



Virtualization

Jana Giceva

jgiceva@doc.ic.ac.uk

Department of Computing
Imperial College London

<http://lsds.doc.ic.ac.uk>

Intro to Virtualization

Why do we need virtualization?

Recall previous lecture...

What is virtualization?

Broadly speaking...

Virtualization: simulate or emulate a function of a resource (object) in software, identical to that of the physical one.
Abstraction to make software look and behave like hardware.

Recall – virtual memory

Similarly, virtualization techniques can be applied to other IT infrastructure layers, including networks, storage, laptop or server hardware, OSs and applications.

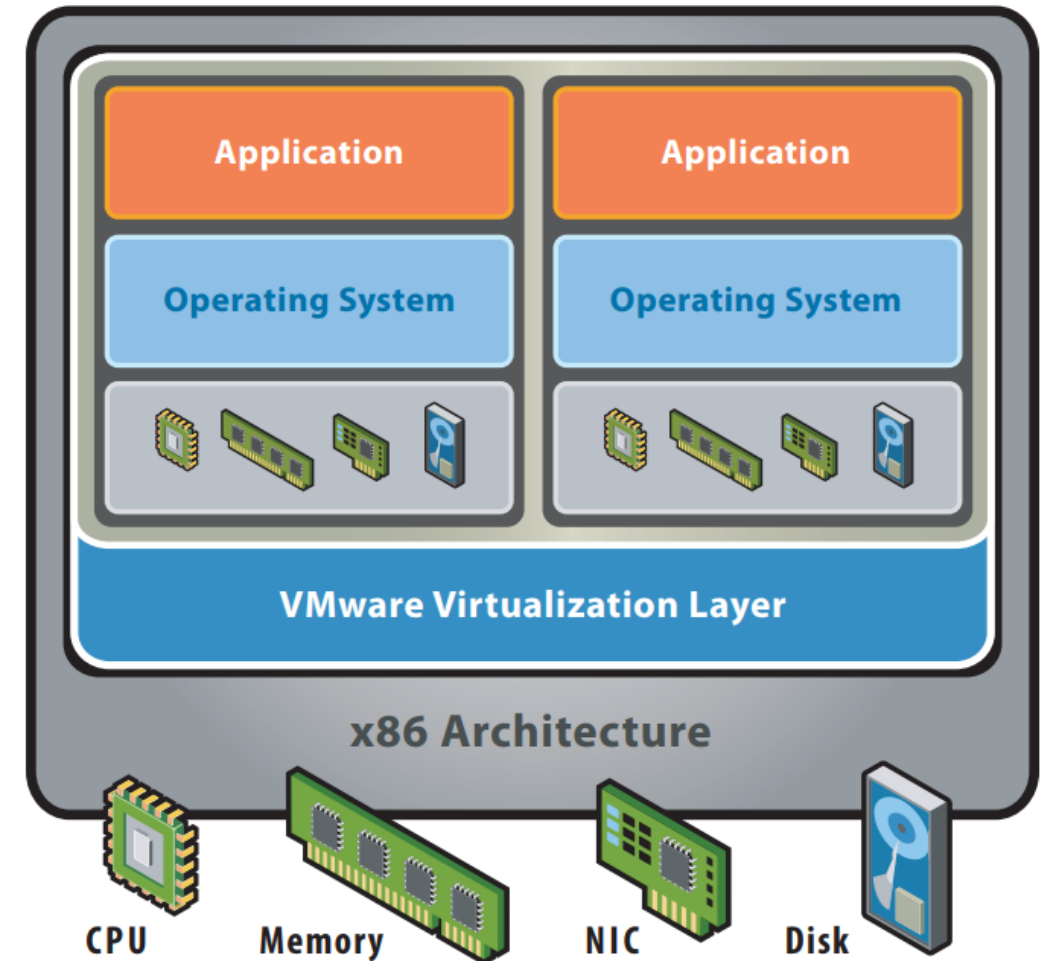
Virtual infrastructure

Allows multiple VMs with different OSs and applications to run in isolation, side by side on the same physical machine.

VMs can be provisioned to any system.

Machine independence of OS and applications.

Can manage an application and its OS as a unit, encapsulating them in a single VM.



src: VMWare whitepaper -- Virtualization

Properties of Virtual Machines

Key properties of VMs:

1. Partitioning

- Run multiple OS on one physical machine
- Divide system resources between VMs

2. Isolation

- Provide fault and security isolation at the hardware level
- Preserve performance with advanced resource controls

3. Encapsulation

- Save the entire state of a VM to files
- Move and copy VMs as easily as copying and moving files

