Module – III
Hardware Virtualization

Full virtualization - para virtualization - server virtualization - OS level virtualization - emulation – binary translation techniques – managing storage for virtual machines.

Category of instructions

- In architecture field, the CPU designers separate instructions into different categories.
  - Privilege instruction
    - Those instructions are trapped if the machine is in user mode and are not trapped if the machine is in kernel mode.
    - ex: Instruction to modify page table base register
  - Non-Privilege instruction
    - All other instructions
    - ex: Software interrupt, Normal arithmetic operation

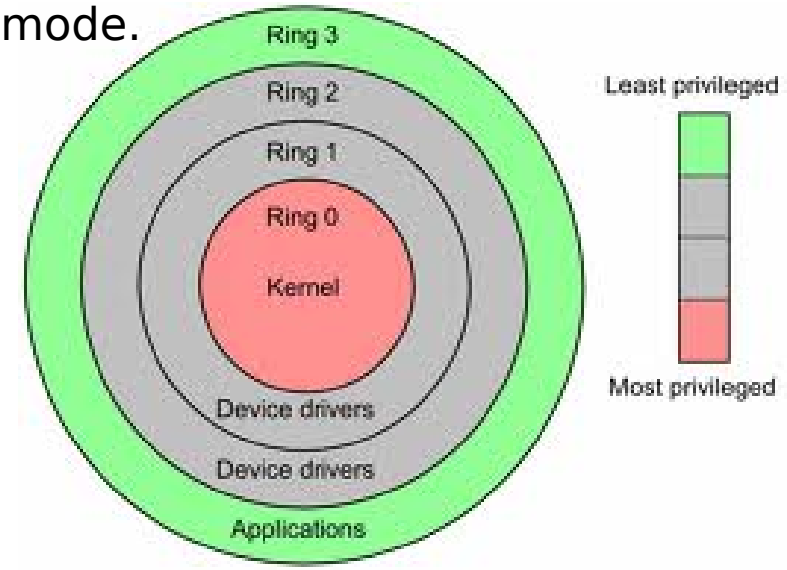
- In virtualization field, the hypervisor designers separate instructions into two categories.
  - Sensitive instruction
    - Those instructions that interact with hardware, which include control-sensitive and behavior-sensitive instructions.
    - ex: Instruction to modify page table base register, Software Interrupt, ...
  - Non-sensitive instruction
    - All other instructions
    - ex: Normal arithmetic operation, ...

Privilege instruction

- In modern computer architecture, CPU contains privilege instructions and non-privilege instructions.
- The OS designer use the privilege instructions and non-privilege instructions to separate between kernel space which access hardware resource directly and user space which access hardware resource indirectly.
- HOWEVER, which instructions are privilege is decided by CPU designers, OS designer cannot change that.
- If you execute privilege instruction in non-privilege mode, it will trigger an event and enter into the privilege mode.
  - This kind of behavior is also known as “trap”.

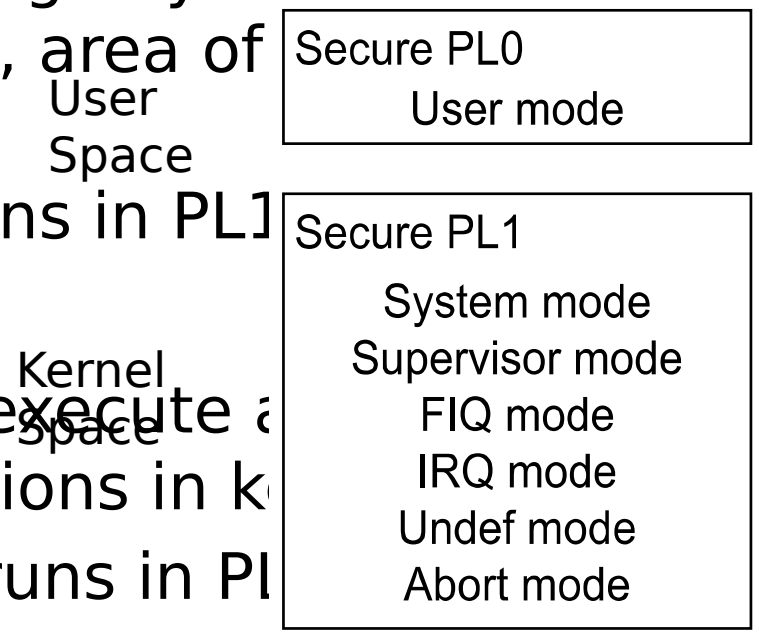
Privilege instruction

- Take x86 architecture for example:
  - Kernel mode (Ring 0)
    - CPU may perform any operation allowed by its architecture, including any instruction execution, IO operation, area of memory access, and so on.
    - Traditional OS kernel runs in Ring 0 mode.
  - User mode (Ring 1 ~ 3)
    - CPU can typically only execute a subset of those available instructions in kernel mode.
    - Traditional application runs in Ring 3 mode.



Privilege instruction

- Taking ARM architecture for another example:
  - Kernel mode (Privilege Level 1,a.k.a. PL1)
    - CPU may perform any operation allowed by its architecture, including any instruction execution, IO operation, area of access, and so on.
    - Traditional OS kernel runs in PL1
  - User mode (PL0)
    - CPU can typically only execute a those available instructions in k
    - Traditional application runs in Pl



Sensitive Instruction

- Those instructions that interact with hardware, which include control-sensitive and behavior-sensitive instructions
- Control sensitive instructions
  - Those that attempt to change the configuration of resources in the system.
- Behavior sensitive instructions
  - Those whose behavior or result depends on the configuration of resources (the content of the relocation register or the processor's mode).

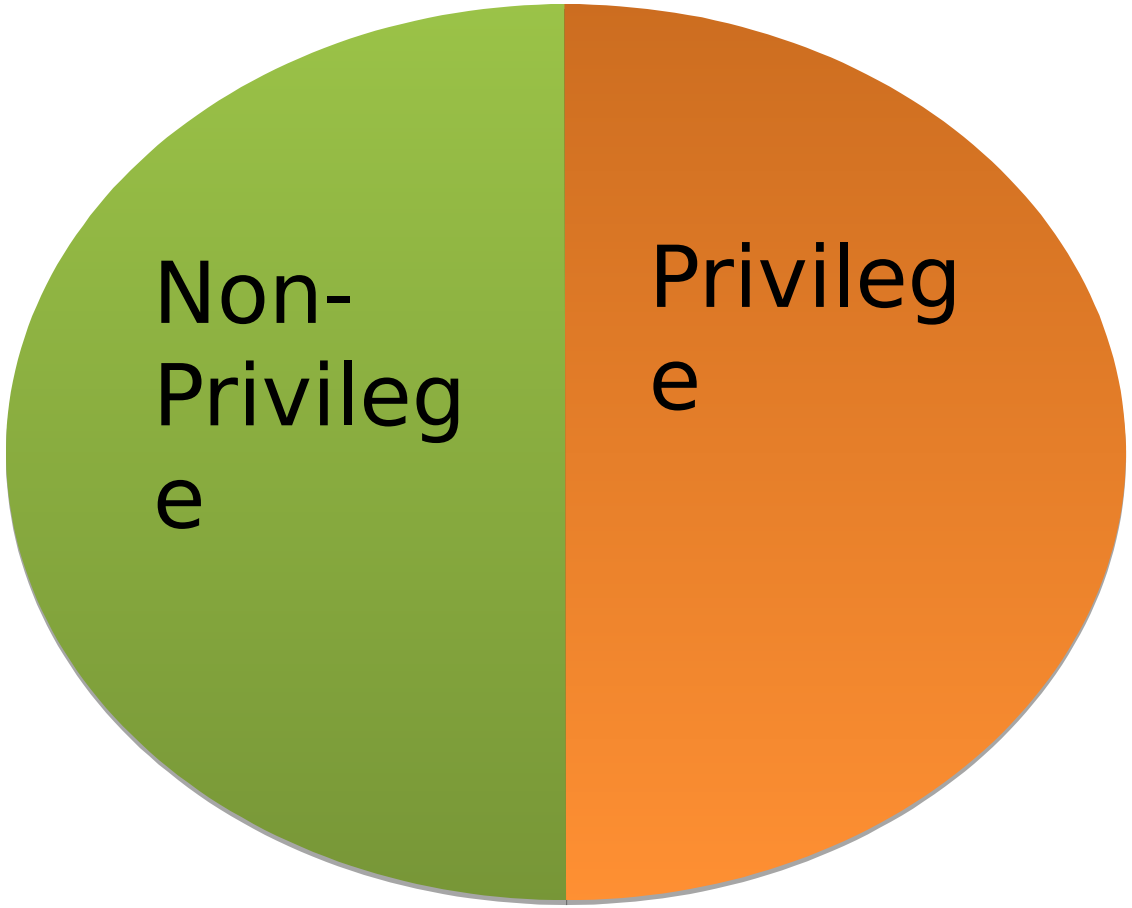
Example: ARMv6 ISA

1. Branch instructions
2. Data-processing instructions
3. Multiply instructions
4. Parallel addition and subtraction instructions
5. Extend instructions
6. Miscellaneous arithmetic instructions
7. Other miscellaneous instructions
8. Status register access instructions
9. Load and store instructions
10. Load and Store Multiple instructions
11. Semaphore instructions
12. Exception-generating instructions
13. Coprocessor instructions

Virtualizable CPU

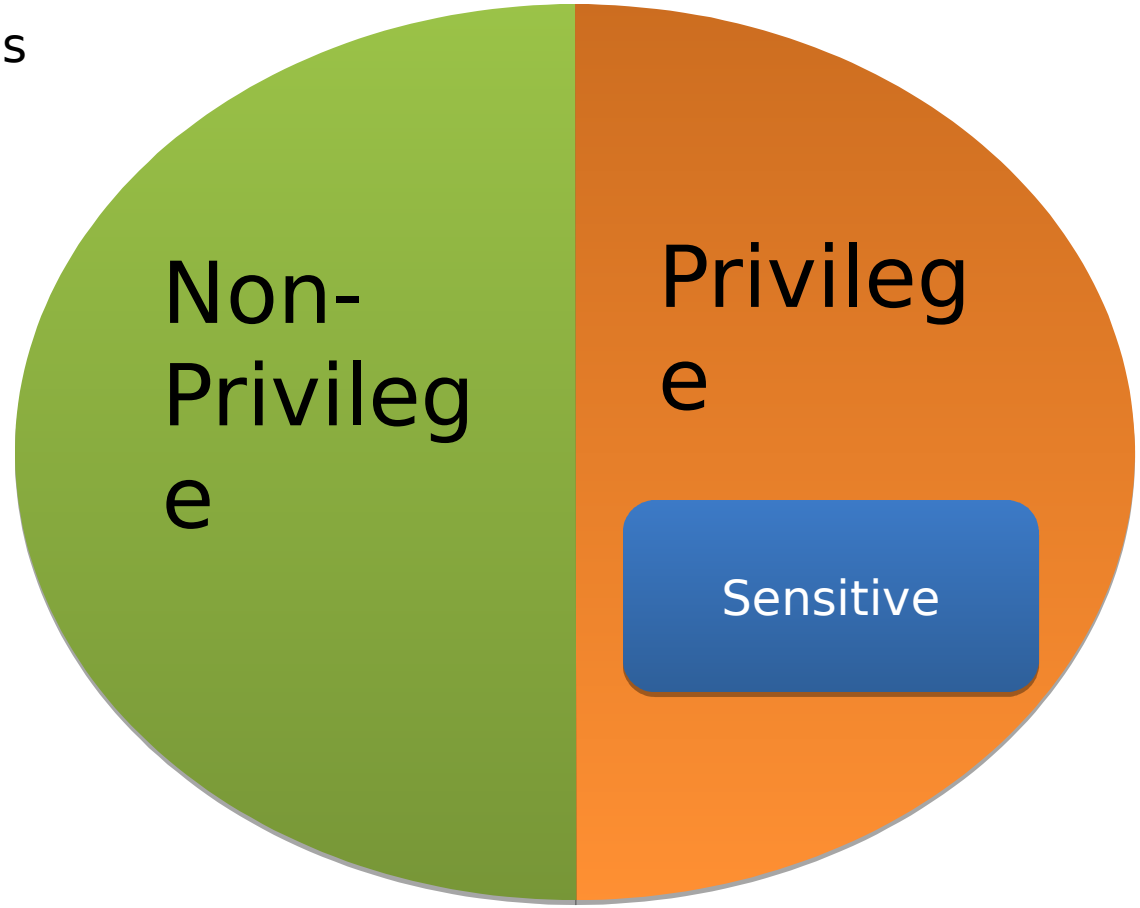
Privilege and Non-Privilege

ps: whole circle is a set of all instructions



Privilege and Sensitive

ps: whole circle is a set of all instructions



All sensitive are privilege: Virtualizable CPU

Virtualizable CPU

- All of sensitive instructions are privilege instructions.
- We call this kind of CPU as “Virtualizable CPU”
- For “Virtualizable CPU”, it is quite easy to implement hypervisor. All you have to do is to put hypervisor in privilege mode and Guest OS in non-privilege mode.
- When Guest OS wants to execute sensitive instructions, the execution will be trapped to hypervisor which is running on privilege mode automatically.
  - By this way, we can make sure that there is no chance for Guest OS to change any important hardware resource directly. All important hardware resource is under management by hypervisor.

