Virtualization

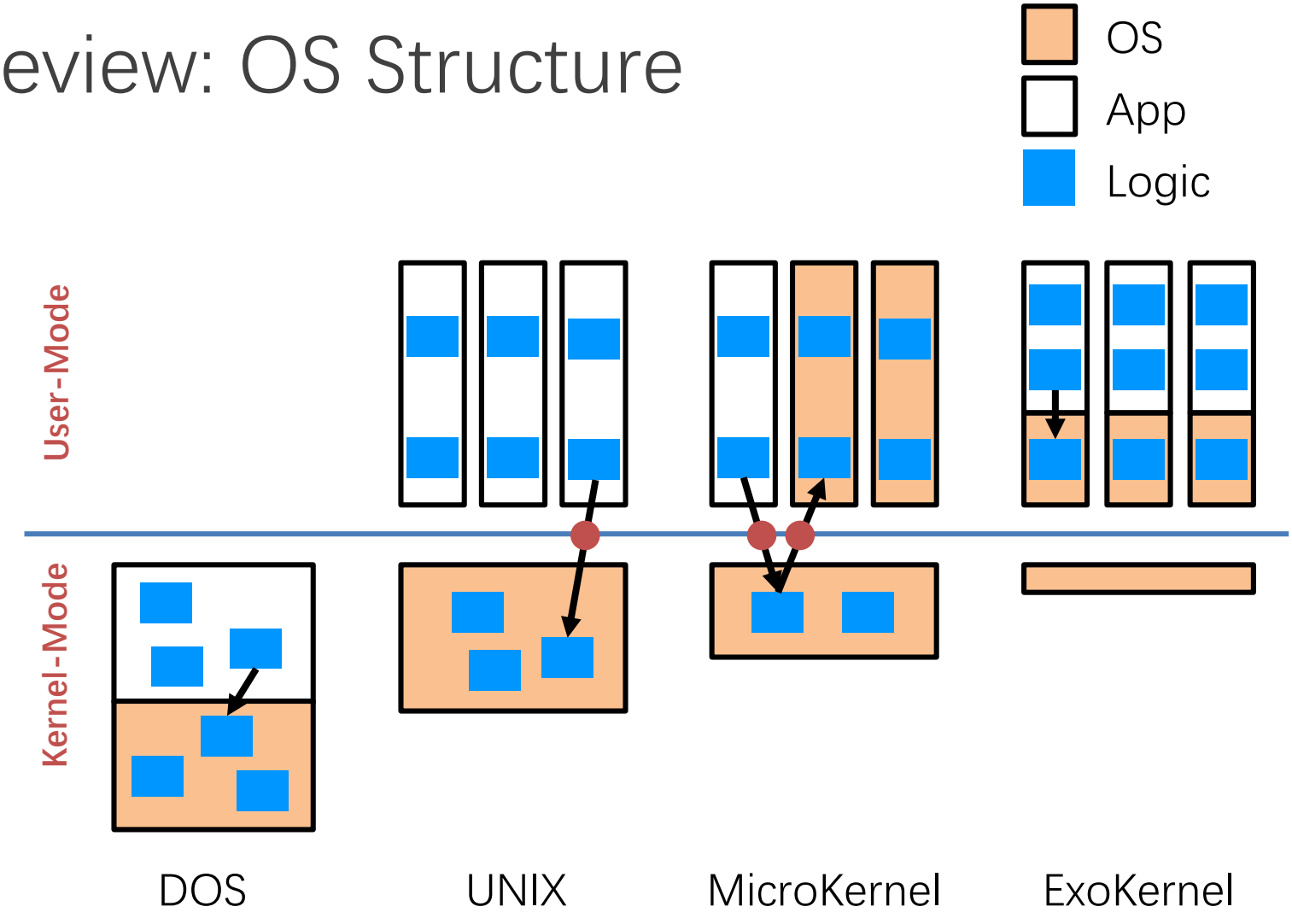
Virtualizing CPU, Memory and Devices

Yubin Xia

When Do You Use Virtual Machine?