4. Hardware independence

- Provision or migrate any VM to any physical server

Virtualization – Definitions and Terms

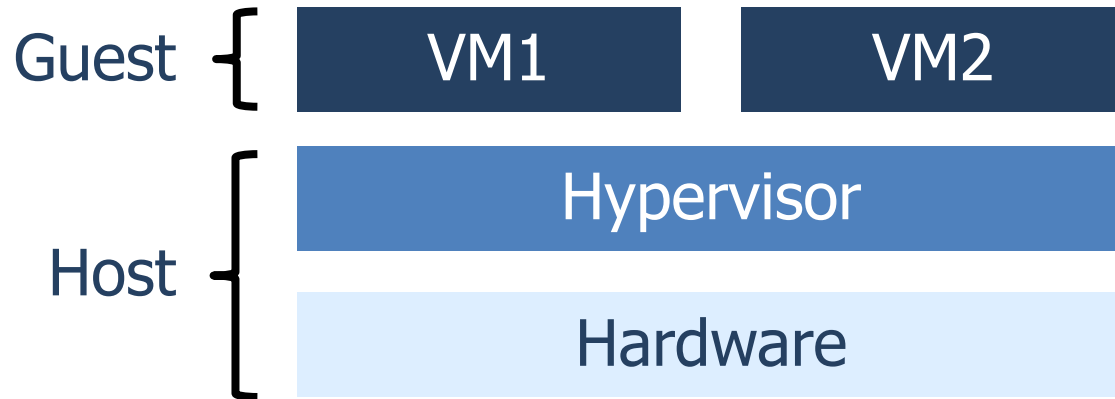
Virtual Machine (VM): a tightly isolated software container with an operating system (OS) and an application inside.

Virtual Machine Monitor (VMM): entity responsible for virtualizing a given architecture, including the ISA, memory, interrupts, and basic I/O operations.

Hypervisor: combines an operating system (OS) with a VMM. It dynamically allocates computing resources to each VM as needed.

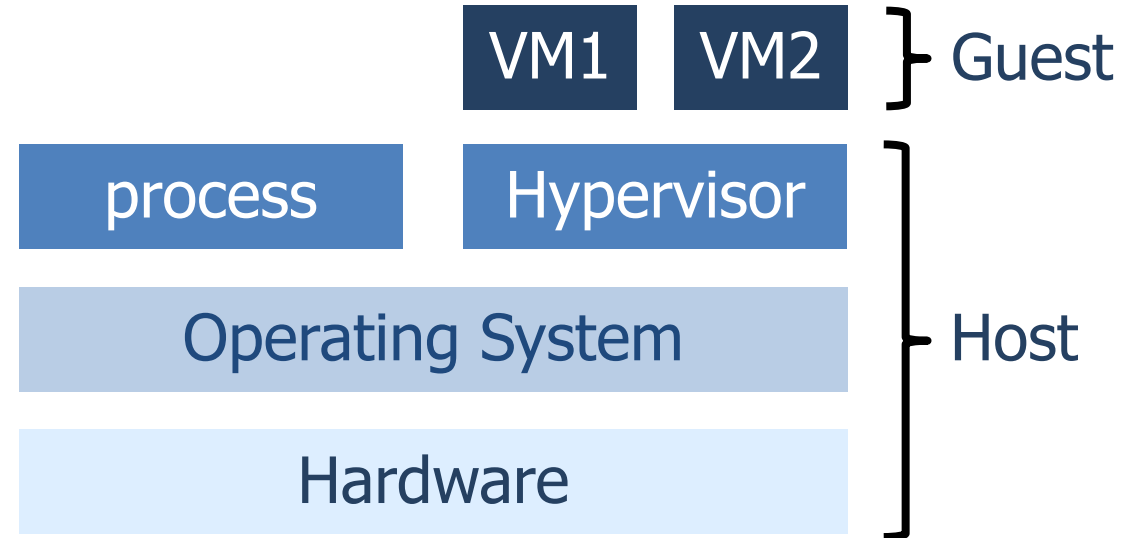
Types of Virtualization

Para ("bare-metal") Virtualization



- e.g., VMware ESX, Microsoft Hyper-V, Xen

Full ("hosted") Virtualization



- e.g., VMware Workstation, Microsoft Virtual PC, Sun VirtualBox, QEMU

Types of Virtualization

Para ("bare-metal") Virtualization

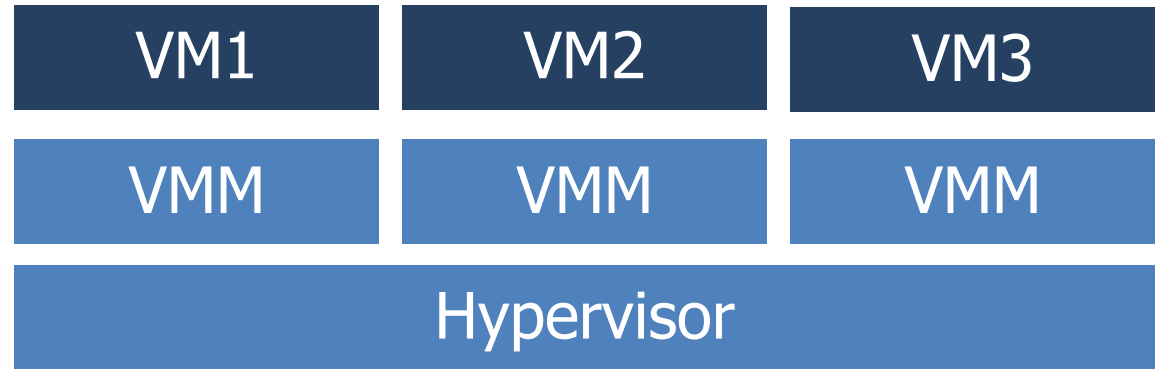
- The guest OS knows it is being virtualized
- The guest OS participates in the process
- Give up the illusion of running on bare-metal for higher performance
- Performance within few percent of unvirtualized

Full ("hosted") Virtualization

- The guest OS is unaware of being virtualized
- Does not require any modification of the guest operating system
- It works with all OSs written to the same ISA
- Guest thinks it runs on bare-metal

How does Virtualization work?

