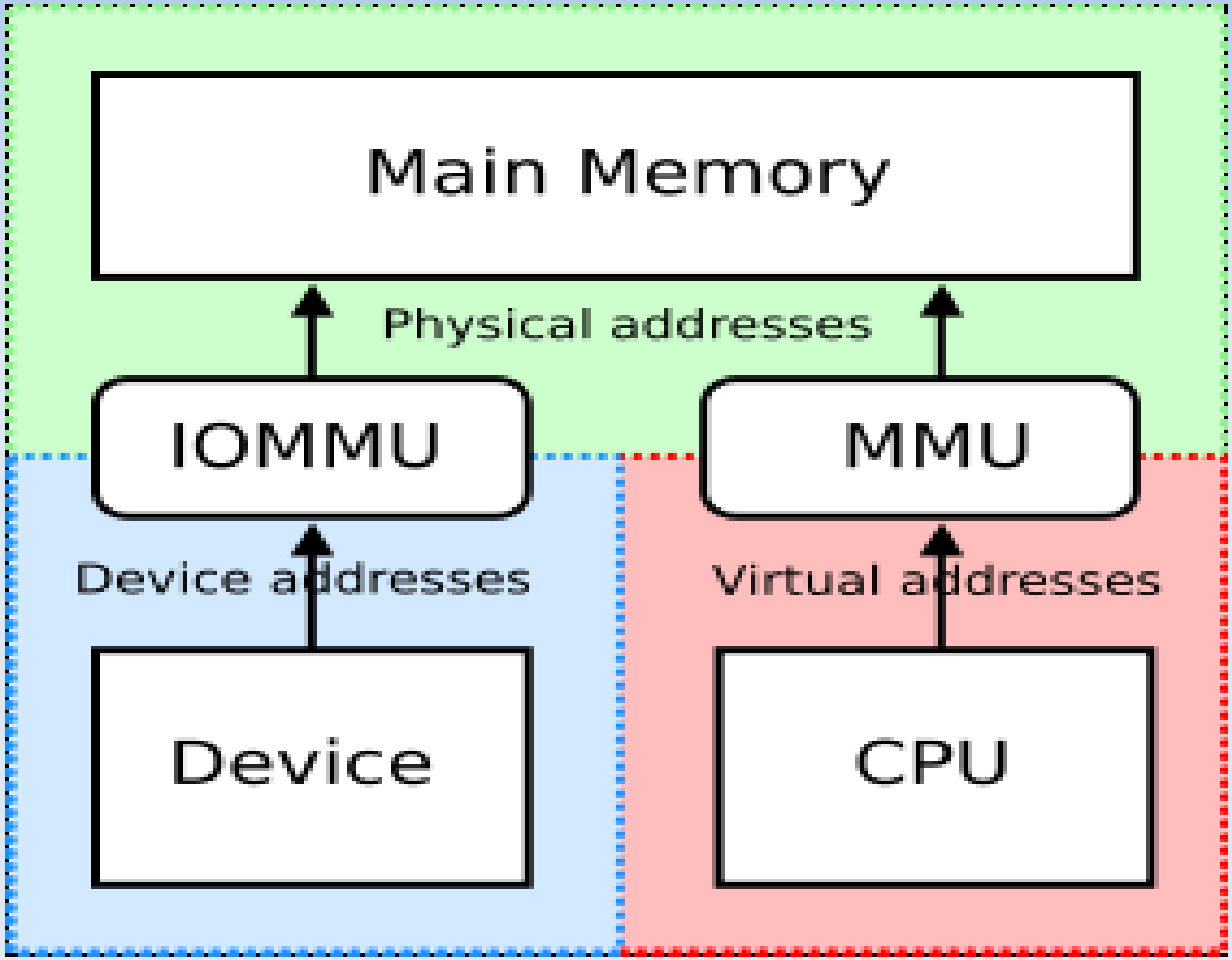
Nested pages performance

- 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.