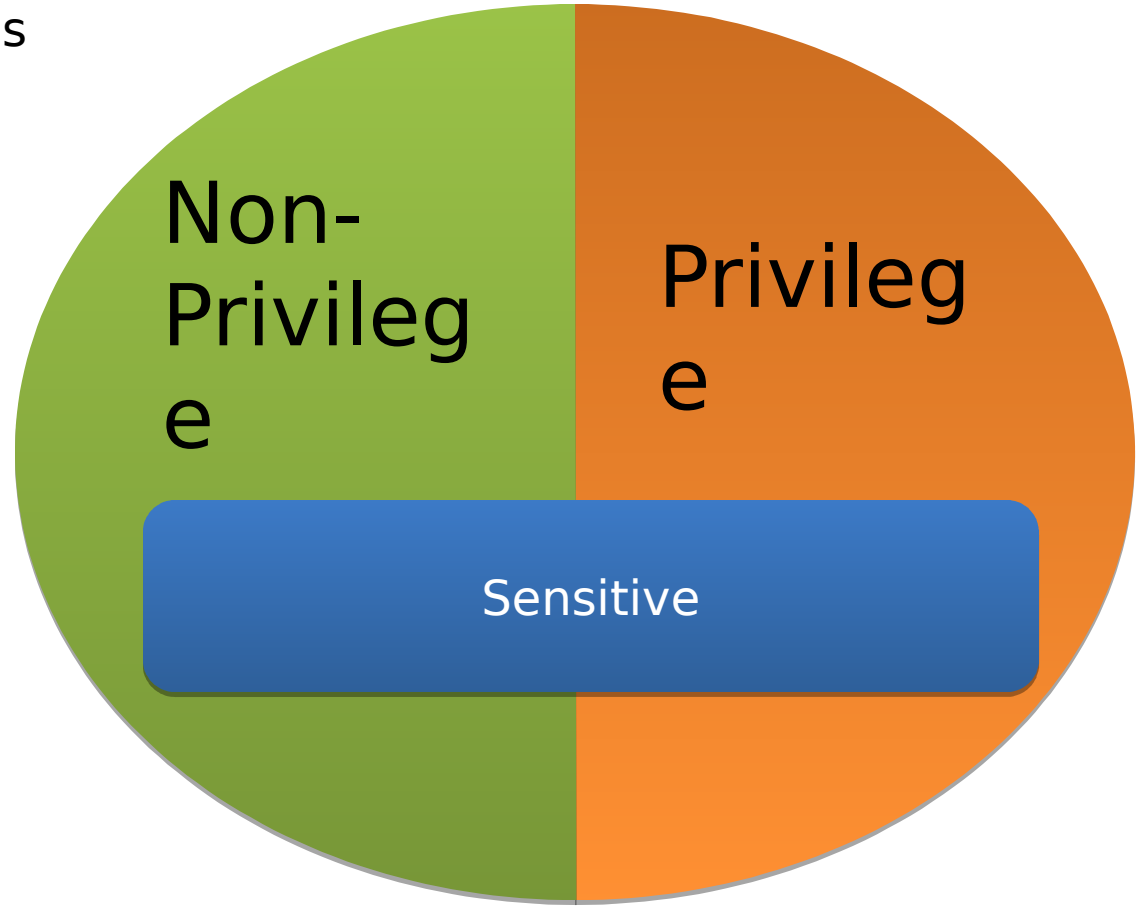
Virtualizable CPU

- Example:
  - IBM PowerPC
  - IBM S/390
    - CPU for IBM mainframe

Non-Virtualizable CPU

Privilege and Sensitive

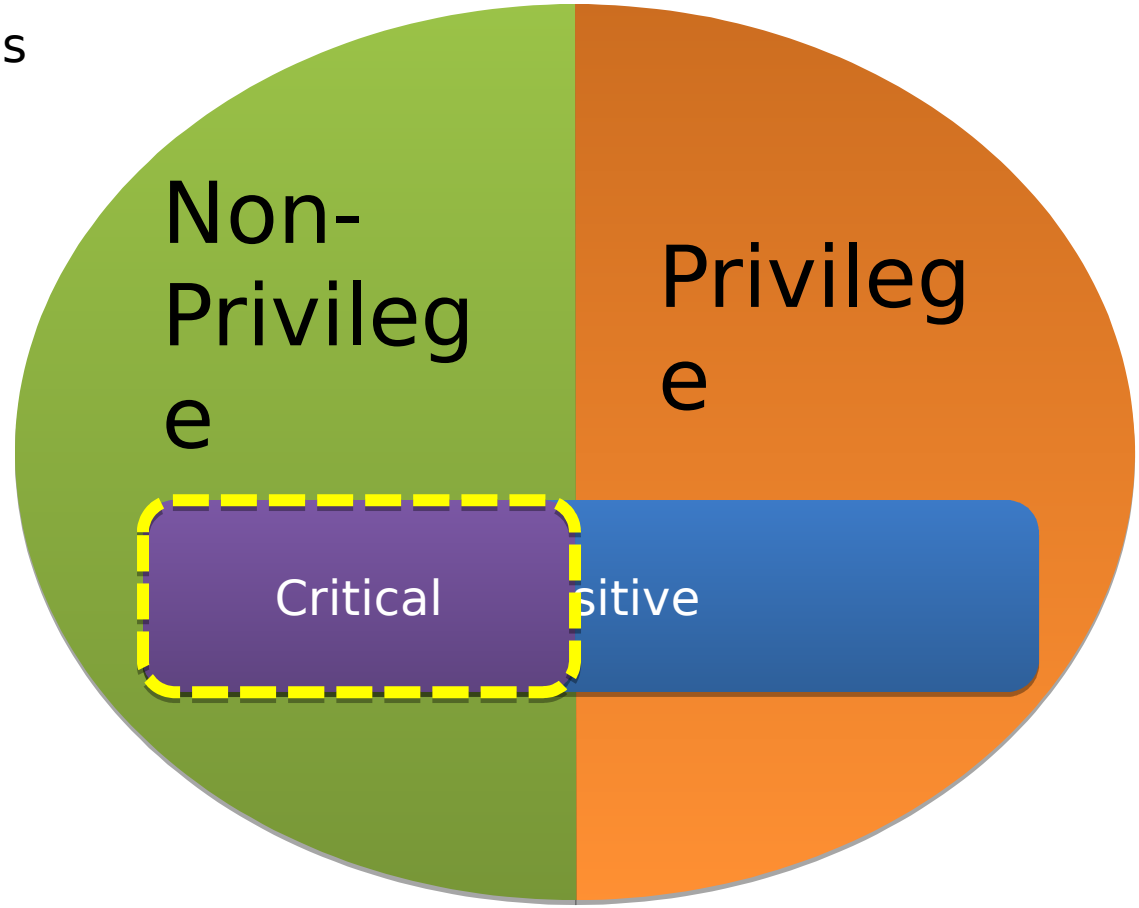
ps: whole circle is a set of all instructions



Some sensitive are non-privilege: Non-Virtualizable CPU

Privilege and Sensitive

ps: whole circle is a set of all instructions



Sensitive but Non-Privilege instruction is the problem.

We call “Sensitive but Non-Privilege instruction” as “Critical Instruction”

Non-Virtualizable CPU

- Some of sensitive instructions are privilege instructions. But some of sensitive instructions are non-privilege instructions.
- We call this kind of CPU as “Non-Virtualizable CPU”
- For “Non-Virtualizable CPU”, it is HARD to implement hypervisor.
- If you put hypervisor in privilege mode and Guest OS in non-privilege mode, when Guest OS wants to execute sensitive instructions,
  - For those privilege and sensitive instructions, they (which is) will be trapped into hypervisor which is running on privilege mode automatically.
  - For those critical instructions, they will NOT be trapped into hypervisor automatically.
- Example:
  - x86
  - ARM

Critical instruction annoy us!

- Critical instruction can be executed in privilege mode and non-privilege mode.
- The behaviors of critical instructions in privilege mode and non-privilege mode are different. It will cause problems.
- As a result, hypervisor designers have to let critical instructions be trapped to hypervisor, and let hypervisor emulate their behaviors.

Challenges in virtualizing x86 architecture

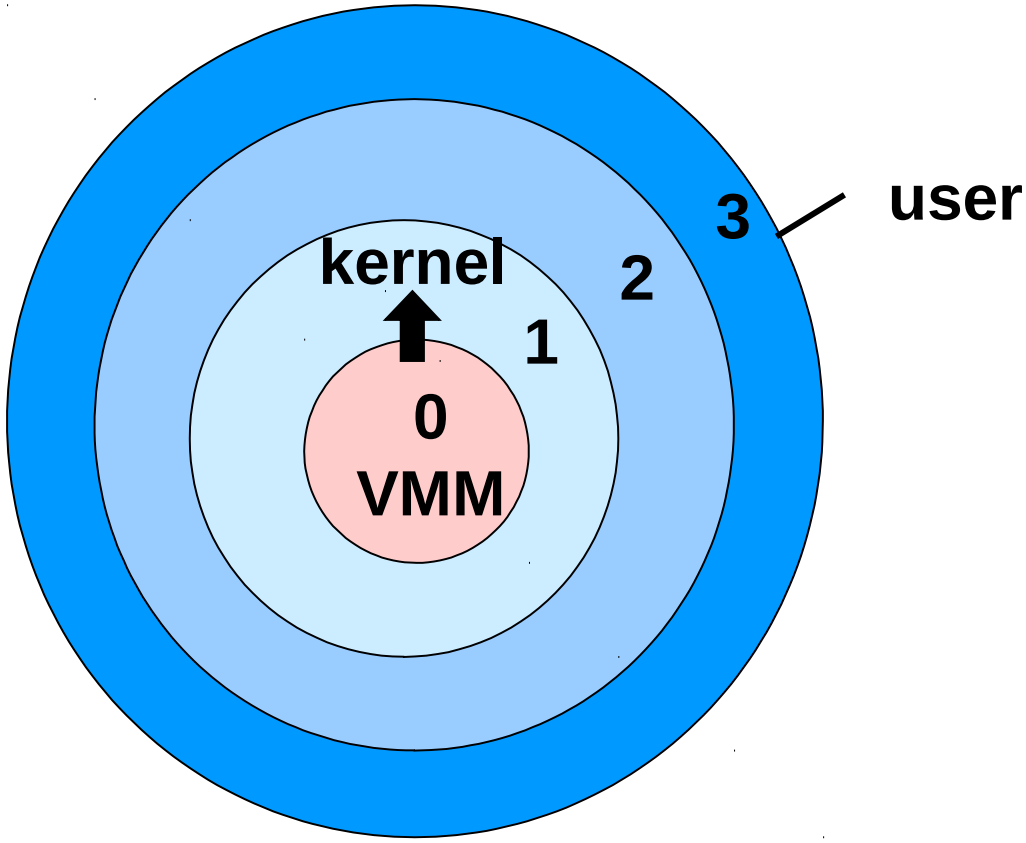
- Ring Aliasing
- Address space compression
- Non faulting access to privileged state (sensitive instructions)
- Interrupt virtualization

Virtualization through Ring Compression

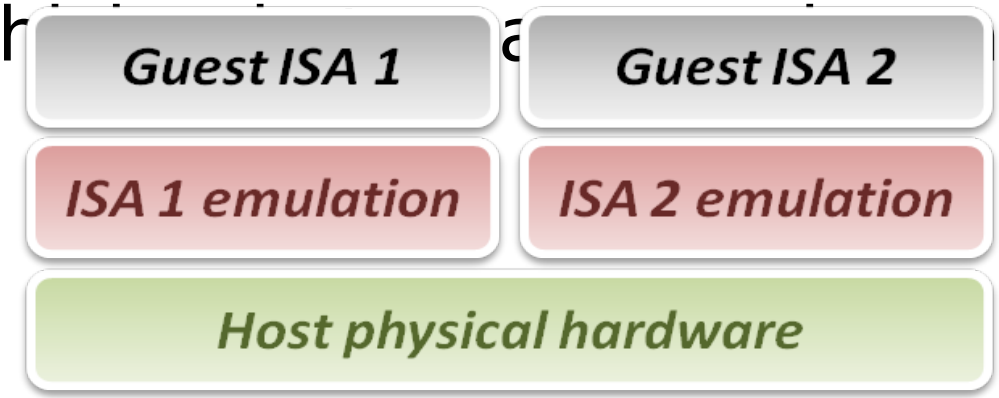
Virtual Machine Monitor (VMM) runs at ring 0