Virtual Machines are like processes.



The hypervisor is like the kernel.

It schedules the VMs, allocates memory for them, multiplexes their I/O, etc.

But the VMs are full blown OSs that assume they fully “own” the computer hardware!

Question: How to run a VM and virtualize the OS?

Solution: The hypervisor must fake it – make it appear as though the guest OS controls the hardware

How can the hypervisor do that?

There are a few alternatives:

1. Emulation
2. Trap-and-emulate
3. Dynamic binary translation
4. Para-virtualization

Need to virtualize: (1) the CPU, (2) memory, (3) storage and (4) network I/O.

Today's focus on: CPU virtualization!

1. Emulate

Do what the CPU does, but in software

- **Fetch** the next instruction
- **Decode** (is it and ADD, a XOR, a MOV?)
- **Execute** (using the software emulated registers and memory)

For example:

```
– addl %ebx, %eax           /* eax += ebx */
```

Is emulated as:

```
– enum {EAX=0, EBX=1, ECX=2, EDI=3, ...};  
  unsigned long regs[8];  
  regs[EAX] += regs[EBX];
```

Pros: it is very simple!

Cons: It is very slow! 😊

What needs to be emulated?

CPU and memory

- Register state
- Memory state

Memory Management Unit

- Page tables, segments

Platform

- Interrupt controller, timer, buses

BIOS

Peripheral devices

- Disk, network interface, etc.

2. Trap-and-Emulate: Background

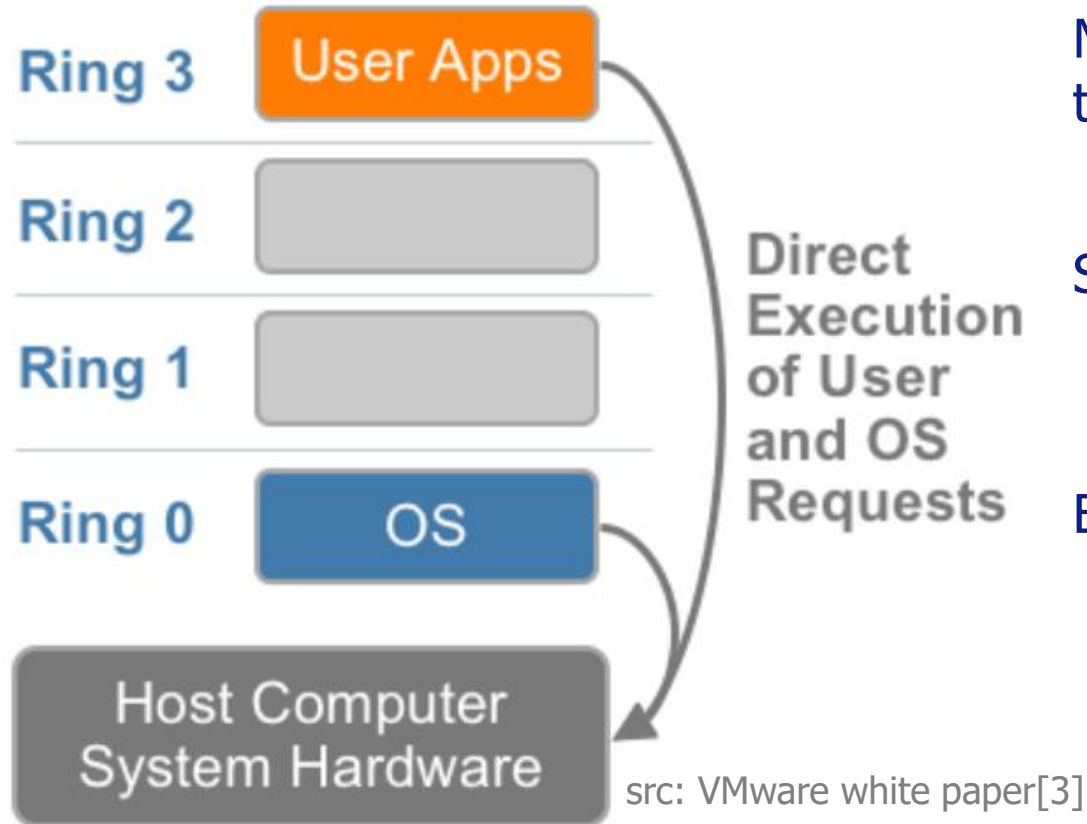
Let's recall some basics from Operating Systems class.

Kernel mode and user mode in stock operating systems?

What is a trap?

Limited Direct Execution?

2. Trap-and-Emulate



Most of the virtual machine code can execute directly on the CPU

- E.g., `addl %ebx, %eax`

So instead of emulating this code, run it directly.

But, some operations are sensitive and need intervention

- Any instruction that generates an interrupt and requires Ring 0
- Guest must not be allowed to run in privileged mode! Why?

Idea: Trap-and-emulate the sensitive instructions.

Leverage the fact that many sensitive ops trigger an interrupt when performed by unprivileged user-mode software.