IOMMU

- **input/output memory management unit (IOMMU)** is a memory management unit (MMU) that **connects a DMA-capable I/O bus to the main memory**.
- Like a traditional MMU, which translates CPU-visible virtual addresses to physical addresses, the IOMMU takes care of **mapping device-visible virtual addresses (also called device addresses or I/O addresses in this context) to physical addresses.** Some units also provide memory protection from misbehaving devices.
  - **AMD** has published a specification for **IOMMU** technology in the HyperTransport architecture.
  - **Intel** has published a specification for IOMMU technology as **Virtualization Technology for Directed I/O**, abbreviated **VT-d**.
  - Information about the **Sun** IOMMU has been published in the **Device Virtual Memory Access (DVMA)** section of the Solaris Developer Connection.
  - The **IBM Translation Control Entry (TCE)** has been described in a document entitled Logical Partition Security in the IBM eServer pSeries 690.
  - The **PCI-SIG** has relevant work under the terms **I/O Virtualization (IOV) and Address Translation Services (ATS).**

Comparison of IOMMU to the MMU



The advantages of having an IOMMU, compared to direct physical addressing of the memory, include:

- **Large regions of memory can be allocated without the need to be contiguous in physical memory** — the IOMMU will take care of mapping contiguous virtual addresses to the underlying fragmented physical addresses. Thus, the use of vectored I/O (scatter-gather lists) can sometimes be avoided.

- For **devices that do not support memory addresses long enough** to address the entire physical memory, the device can still address the entire memory through the IOMMU. This avoids overhead associated with copying buffers to and from the memory space the peripheral can address.

  - An **ordinary 32-bit PCI device simply cannot address the memory above the 4 GiB boundary**, and thus it cannot perform DMA to it. Without an IOMMU, the operating system is forced to implement time consuming double buffers (Windows nomenclature) also known as bounce buffers (Linux).

- **Memory protection from malicious or misbehaving devices**: a device cannot read or write to memory that hasn't been explicitly allocated (mapped) for it. The memory protection is based on the fact that OS running on the CPU exclusively controls both the MMU and the IOMMU. The devices are physically unable to circumvent or corrupt configured memory management tables.

Performance penalties

- The **disadvantages of having an IOMMU**, compared to direct physical addressing of the memory, include:[7]
  - Some degradation of performance from **translation and management** overhead (e.g., page table walks).
  - Consumption of **physical memory for the added I/O page (translation) tables**. This can be mitigated if the tables can be shared with the processor.
  - *”The Price of Safety: Evaluating IOMMU Performance”, 2007 Ottawa Linux Symposium, Volume One.*

IOMMU in relation to virtualization

- Higher performance hardware such as **NICs use DMA** to access memory directly; When an **operating system is running inside a virtual machine**, including systems that use paravirtualization, **it does not usually know the physical addresses** of memory that it accesses.

- This makes providing **direct access to the computer hardware difficult**, because if the OS tried to instruct the hardware to perform a direct memory access (DMA), it would likely corrupt the memory, as **the hardware does not know about the mapping between the virtual and real addresses used by the virtualized guest system**.

- The **corruption is avoided because the hypervisor or OS intervenes in the I/O operation to apply the translations**; unfortunately, this delays the I/O operation.

- An IOMMU can solve this problem by **re-mapping the addresses accessed by the hardware according to the same (or a compatible) translation table used by the virtual machine guest.**

- *"Intel Virtualization Technology for Directed I/O"*, Inte Technology Journal, Volume 10, Issue 3

Conclusion

- Virtualization techniques will be present as a **standard abstraction layer** in the modern data center stack
- Some of the **concepts we have studied in the past should be updated**:
  - Computer architectures
  - Operating Systems
  - Networking
  - Security
  - Performance
  - Etc...
- The final solution to virtualization seems to be a mix of the ones studied here:
  - ”Tomorrow’s virtualization likely involves vendor-supported **paravirtualized OSes** that are installed into industry standard disk format files and **able to run either natively or on a variety of compatible and interchangeable hypervisors** that take advantage of **hardware assisted** management of the CPU, memory and I/O devices.”