Kernel(s) run at ring 1

Requires that CPU is virtualizable

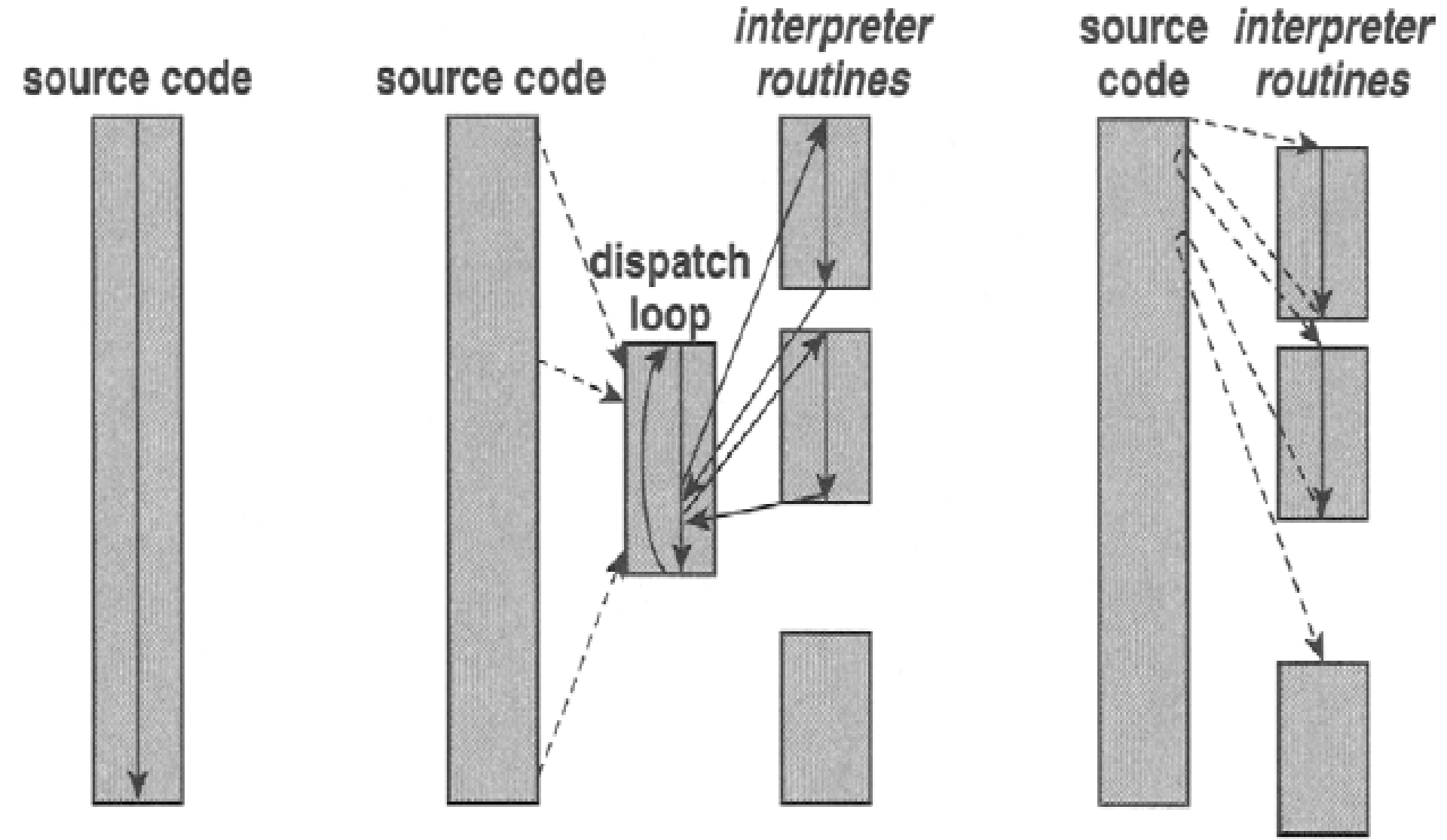


- Three emulation implementations :
  - Interpretation
    - Emulator interprets only one instruction at a time.
  - Static Binary Translation
    - Emulator translates a block of guest binary at a time and further optimizes for repeated instruction executions.
  - Dynamic Binary Translation
    - This is a hybrid approach of emulator, wh[illegible] a[illegible] bove.

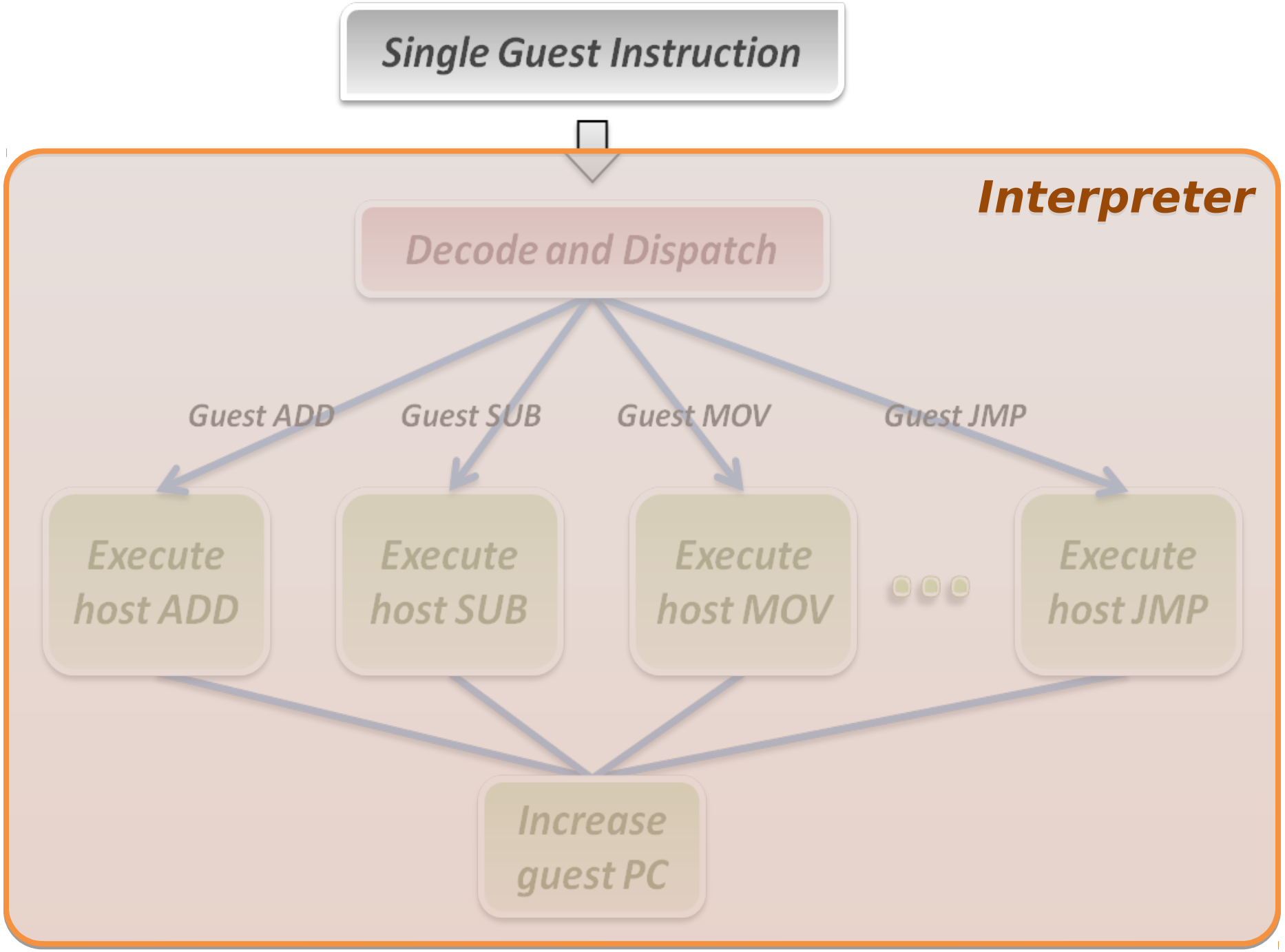


- **Approach #1: Hosted Interpretation**
  - Run the VMM as regular user application atop of host OS
    - VMM maintains a software-level representation of physical hardware
    - **Interpreter execution flow :**
      - Fetch one guest instruction from guest memory image.
      - Decode and dispatch to corresponding emulation unit.
      - Execute the functionality of that instruction and modify some related system states, such as simulated register values.
      - Increase the guest PC (Program Counter register) and then repeat this process

Interpretation

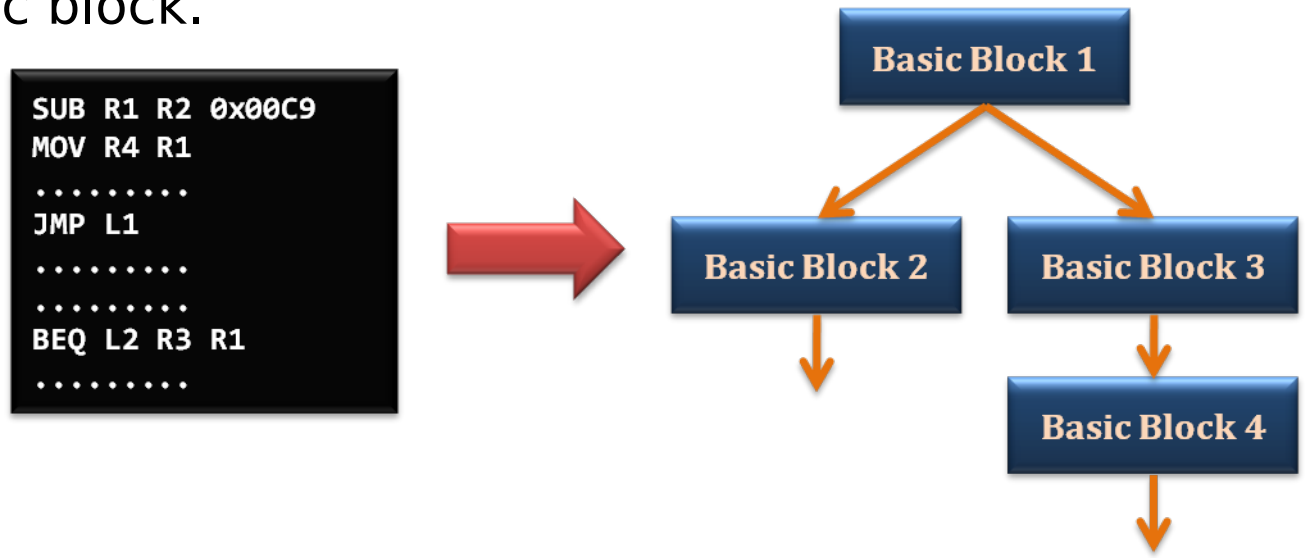


(a) Native execution (b) decode-and-dispatch interpretation (c) threaded interpretation


Single Guest Instruction
Interpreter
Decode and Dispatch
Guest ADD
Guest SUB
Guest MOV
Guest JMP
Execute host ADD
Execute host SUB
Execute host MOV
Execute host JMP
Increase guest PC

Static Binary Translation

- Using the concept of basic block which comes from compiler optimization technique.
  - A basic block is a portion of the code within a program with certain desirable properties that make it highly amenable to analysis.
  - A basic block has **only one entry point**, meaning no code within it is the destination of a jump instruction anywhere in the program.
  - A basic block has **only one exit point**, meaning only the last instruction can cause the program to begin executing code in a different basic block.



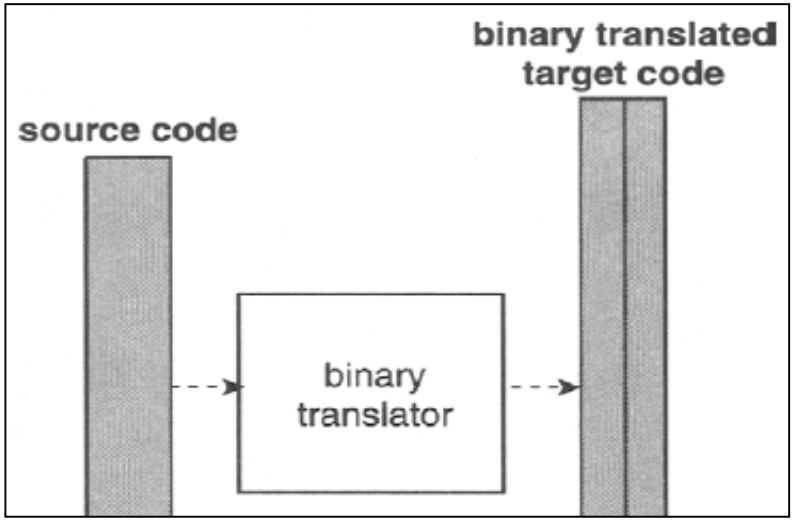
- Static binary translation flow :
  1. Fetch one block of guest instructions from guest memory image.
  2. Decode and dispatch each instruction to the corresponding translation unit.
  3. Translate guest instruction to host instructions.
  4. Write the translated host instructions to code cache.
  5. Execute the translated host instruction block in code cache.