2. Trap-and-Emulate Problem on x86

Some instructions (sensitive) read or update the state of the VM and do not trap

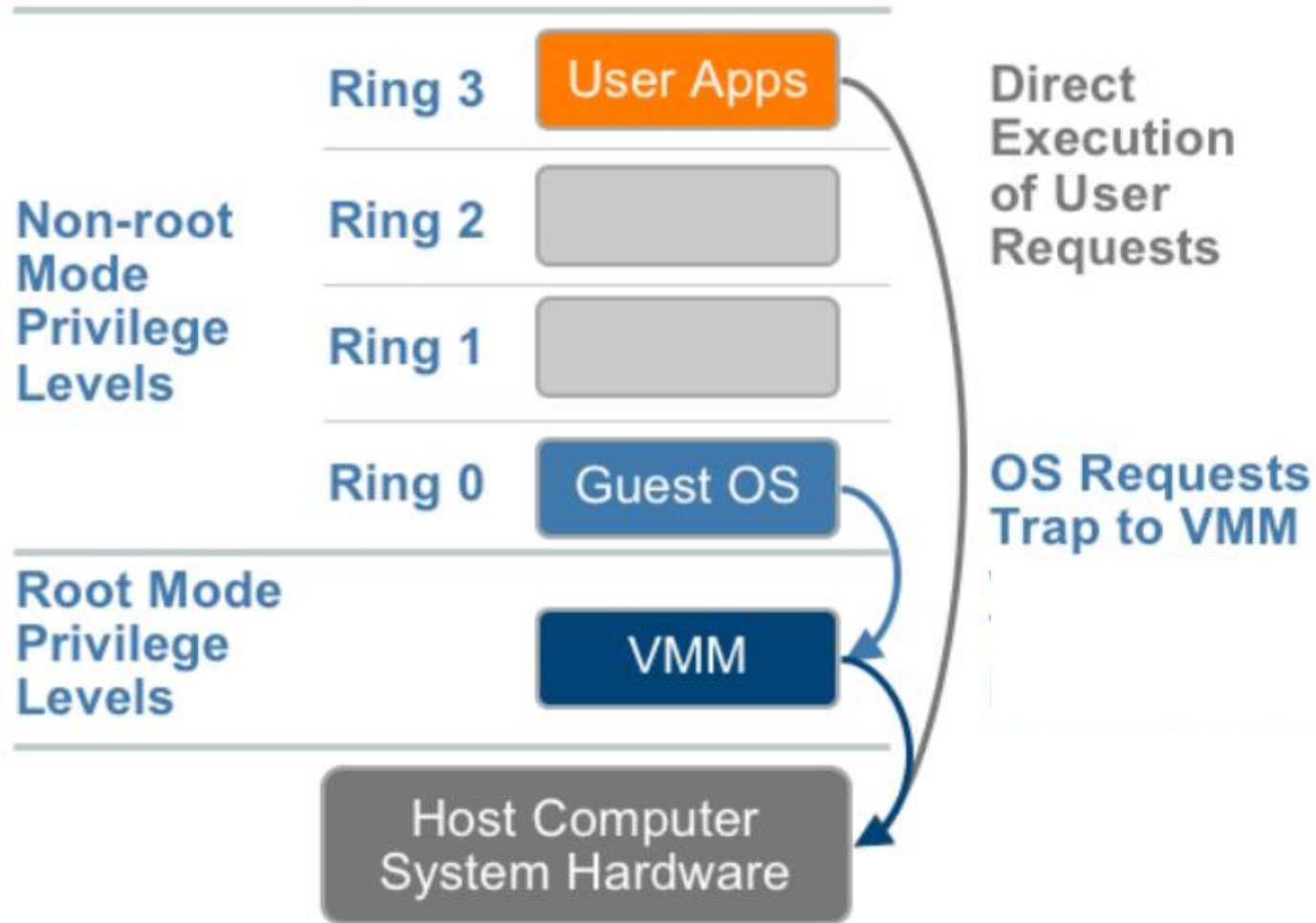
- 17 sensitive, non-privileged instructions

Group	Instructions
Access to interrupt flag	<code>pushf, popf, iret</code>
Visibility into segment descriptors	<code>lar, verr, verw, lsl</code>
Segment manipulation instructions	<code>pop <seg>, push <seg>, mov <seg></code>
Read-only access to privileged state	<code>sgdt, sltd, sidt, smsw</code>
Interrupt and gate instructions	<code>fcall, longjump, retfar, str, int <n></code>

Examples

- `popf` does not update the interrupt flag (IF)
 - So it is not possible to detect when guest disables interrupts
- `push %cs` can read code segment selector (`%cs`) and learn its CPL
 - Guest gets confused

Solution for Trap-and-Emulate on x86: add HW support



Hardware support for virtualization
(modern chips rectify the problem)