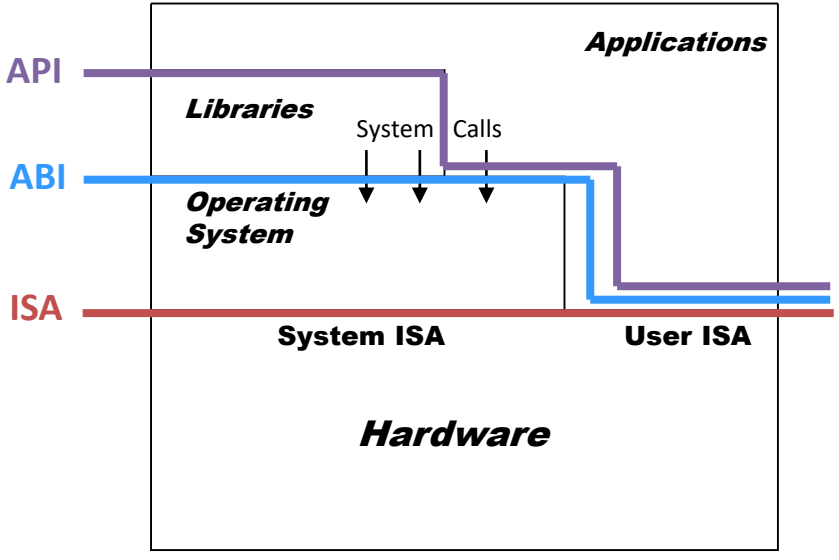
- VM in daily-use:
  - Debug, especially for OS
  - Sandbox some untrusted apps
- VM can also be used to:
  - Multiplex physical machine, e.g., in cloud
  - Deploy complex software in a VM image
  - Make multiple machines look like one
  - Enhance security, fault tolerance, etc.

Review: OS Structure



Virtualization Layers

- At which layer?
  - Hello world
  - Web game
  - Dota
  - Office 2013
  - Windows 8
  - Java applications
  - Python scripts
  - High Sierra