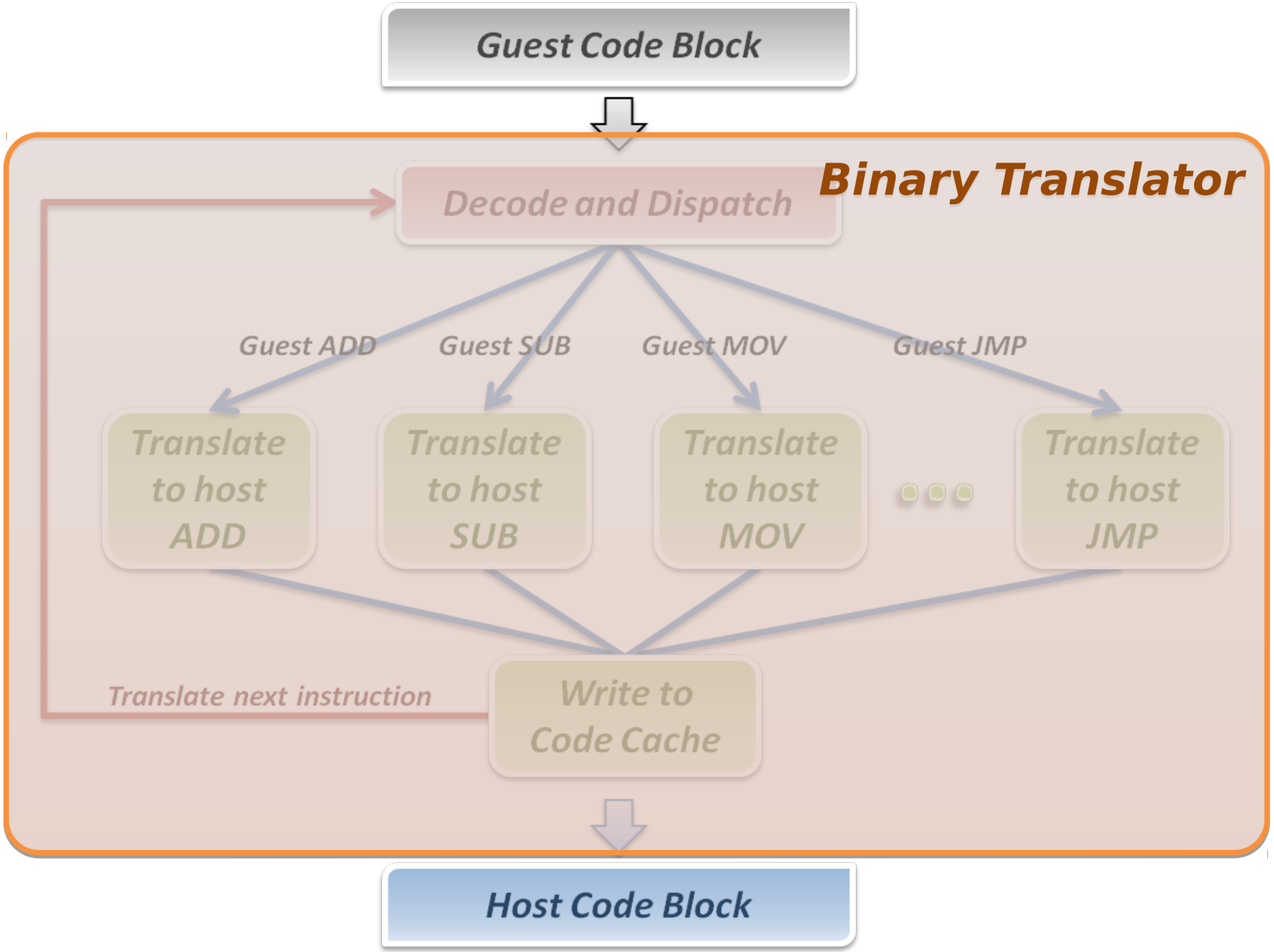
Binary Translation

```
addl    %edx,4(%eax)
movl    4(%eax),%edx
add     %eax,4
```

(a)

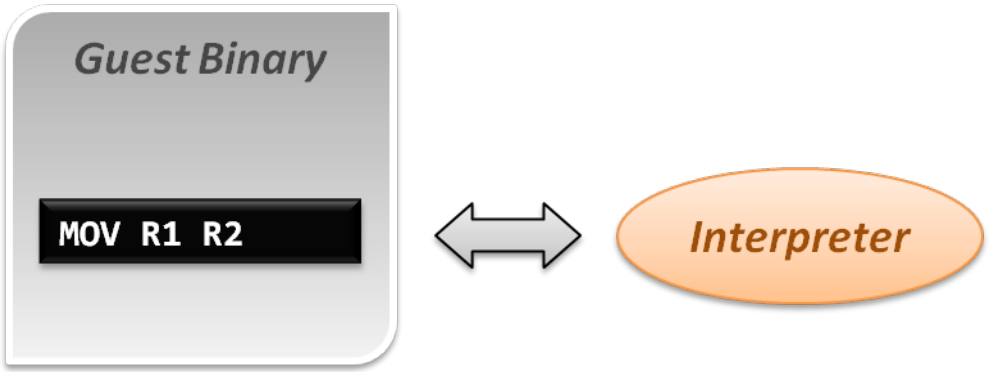
```
lwz     r4,0(r1)        ;load %eax from register block
addi    r5,r4,4         ;add 4 to %eax
lwzx    r5,r2,r5        ;load operand from memory
lwz     r4,12(r1)       ;load %edx from register block
add     r5,r4,r5        ;perform add
stw     r5,12(r1)       ;put result into %edx
addi    r3,r3,3         ;update PC (3 bytes)

lwz     r4,0(r1)        ;load %eax from register block
addi    r5,r4,4         ;add 4 to %eax
lwz     r4,12(r1)       ;load %edx from register block
stwx    r4,r2,r5        ;store %edx value into memory
addi    r3,r3,3         ;update PC (3 bytes)

lwz     r4,0(r1)        ;load %eax from register block
addi    r4,r4,4         ;add immediate
stw     r4,0(r1)        ;place result back into %eax
addi    r3,r3,3         ;update PC (3 bytes)
```



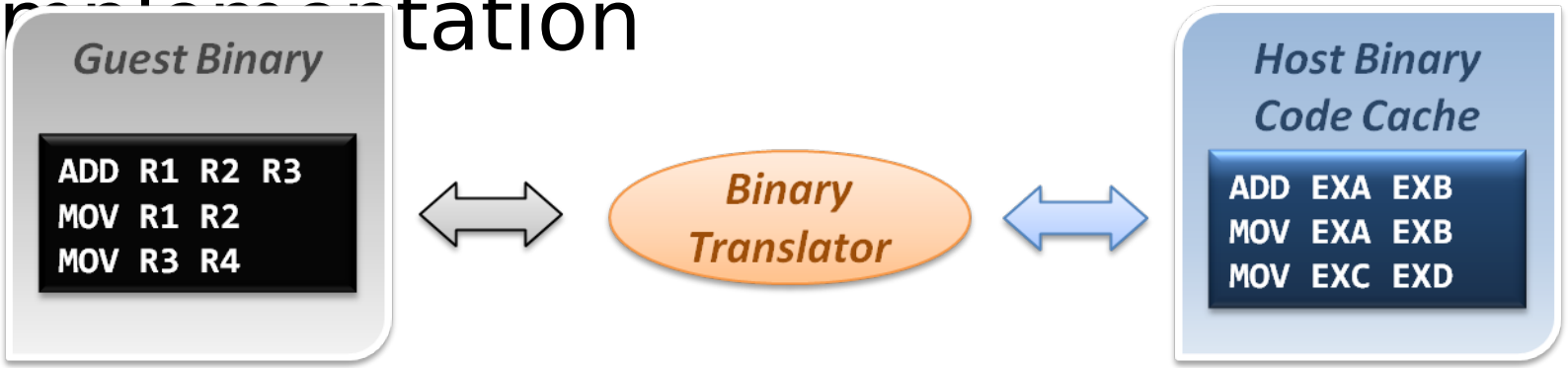

Guest Code Block
Binary Translator
Decode and Dispatch
Guest ADD
Guest SUB
Guest MOV
Guest JMP
Translate to host ADD
Translate to host SUB
Translate to host MOV
Translate to host JMP
Translate next instruction
Write to Code Cache
Host Code Block

Comparison

- Interpretation implementation

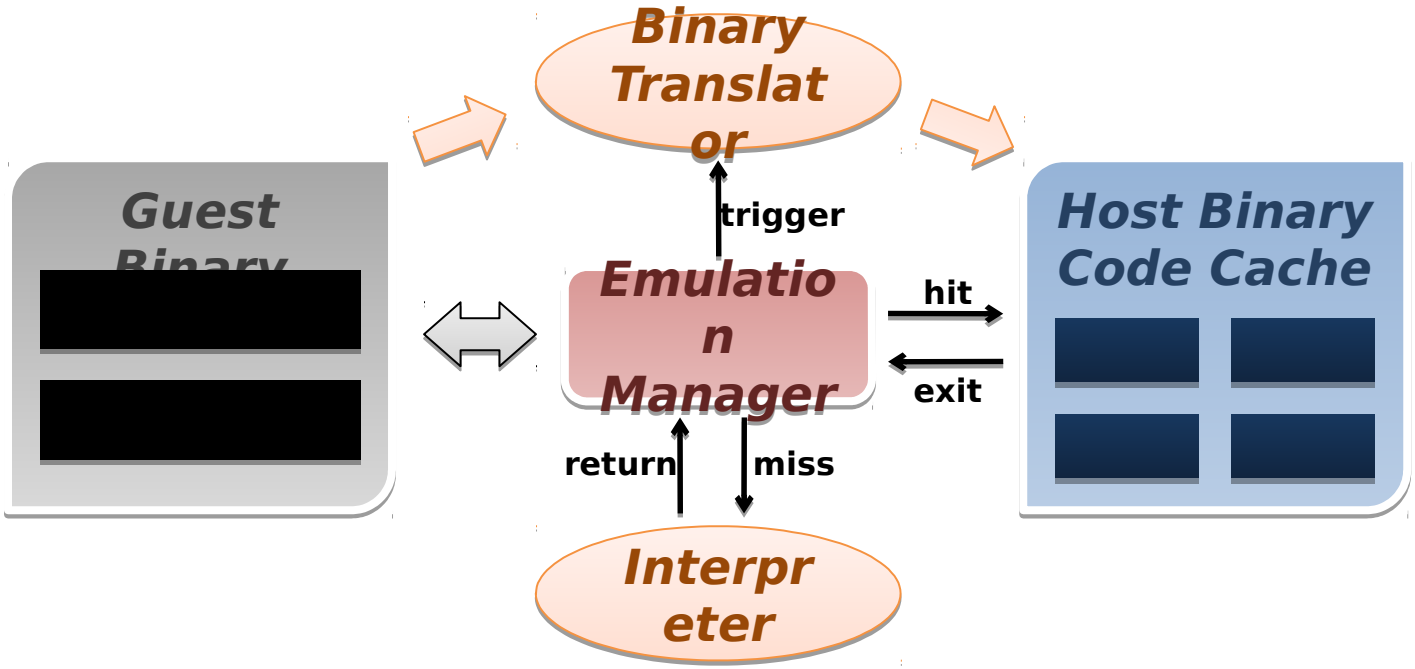


- Static binary translation implementation



Dynamic Binary Translation

1.First time execution, no translated code in code cache.
2.Miss code cache matching, then directly interpret the guest instructio
3.As a code block discovered, trigger the binary translation module.
4.Translate guest code block to host binary, and place it in the code cac
5.Next time execution, run the translated code clock in the code cache.



CPU Architecture

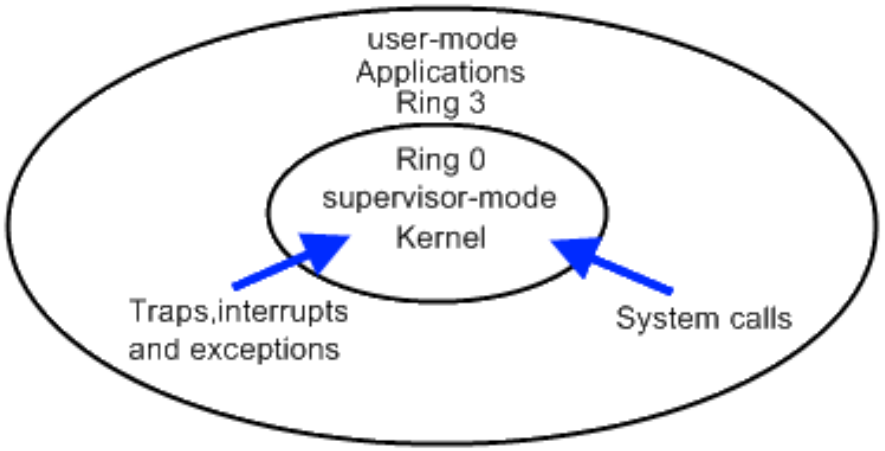
- What is trap ?
  - When CPU is running in user mode, some internal or external events, which need to be handled in kernel mode, take place.
  - Then CPU will jump to hardware exception handler vector, and execute system operations in kernel mode.
- Trap types :
  - System Call
    - Invoked by application in user mode.
    - For example, application ask OS for system IO.
  - Hardware Interrupts
    - Invoked by some hardware events in any mode.
    - For example, hardware clock timer trigger event.
  - Exception
    - Invoked when unexpected error or system malfunction occur.
    - For example, execute privilege instructions in user mode.