Hypervisors can configure which
operations would generate traps

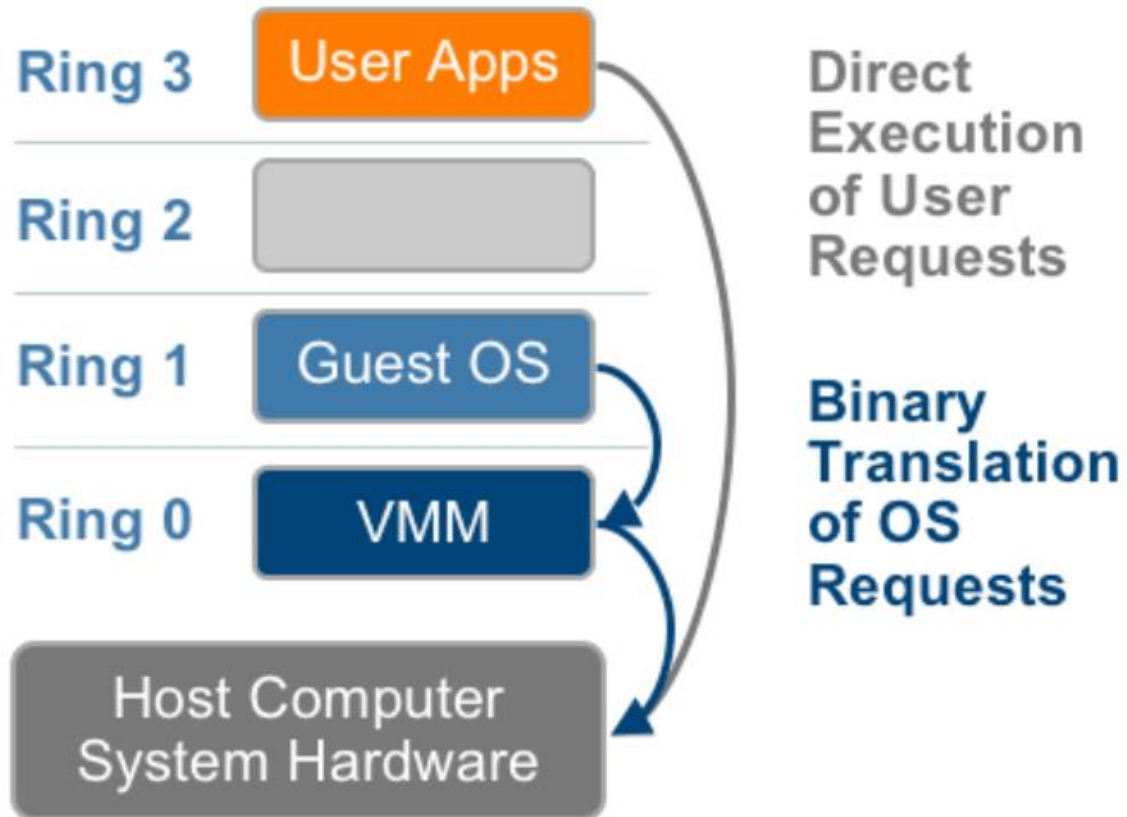
Intel extension VT-x

AMD-V virtualization support

Example hypervisor: KVM

src: VMware white paper[3]

3. Dynamic binary translation



src: VMware white paper[3]

Idea: take a block of VM operations and translate it on-the-fly to “safe” code

Similar to JIT-ing

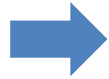
Translation rules:

- Most code translates identically – direct execution of user requests
- Sensitive operations are translated into **hypercalls**
- Calls to the VMM to ask for service
- Implemented as trapping instructions
- Similar to syscall

Binary Translation I

Input

X86 instructions (full ISA)



Output

X86 instructions (restricted subset of the ISA)

Dynamic translation on-demand

If an instruction is not executed, it will not be translated

No assumptions are made about the instructions

Process:

- X86 hex code -> IR code -> Translation Unit (TU) – 12 instructions or a terminating instruction
- Translate privileged instruction into operation on VMM
- Since we change code, the `jmp` instruction won't work, need to translate those as well

Result:

- Translating a translation unit (TU) results in a compiler code fragment (CCF)

Binary Translation: Example


```
push %ebx           ; callee saved
mov  %eax, %edx      ; %edx = %eax = lock
cli                ; disable interrupts
mov  $1, %ecx        ; %ecx = 1
xor  %ebx, %ebx      ; %ebx = 0
jmp  doTest
```



```
push %ebx           ; callee saved
mov  %eax, %edx      ; %edx = %eax = lock
and  $0xfd, %gs:vcpu.flags ; disable interrupts
mov  $1, %ecx        ; %ecx = 1
xor  %ebx, %ebx      ; %ebx = 0
jmp  [doTest]
```

The segment register `gs` provides an escape into VMM-level data structures

The `cli` translation must clear the virtual interrupt flag, which exists as part of an in-memory image of the guest's virtual CPU



So we use an `and` instruction.

As the translator does not preserve code layout, all control flow instructions must use non-IDENT translations.

Binary Translation II

Optimization 1: To speed-up inter-CCF transfers -> use a “chaining” optimization

- Allowing CCF to jump directly to another without calling the translation cache
- Often possible to elide chaining jumps and fall through from one CCF to another.

Optimization 2: Populate the translation cache (TC)

- Fill in the TC gradually with the guest’s working set.
- Hence, the interleaving of translation and execution decreases as the TC is filled up.

e.g., Booting Windows XP Professional and then immediately shutting it down translates 933,444 32-bit TUs and 28,339 16-bit TUs

But, translating each unit takes 3ms, for a total translation time of 3s.

3. Dynamic binary translation (cont.)

Pros

- No hardware support is needed
- Performance is much better than emulation

Cons

- Performance is still worse than trap-and-emulate
- Non trivial to implement
- The VMM needs on-the-fly x86-to-x86 binary compiler
- Think about the challenge of betting the branch target addresses right

Examples

- VMWare (x86 32bit)
- QEMU
- Microsoft Virtual Server

History of Virtual Machines

Trace their roots back to a small number of mainframes from the 1960s – most notably the IBM 360/67 – and became popular in the mainframe world during the 1970s.