Approach #2: Direct Execution with Trap and Emulation

- This approach requires that a processor be "virtualizable"
  - Privileged instructions cause a trap when executed in Rings 1—3
  - Sensitive instructions access low-level machine state that should be managed by an OS or VMM
    - Ex: Instructions that modify segment/page table registers
    - Ex: IO instructions
  - Virtualizable processor: all sensitive instructions are privileged
    - If a processor is virtualizable, a VMM can interpose on any sensitive instruction that the VM tries to execute
    - VMM can control how the VM interacts with the "outside world" (i.e., physical hardware)
    - VMM can fool the guest OS into thinking that guest OS runs at the highest privilege level (e.g., if guest OS invokes sensitive instruction to check the current privilege level)

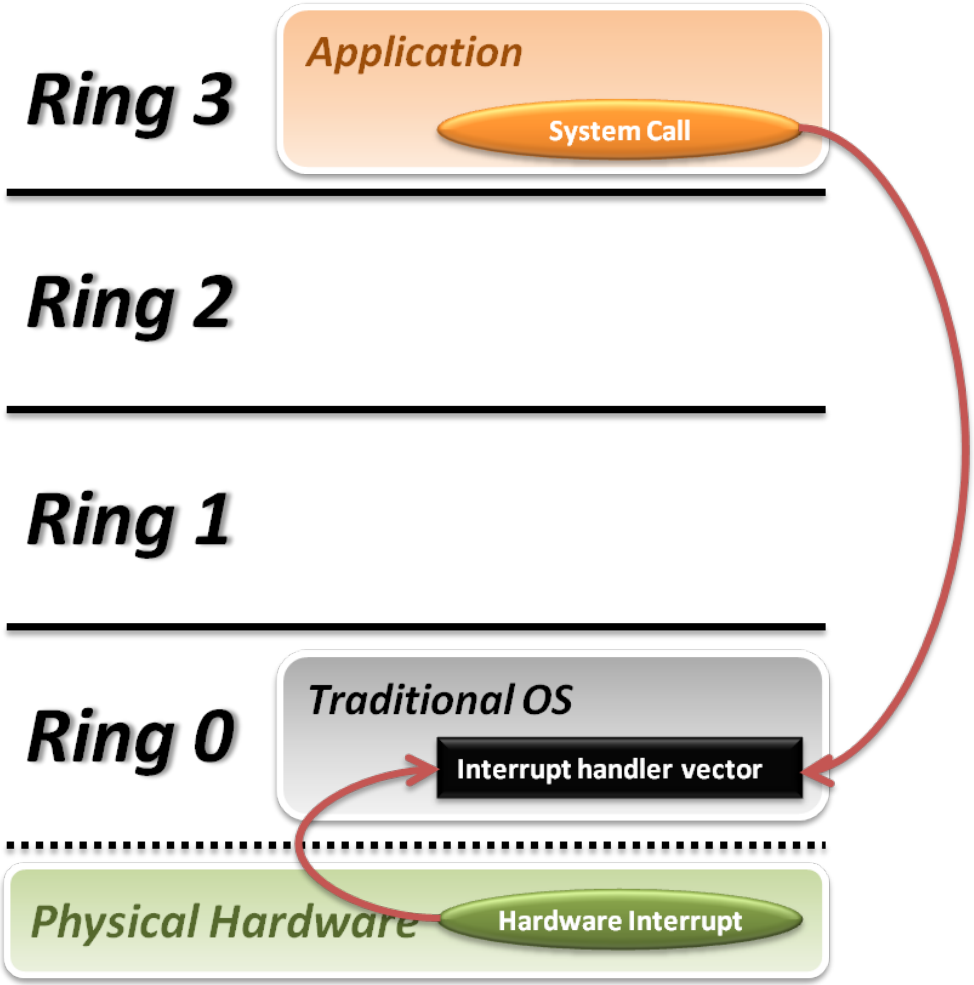
Trap and Emulate Model

- VMM virtualization paradigm *(trap and emulate)* :
  1. Let normal instructions of guest OS run directly on processor in user mode.
  2. When executing privileged instructions, hardware will make processor trap into the VMM.
  3. The VM[illegible] of the privileg[illegible] guest OS and re[illegible]



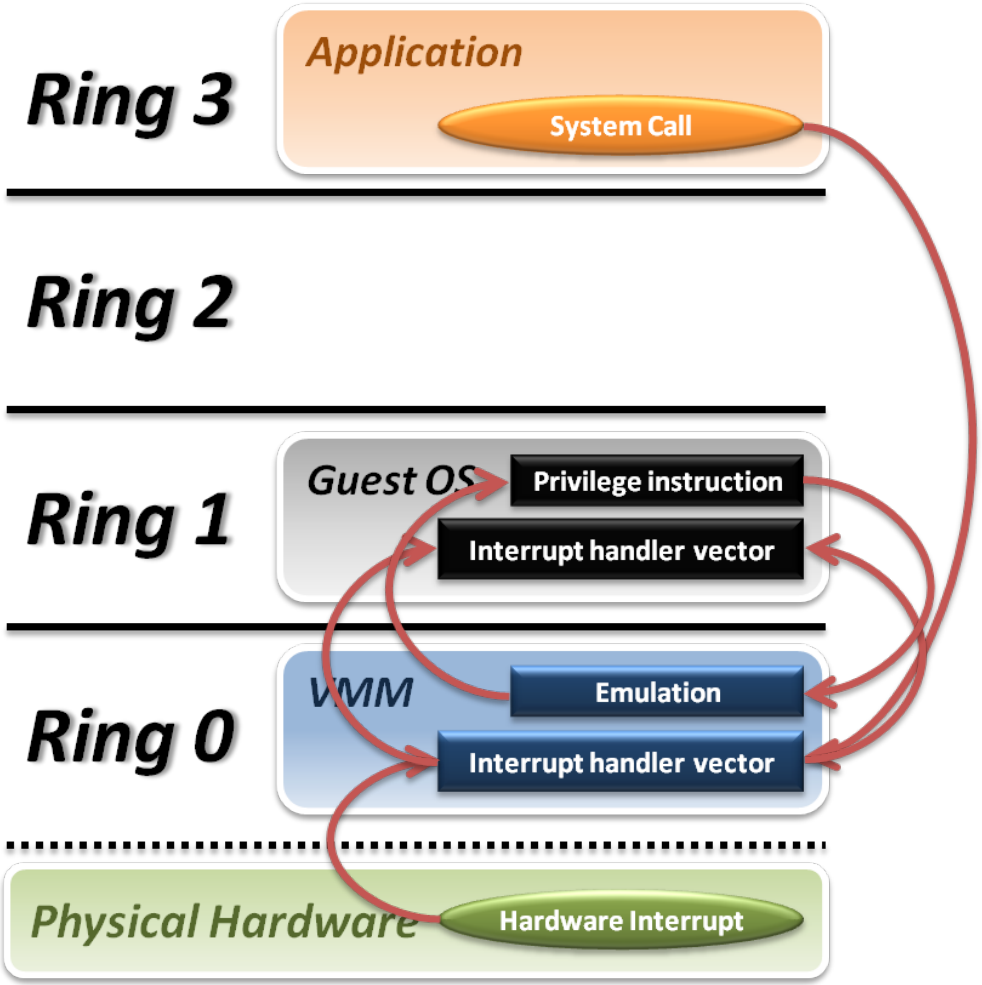
Trap and Emulate Model

- Traditional OS :
  - When application invoke a system call :
    - CPU will trap to interrupt handler vector in OS.
    - CPU will switch to kernel mode (Ring 0) and execute OS instructions.
  - When hardware event :
    - Hardware will interrupt CPU execution, and jump to interrupt handler in OS.



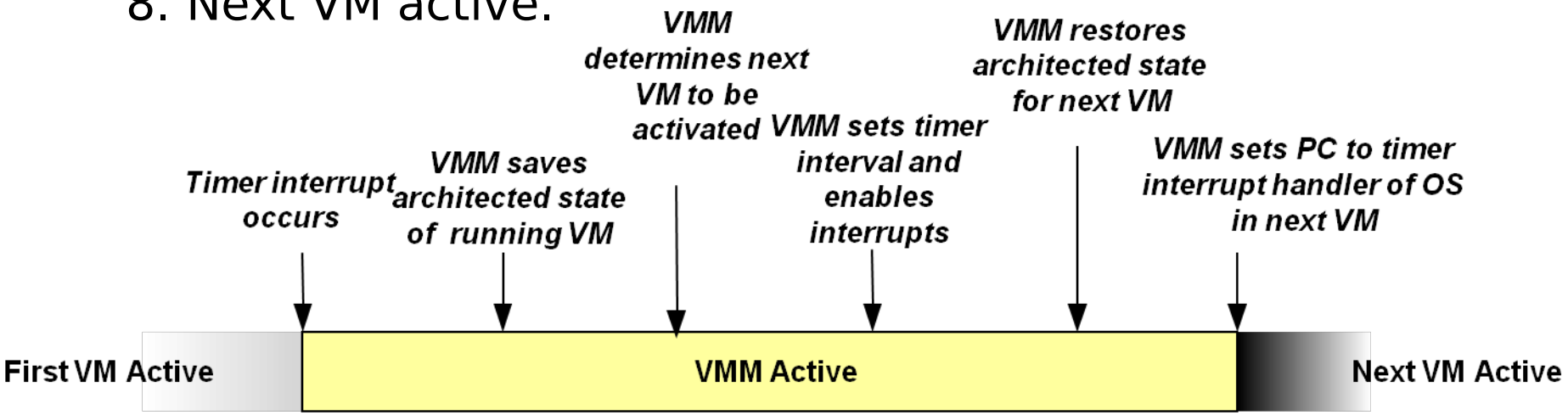
Trap and Emulate Model

- VMM and Guest OS :
  - System Call
    - CPU will trap to interrupt handler vector of VMM.
    - VMM jump back into guest OS.
  - Hardware Interrupt
    - Hardware make CPU trap to interrupt handler of VMM.
    - VMM jump to corresponding interrupt handler of guest OS.
  - Privilege Instruction
    - Running privilege instructions in guest OS will be trapped to VMM for instruction emulation.
    - After emulation, VMM jump back to guest OS.



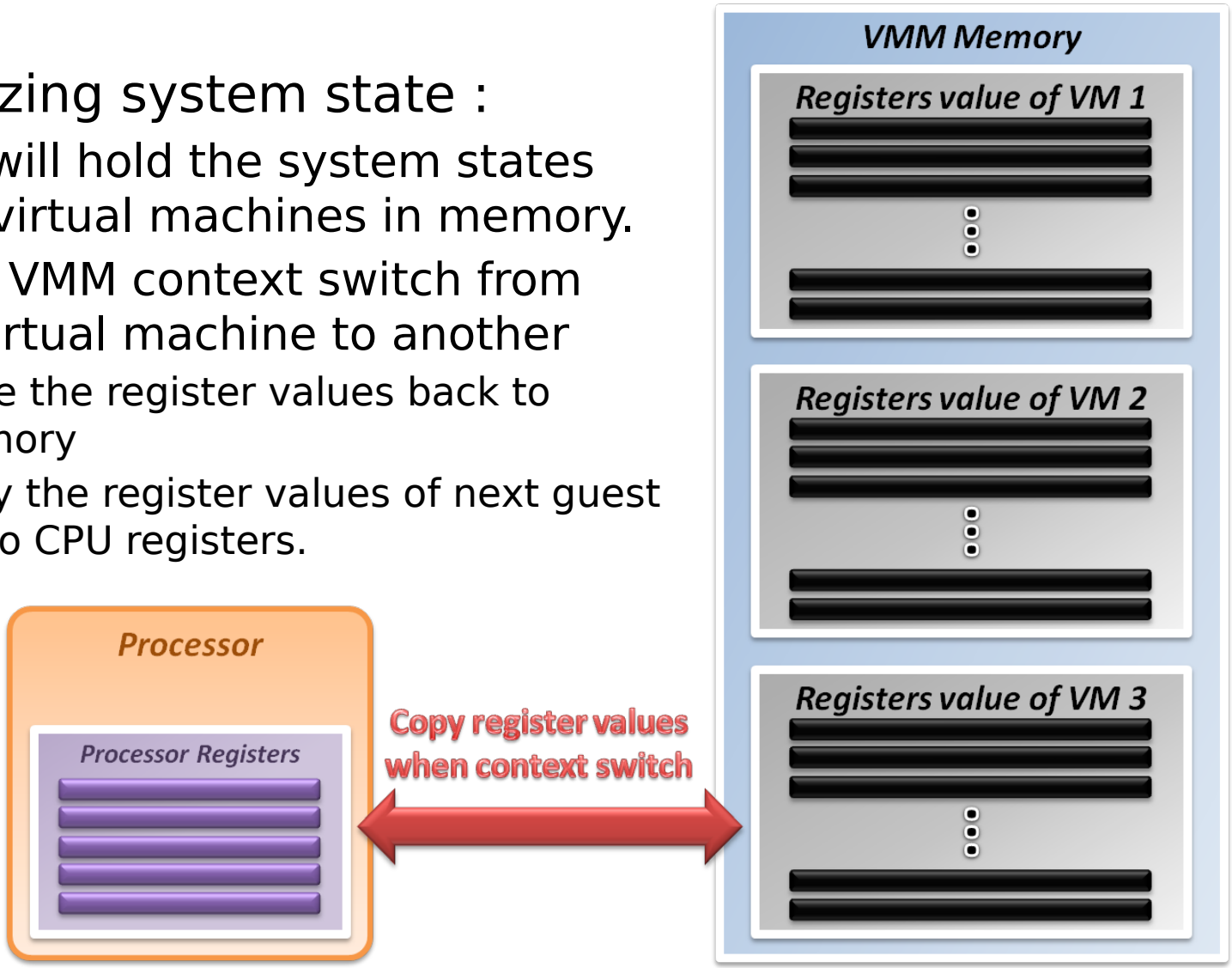
Context Switch

- Steps of VMM switch different virtual machines :
  1. Timer Interrupt in running VM.
  2. Context switch to VMM.
  3. VMM saves state of running VM.
  4. VMM determines next VM to execute.
  5. VMM sets timer interrupt.
  6. VMM restores state of next VM.
  7. VMM sets PC to timer interrupt handler of next VM.
  8. Next VM active.