With the introduction of Intel's 386 in 1985, VMs took up residence in the microprocessors for personal computers.

X86 widely believed not suitable for virtualization even as late as 2000

- Main reason: x86 could not generate the required traps

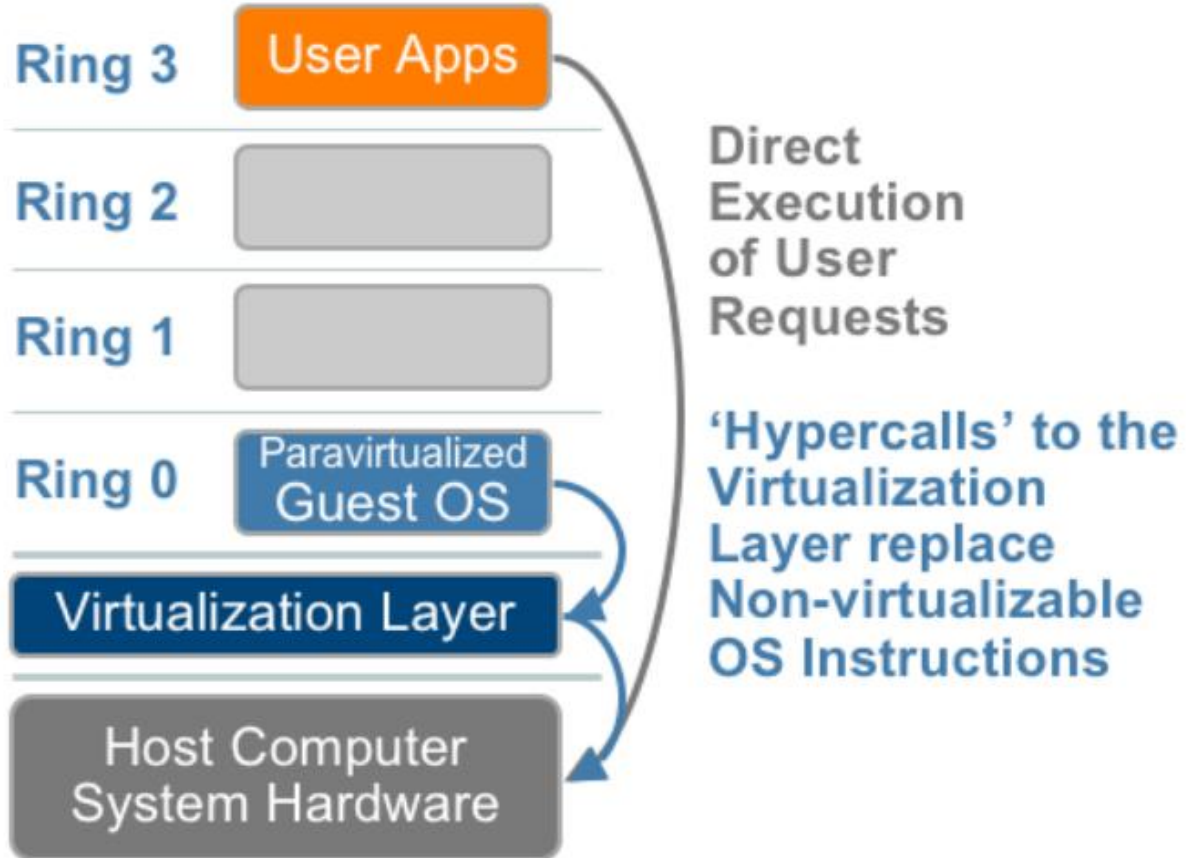
Until VMware engineers succeeded using binary translation

2002 – VMWare virtualizes multiprocessors

2005 – Intel and AMD introduce hardware features that support virtualization

Para virtualization

4. Para-virtualization



src: VMware white paper[3]

So far the guest OS was unmodified

Alternative, paravirtualization

- Requires guest OS to **know** it is being virtualized
- And to explicitly use the hypervisor services through a hypercall

Pros:

- No hardware support is needed
- Performance is almost as good as with HW ext.

Cons:

- Requires guest OS modifications
- Same OS cannot run in the VM and on bare-metal

Example: Xen

Full vs para virtualization

Hosted virtualization (or full virtualization) has significant performance overhead due to all the trap-and-emulate crossings that happen

Before hardware support for virtualization was introduced, **binary translation** was the only practical solution, but even that one had performance overheads

Para-virtualization:

- Give up the illusion of each virtual machine running on bare-metal in exchange for better performance
- Main appeal: performance within a few percent of unvirtualized case