System State Management

- Virtualizing system state :
  - VMM will hold the system states of all virtual machines in memory.
  - When VMM context switch from one virtual machine to another
    - Write the register values back to memory
    - Copy the register values of next guest OS to CPU registers.



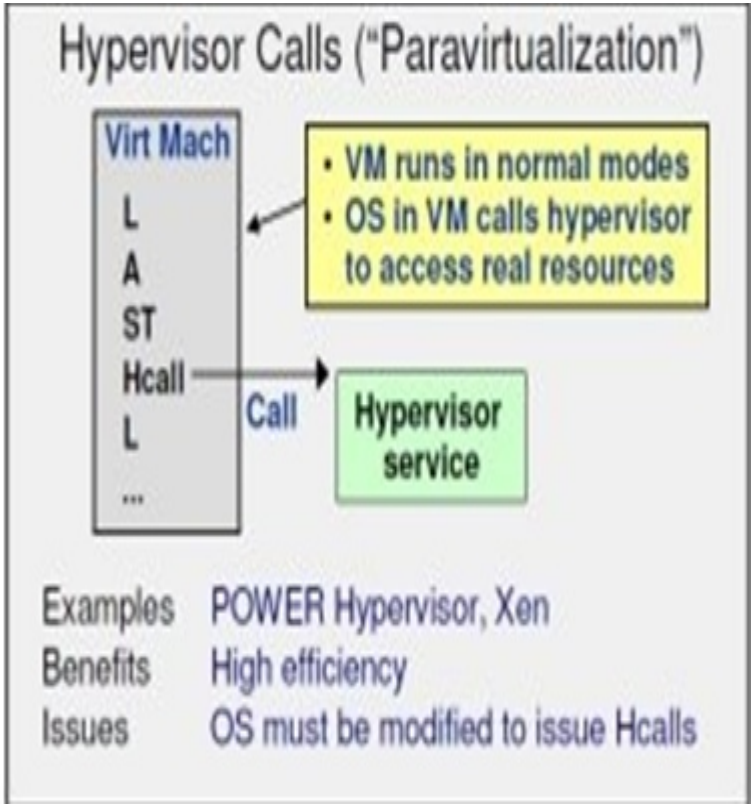
Paravirtualization!

- Does **not run unmodified** guest OSes
- Requires **guest** OS to "**know**" it is running on top of a **hypervisor**

- E.g., instead of doing **cli** to turn off interrupts, guest OS should do **hypercall(DISABLE_INTERRUPTS)**

Continued …

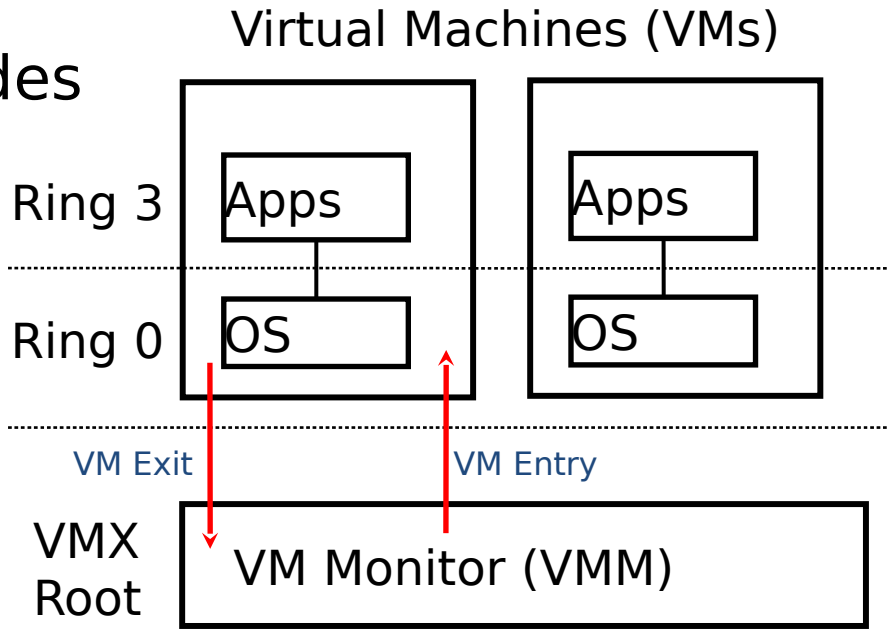
- Pros:
  - No **hardware** support required
  - **Performance** – better than emulation
- Con:
  - Requires **specifically modified guest**
  - Same guest OS cannot run in the VM and bare-metal
- Example hypervisor: Xen



CPU Hardware Virtualization Techniques

Hardware Technique - VTx

- Two new VT-x operating modes
  - Less-privileged mode (VMX non-root) for guest OSes
  - More-privileged mode (VMX root) for VMM
- Two new transitions
  - VM entry to non-root operation
  - VM exit to root operation



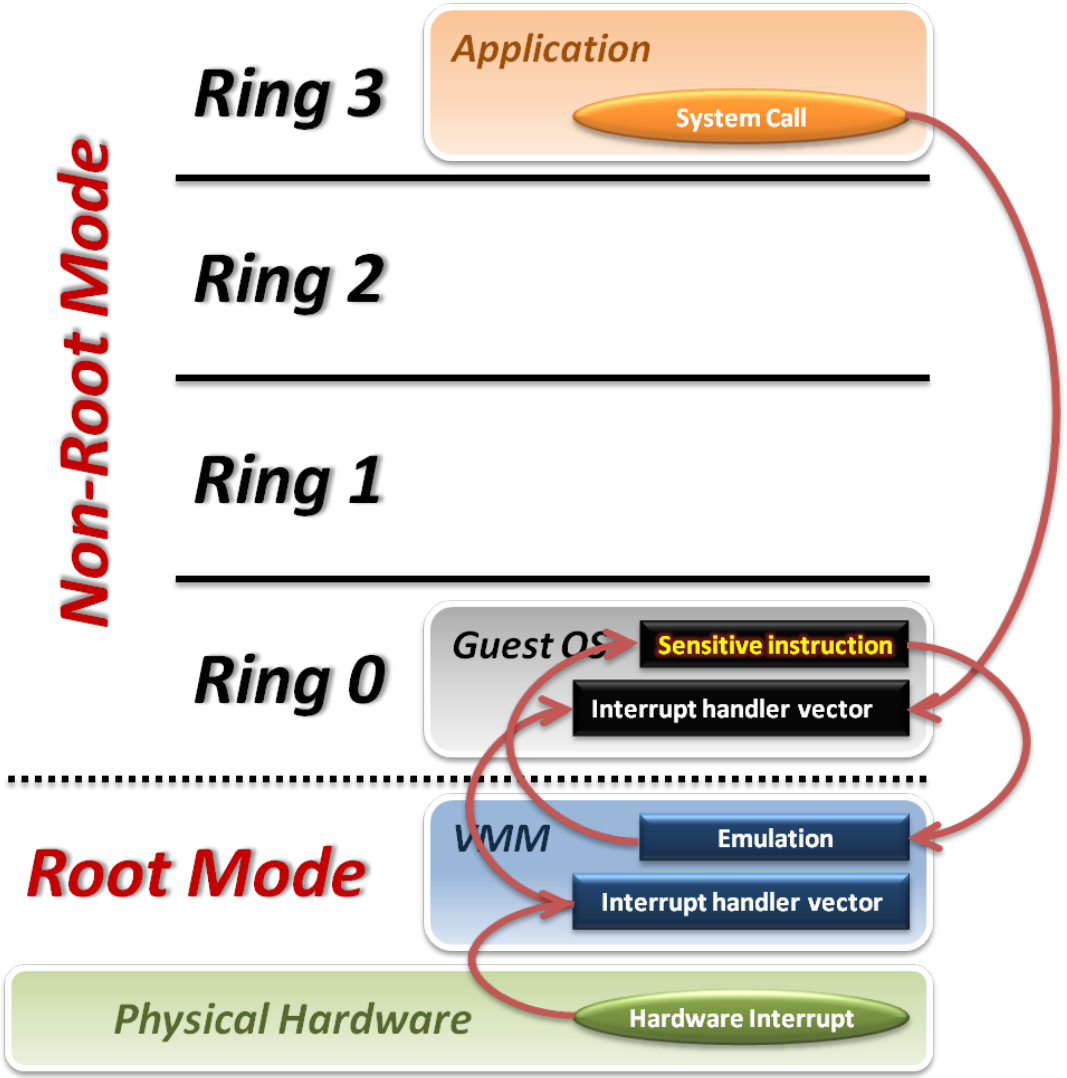
- Execution controls determine when exits occur
  - Access to privilege state, occurrence of exceptions, etc.
  - Flexibility provided to minimize unwanted exits
- VM Control Structure (VMCS) controls VT-x operation
  - Also holds guest and host state

Intel VT-x

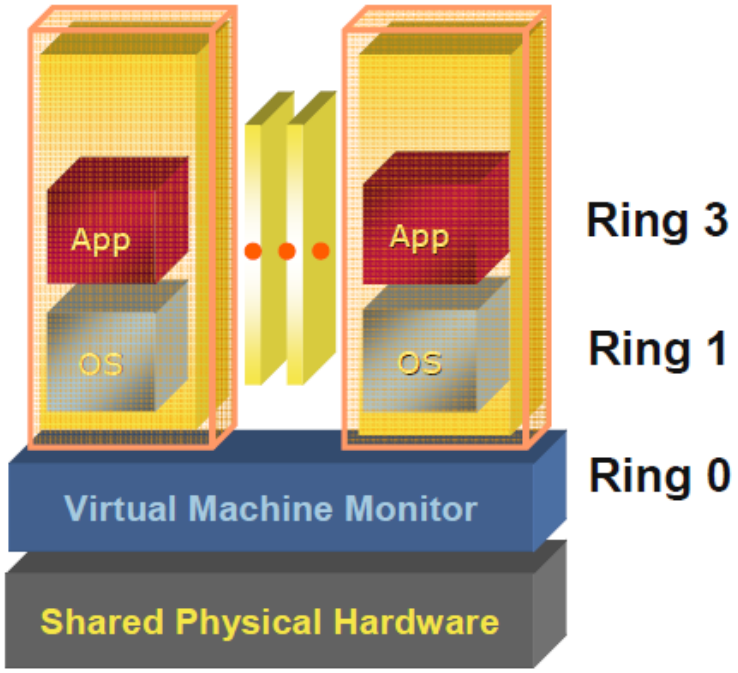
- In order to straighten those problems out, Intel introduces one more operation mode of x86 architecture.
  - VMX Root Operation (Root Mode)
    - All instruction behaviors in this mode are no different to traditional ones.
    - All legacy software can run in this mode correctly.
    - VMM should run in this mode and control all system resources.
  - VMX Non-Root Operation (Non-Root Mode)
    - All sensitive instruction behaviors in this mode are redefined.
    - The sensitive instructions will trap to Root Mode.
    - Guest OS should run in this mode and be fully virtualized through typical “*trap and emulation model*”.

Intel VT-x

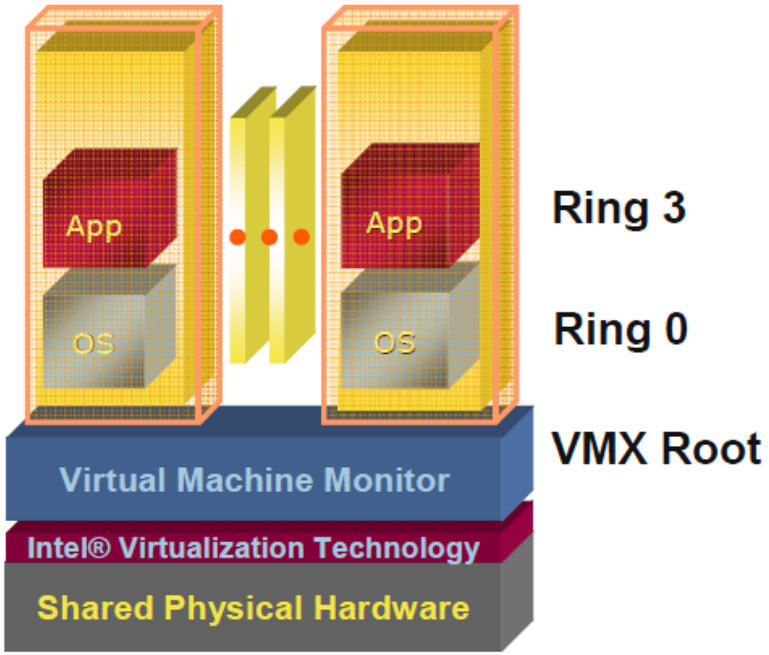
- VMM with VT-x :
  - System Call
    - CPU will directly trap to interrupt handler vector of guest OS.
  - Hardware Interrupt
    - Still, hardware events need to be handled by VMM first.
  - Sensitive Instruction
    - Instead of trap all privilege instructions, running guest OS in Non-root mode will trap sensitive instruction only.



Pre & Post Intel VT-x



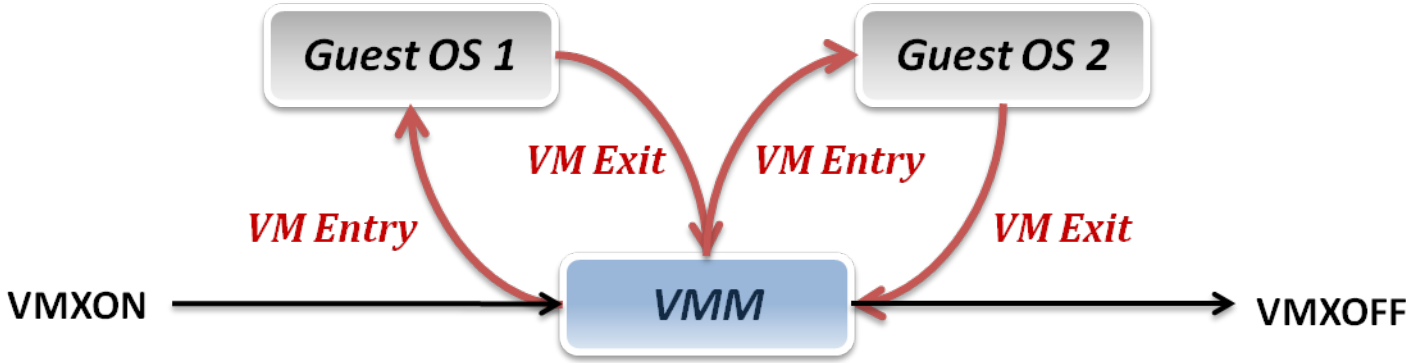
- VMM de-privileges the guest OS into Ring 1, and takes up Ring 0
- OS un-aware it is not running in traditional ring 0 privilege
- Requires compute intensive SW translation to mitigate



- VMM has its own privileged level
  where it executes
- No need to de-privilege the guest OS
- OSes run directly on the hardware

Context Switch

- VMM switch different virtual machines with Intel VT-x :
  - **VMXON/VMXOFF**
    - These two instructions are used to turn on/off CPU Root Mode.
  - VM Entry
    - This is usually caused by the execution of **VMLAUNCH/VMRESUME** instructions, which will switch CPU mode from Root Mode to Non-Root Mode.
  - VM Exit
    - This may be caused by many reasons, such as hardware interrupts or sensitive instruction executions.
    - Switch CPU mode from Non-Root Mode to Root Mode.

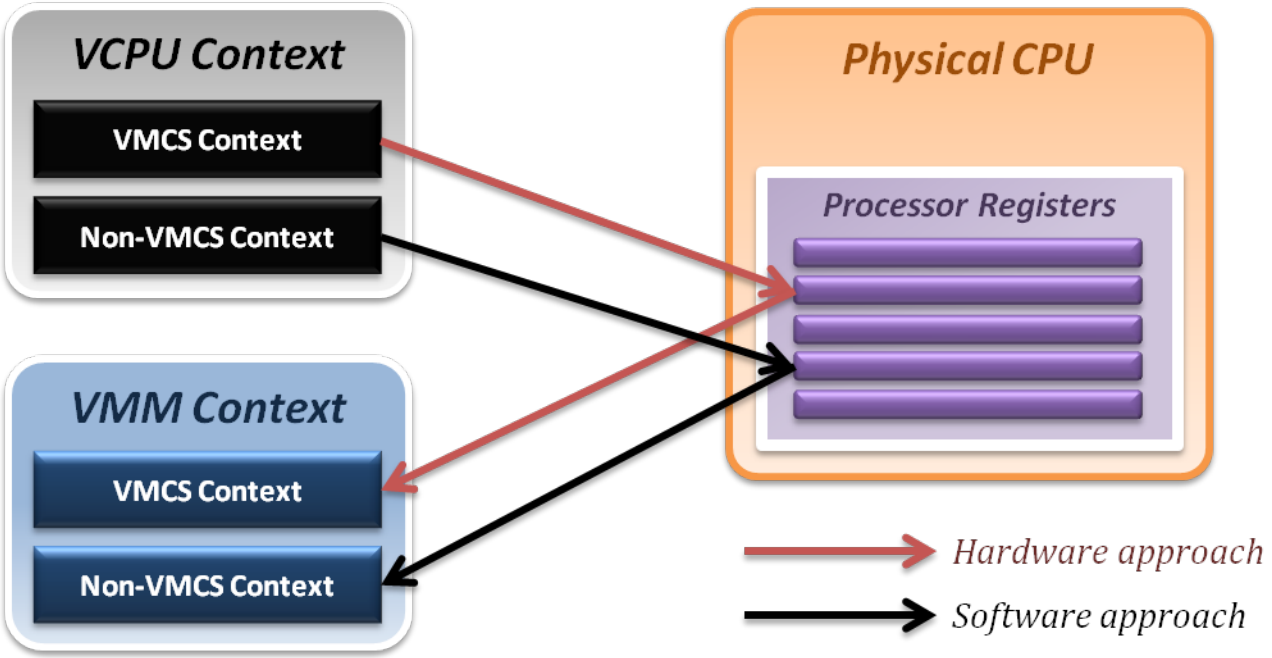


System State Management

- Intel introduces a more efficient hardware approach for register switching, **VMCS** (*Virtual Machine Control Structure*) :
  - State Area
    - Store host OS system state when VM-Entry.
    - Store guest OS system state when VM-Exit.
  - Control Area
    - Control instruction behaviors in Non-Root Mode.
    - Control VM-Entry and VM-Exit process.
  - Exit Information
    - Provide the VM-Exit reason and some hardware information.
- Whenever VM Entry or VM Exit occur, CPU will automatically read or write corresponding information into VMCS.

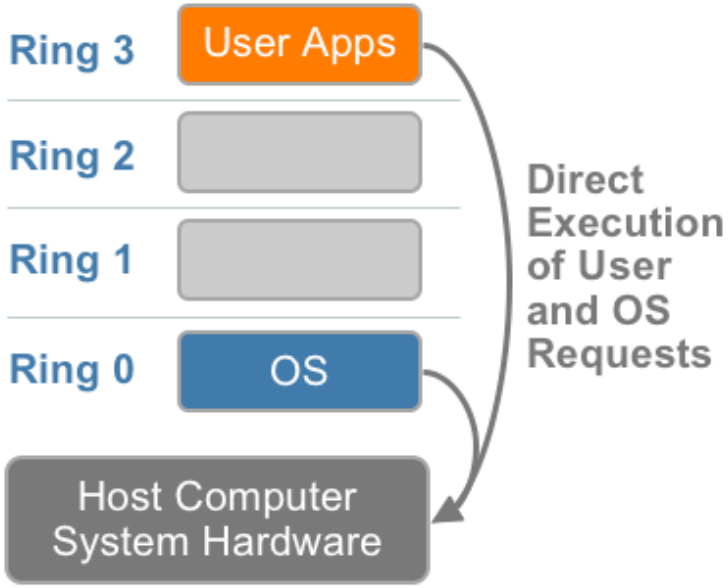
System State Management

- Binding virtual machine to virtual CPU
  - VCPU (Virtual CPU) contains two parts
    - VMCS maintains virtual system states, which is approached by hardware.
    - Non-VMCS maintains other non-essential system information, which is approach by software.
  - VMM needs to handle Non-VMCS part.



x86 Hardware Virtualization

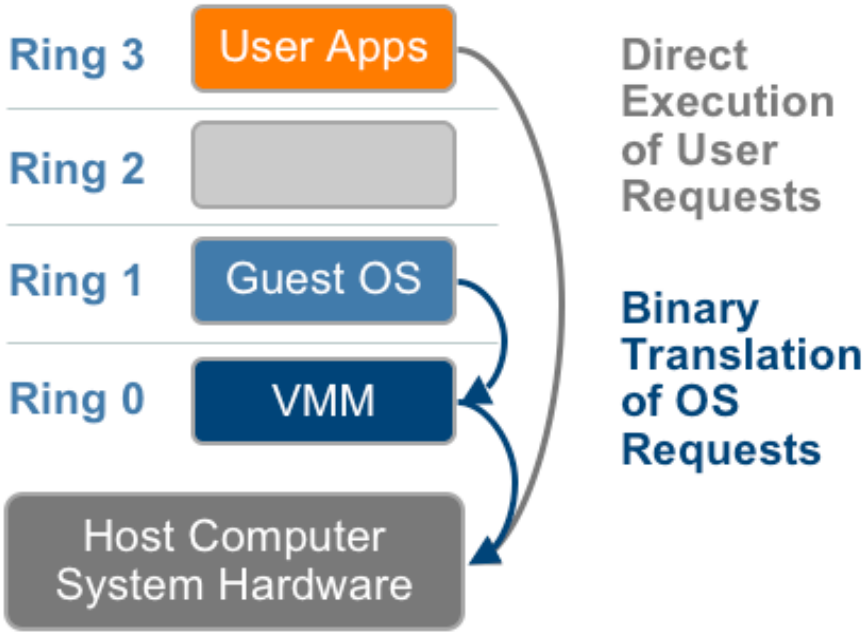
- The x86 architecture offers four levels of privilege known as Ring 0, 1, 2 and 3 to operating systems and applications to manage access to the computer hardware. While user level applications typically run in Ring 3, the operating system needs to have direct access to the memory and hardware and must execute its privileged instructions in Ring 0



x86 privilege level architecture without virtualization

Technique 1: Full Virtualization using Binary Translation

- This approach relies on binary translation to trap (into the VMM) and to virtualize certain sensitive and non-virtualizable 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.



Binary translation approach to x86 virtualization

Full Virtualization using Binary Translation

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

- The guest OS is not aware it is being virtualized and requires no modification.

- **The hypervisor translates all operating system instructions** at run-time on the fly and caches the results for future use, while user level instructions run unmodified at native speed.

- VMware's virtualization products such as VMWare ESXi and Microsoft Virtual Server are examples of full virtualization.

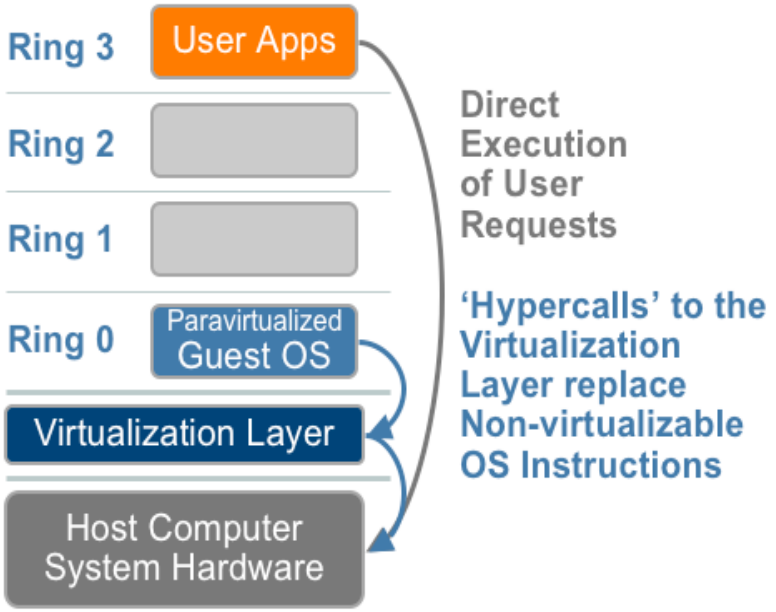
Full Virtualization using Binary Translation

- The performance of full virtualization may not be ideal because it involves binary translation at run-time which is time consuming and can incur a large performance overhead.
- The full virtualization of I/O – intensive applications can be a challenge.
- Binary translation employs a code cache to store translated hot instructions to improve performance, but it increases the cost of memory usage.
- The performance of full virtualization on the x86 architecture is typically 80% to 97% that of the host machine.