Example para-virtualized hypervisor: Xen

Xen – the most famous para-virtualization hypervisor

Developed at the University of Cambridge

Released in 2003

Maintained as open source

Based on the exokernel architecture

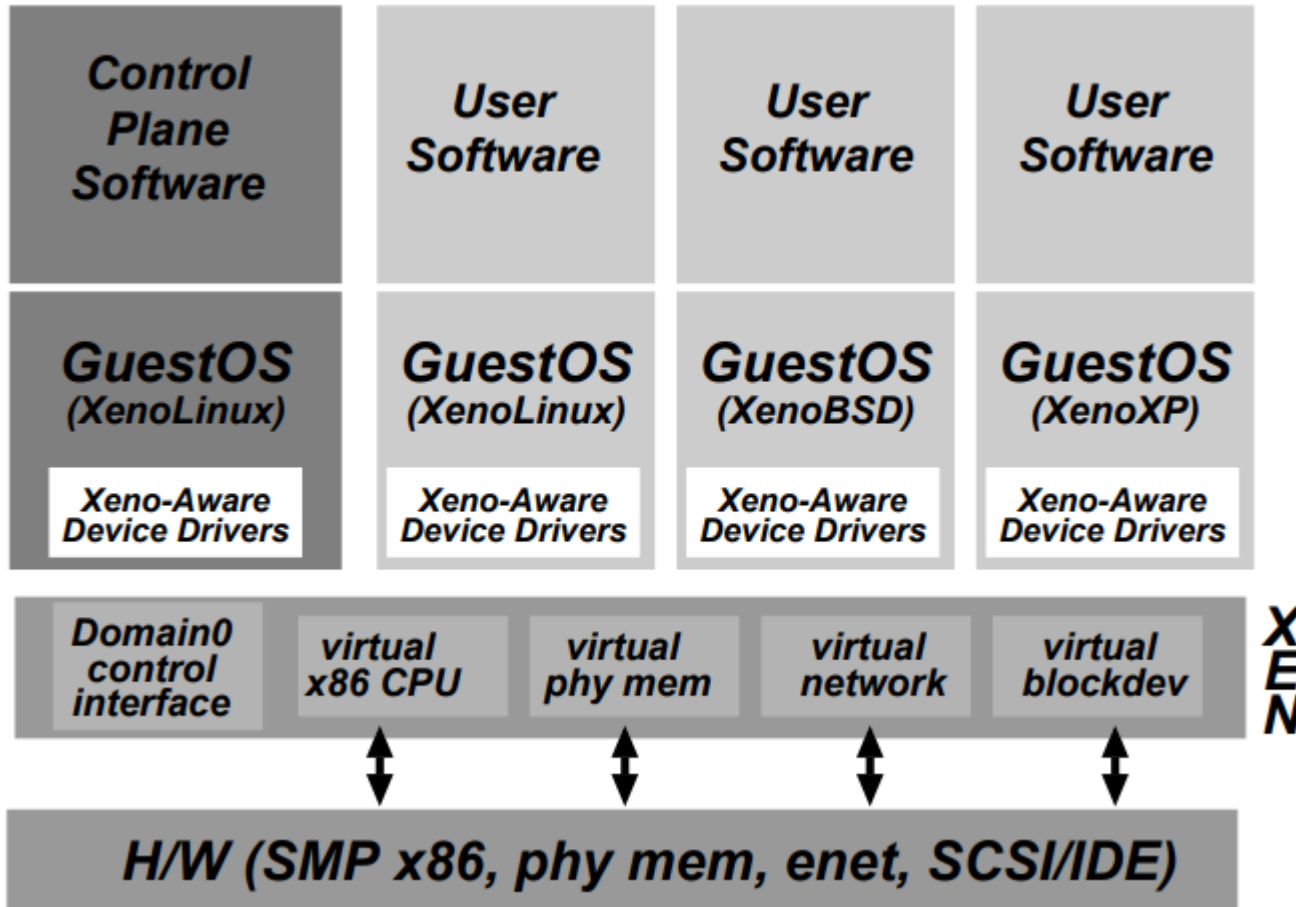
"Exokernel: An Operating System Architecture for Application-Level Resource Management"
Engler, Kaashoek, O'Toole Jr. (SOSP'95) ([link](#))

Requirements for the Xen hypervisor

Three main requirements that Xen tries to satisfy:

1. VMs should be isolated
2. A variety of OS should be supported
3. High Performance -> this is where Xen excels
 - Co-design with the VM OS
 - Optimize the OS to run in a virtualized environment
 - Maintain ABI – applications stay the same

Xen Hierarchy



Divides the machine into “domains”

- Each virtual machine runs on one domain
- The hypervisor is managed from domain 0 (or dom0)
- Dom0 runs para-virtualized operating system
- Other domains can run para-virtualized operating system (at nearly full performance)

src: Xen – The art of Virtualization paper

Properties of Xen

Main properties of Xen and para-virtualization:

Without HW support, each operating system has to be ported to Xen

- But, while the OS has to be changed, the applications inside the OS do not have to be changed
- Changing the OS is significantly easier than doing the JIT binary translation

Para-virtualized (PV) OS knows it does not run on bare-metal

- Instead of privileged instructions, will make hypercalls
- Hypercalls are just calls to the hypervisor

Intel and AMD contributed to Xen to support their virtualization extensions

- To create so-called Hardware Virtual Machines (HVM)
- With HVM, unmodified OS can run on Xen.

Strengths of the approach

Para-virtualization is particularly useful for I/O and networking

PV virtual machines can directly use specialized device drivers through dom0

HVM virtual machines have to emulate the hardware (pay performance cost)

More on this when we cover storage and networking virtualization on Thursday

Virtual Machine (para-virtualization) Interface (VMI)

In 2005, VMware proposed a transparent para-virtualization interface, the VMI:

A standard interface would help different hypervisors and guests to communicate

Developed by consortium of companies

The OS is written to the VMI interface

- The VMI calls can be implemented using native code when the OS runs on bare-metal
- The VMI calls can be implemented using VMware or Xen-specific code for virtualization

Modifications to the operating system

The main modification that is required of the operating system, is to execute in x86 ring 1 instead of ring 0 (highest privilege).