Technique 2: OS Assisted Virtualization or Paravirtualization (PV)

- Paravirtualization refers to communication between the guest OS and the hypervisor to improve performance and efficiency.
- Paravirtualization 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 approach to x86 Virtualizati



Technique 3: Hardware Assisted Virtualization (HVM)

- Intel's Virtualization Technology (VT-x) (e.g. Intel Xeon) and AMD's AMD-V 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, also referred to as Ring 0P (for privileged root mode) while the Guest OS runs in Ring 0D (for de-privileged non-root mode).

- Privileged and sensitive calls are set to automatically trap to the hypervisor and handled by hardware, removing the need for either binary translation or para-virtualization.

- Vmware only takes advantage of these first generation hardware features in limited cases such as for 64-bit guest support on Intel processors.

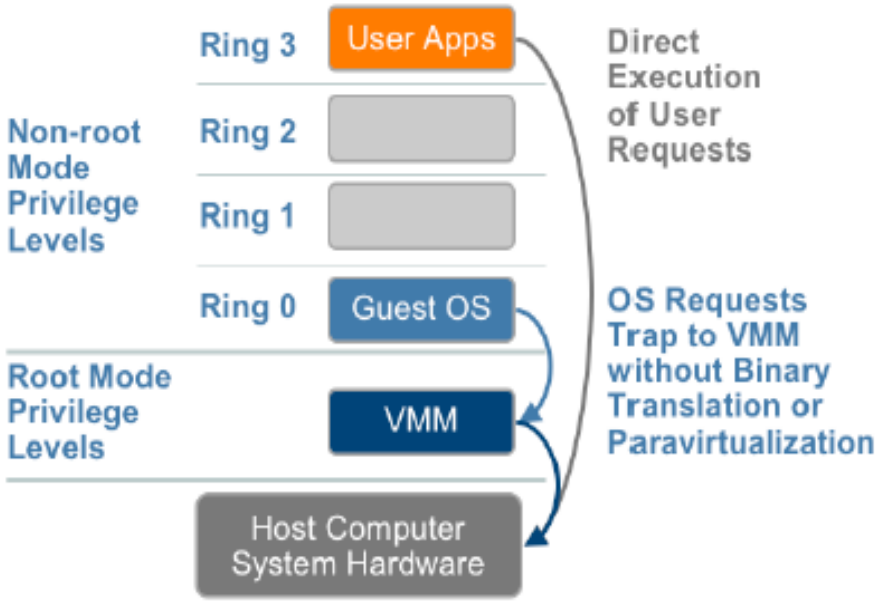
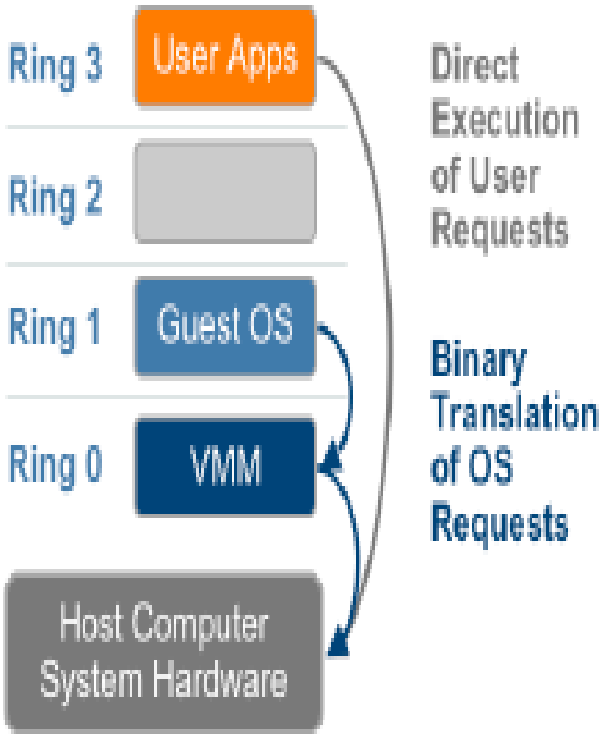
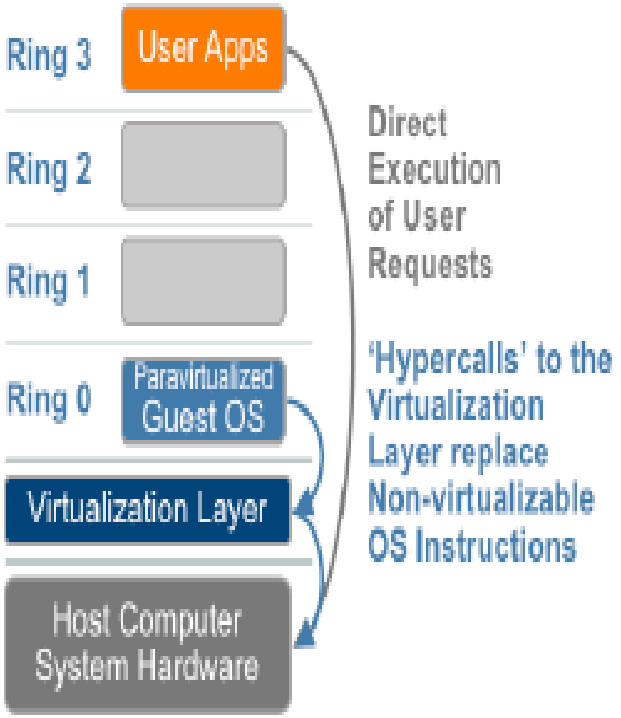
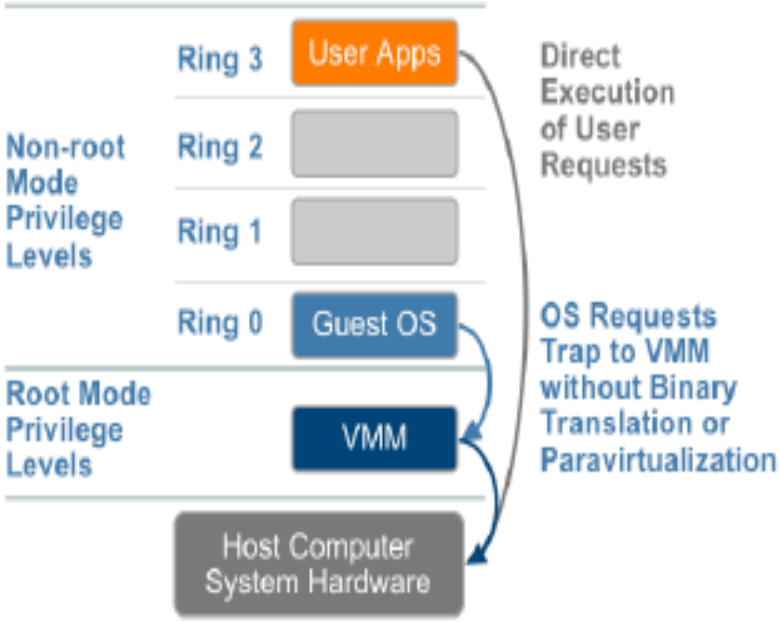
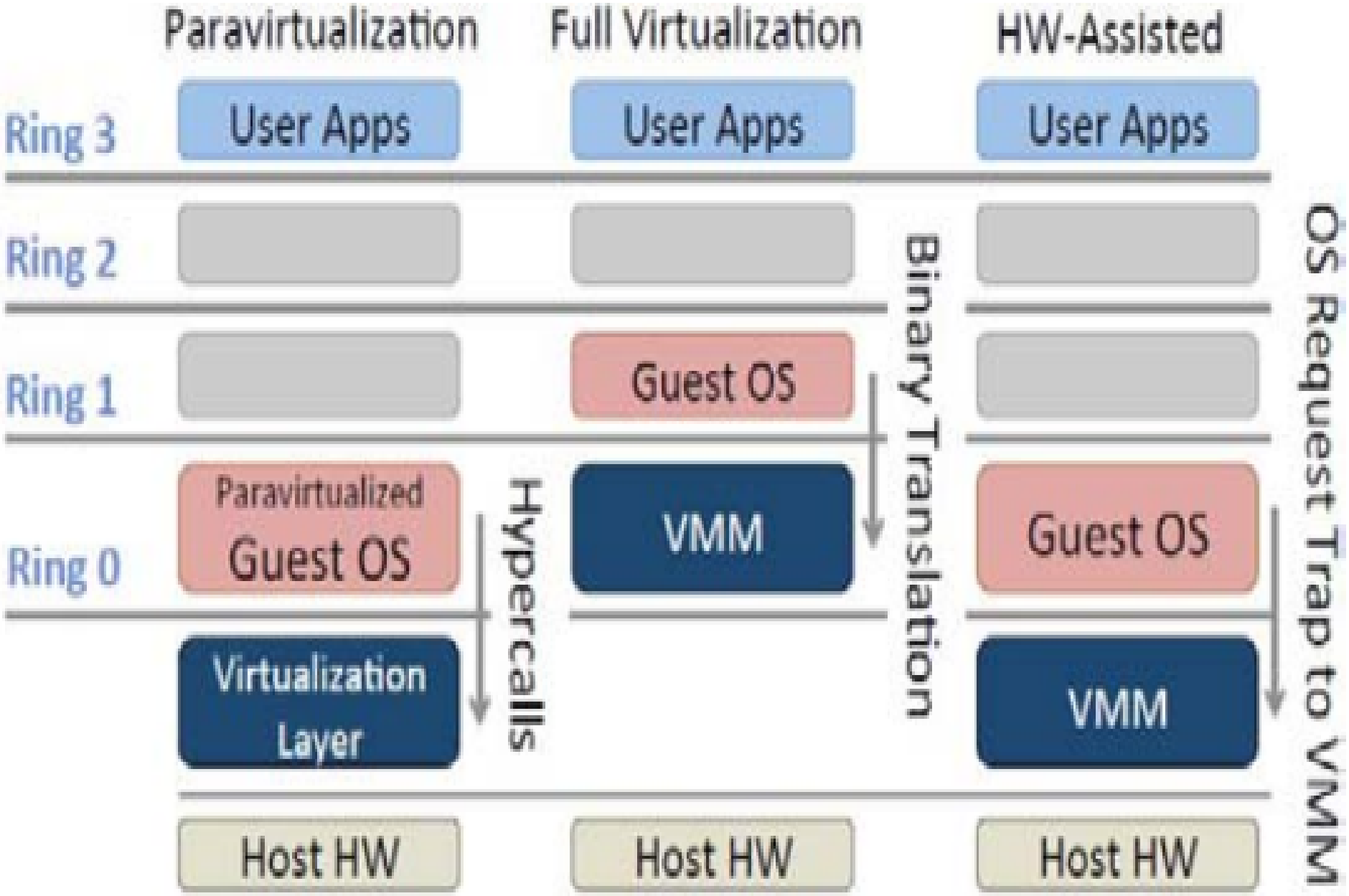


Figure 7 – The hardware assist approach to x86 virtualization


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


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


Non-root Mode Privilege Levels
Ring 3
User Apps
Ring 2
Ring 1
Ring 0
Guest OS
Root Mode Privilege Levels
VMM
Host Computer System Hardware
Direct Execution of User Requests
OS Requests Trap to VMM without Binary Translation or Paravirtualization


Paravirtualization
Full Virtualization
HW-Assisted
Ring 3
Ring 2
Ring 1
Ring 0
User Apps
User Apps
User Apps
Guest OS
Paravirtualized
Guest OS
VMM
Guest OS
Virtualization
Layer
VMM
Hypercalls
Binary Translation
OS Request Trap to VMM
Host HW
Host HW
Host HW

Summary Comparison of the Current State of x86 Virtualization Techniques

	Full Virtualization with Binary Translation	Hardware Assisted Virtualization	OS Assisted Virtualization / Paravirtualization
Technique	Binary Translation and Direct Execution	Exit to Root Mode on Privileged Instructions	Hypercalls
Guest Modification / Compatibility	Unmodified Guest OS Excellent compatibility	Unmodified Guest OS Excellent compatibility	Guest OS codified to issue Hypercalls so it can't run on Native Hardware or other Hypervisors Poor compatibility; Not available on Windows OSes
Performance	Good	Fair Current performance lags Binary Translation virtualization on various workloads but will improve over time	Better in certain cases
Used By	VMware, Microsoft, Parallels	VMware, Microsoft, Parallels, Xen	VMware, Xen
Guest OS Hypervisor Independent?	Yes	Yes	XenLinux runs only on Xen Hypervisor VMI-Linux is Hypervisor agnostic

Full Virtualization vs. Paravirtualization

- 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 at run time. In paravirtualization, these instructions are handled at compile time when the non-virtualizable OS instructions are replaced with hypercalls.

- The advantage of paravirtualization is lower virtualization overhead, but the performance advantage of paravirtualization over full virtualization can vary greatly depending on the workload. Most user space workloads gain very little, and near native performance is not achieved for all workloads.

- As paravirtualization cannot support unmodified operating systems (e.g. Windows 2000/XP), its compatibility and portability is poor.