System calls, page faults, and other exceptions always trap to Xen

- Xen then redirects to dom0 (if required), does the work, and returns back to faulting VM
- Clearing interrupts, etc. can be done directly by Xen
- Redirection usually done when IO is involved –
dom0 will have much better device drivers than Xen itself

System calls have a “fast track” path

- VM can register a function to be called on system call
- This function is validated by Xen
- The system call then no longer needs to trap to Xen every time

Xen – implementation details

The Xen hypervisor is mapped into the top 64MB of each VM

- When doing hypercalls, there is no need to context switch
- If this was not done, the TLB would need to be flushed on each hypercall

Device IO is performed by Xen

- Translated to each domain using shared memory

VM Page Tables are directly read by the virtual machine

- But updates have to go through Xen (through hypercalls)
- Each page table needs to be registered with Xen

Xen – implementation details II

Interrupts are replaced by Xen events

- Each VM has a bitmap indicating whether an event has occurred
- Xen updates these bitmaps
- Xen can also call a handler that is registered by the VM

Communication between Domains and Xen

- Domain to Xen: synchronous hypercalls
- Xen to Domains: async events

Memory is statically partitioned among domains

Xen's original performance

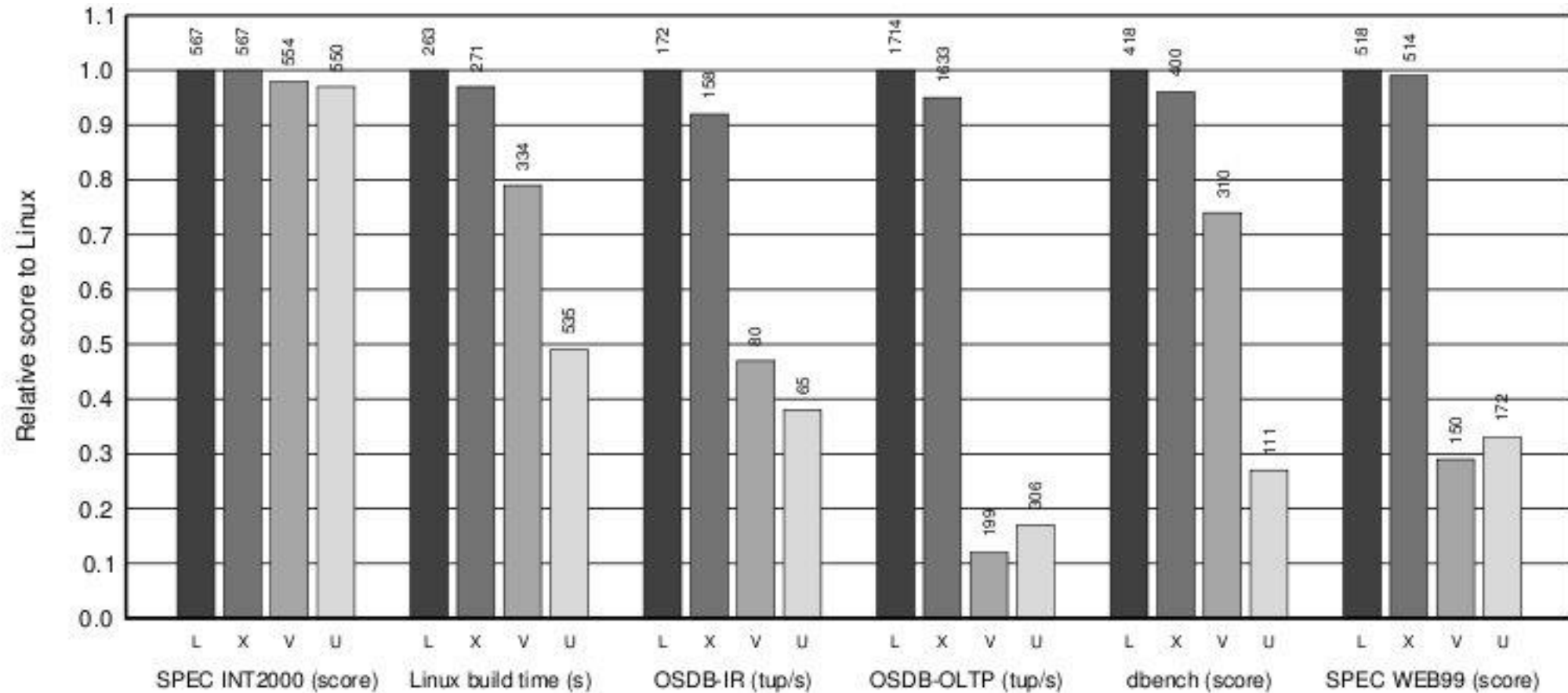


Figure 3: Relative performance of native Linux (L), XenLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

src: Xen – The Art of Virtualization

References and reading material

- [1] The Evolution of an x86 Virtual Machine Monitor (ACM SIGOPS OS review, 2010)
- [2] Xen and the Art of Virtualization (SOSP'03)
- [3] VMware white paper: Understanding Full Virtualization, Paravirtualization, and Hardware Assist (techreport 2008)
- [4] Binary Translation (Comm. of the ACM 1993)

Some material based on slide decks by Vijay Chidambaram, Dan Tsafir, Muli Ben-Yehuda, and Andrew Warfield.