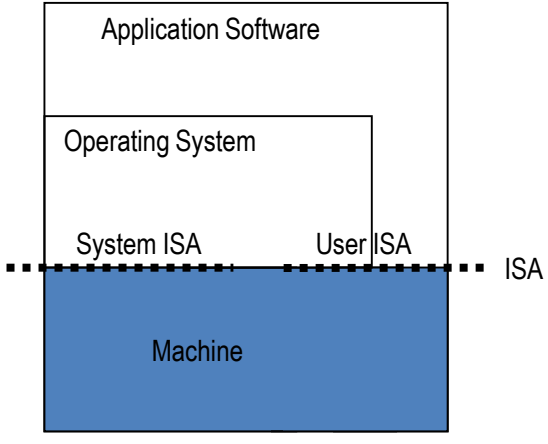
API – application programming interface

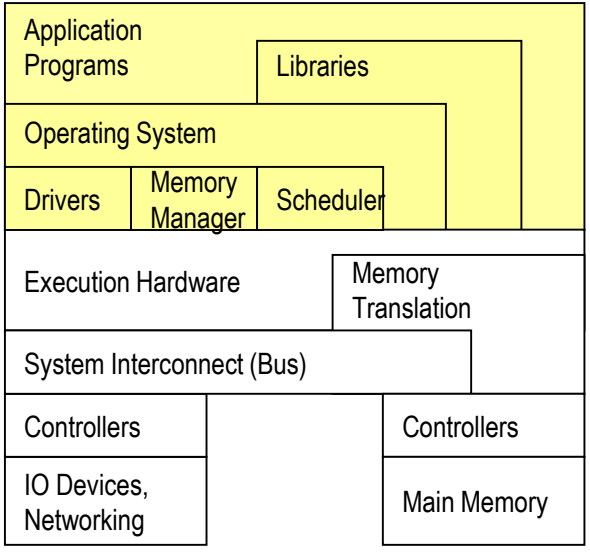
ABI – application binary interface

ISA – instruction set architecture

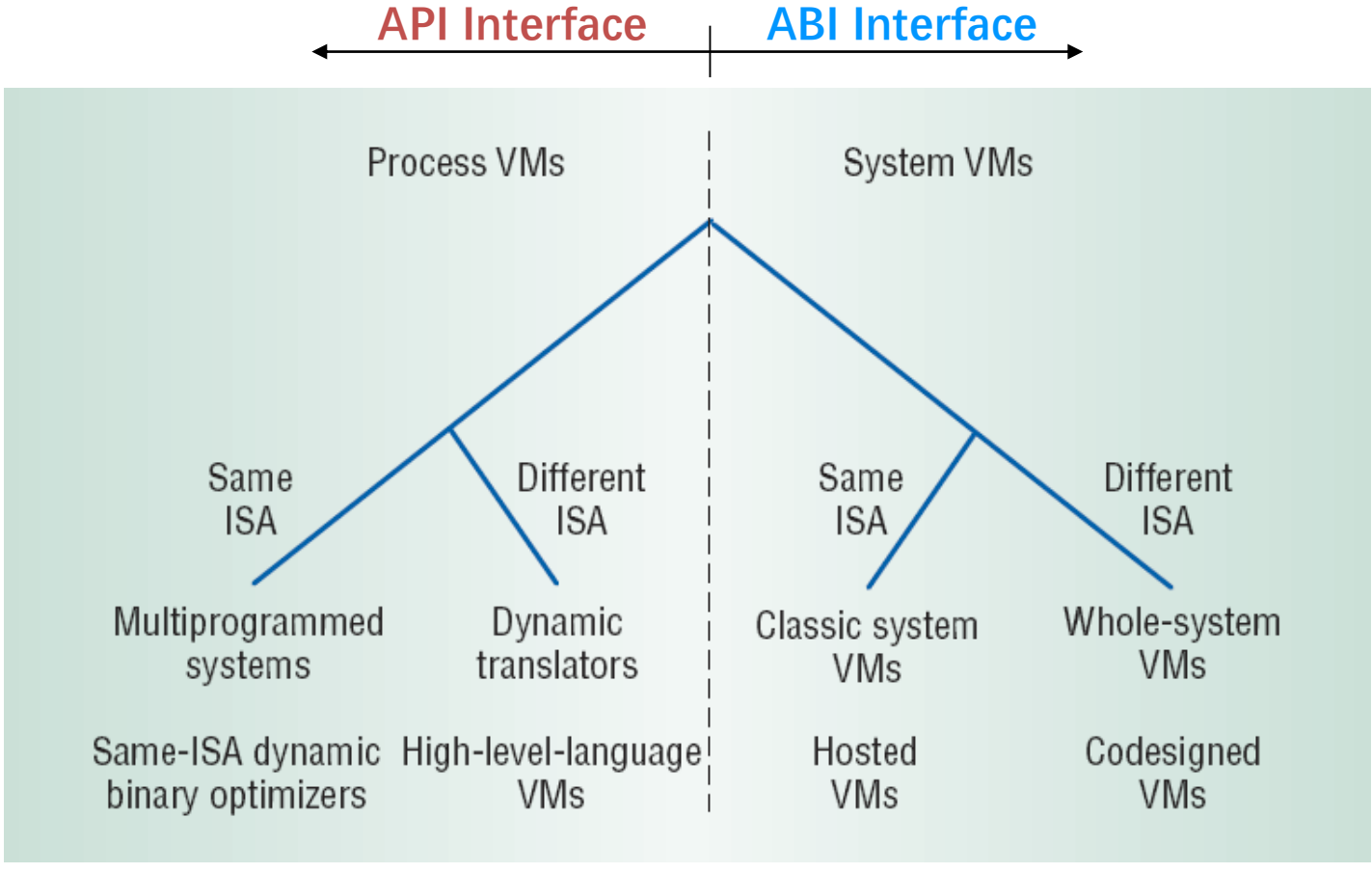
What is a VM and Where is the VM?

- Machine from the perspective of a system
  - ISA provides interface between system and machine

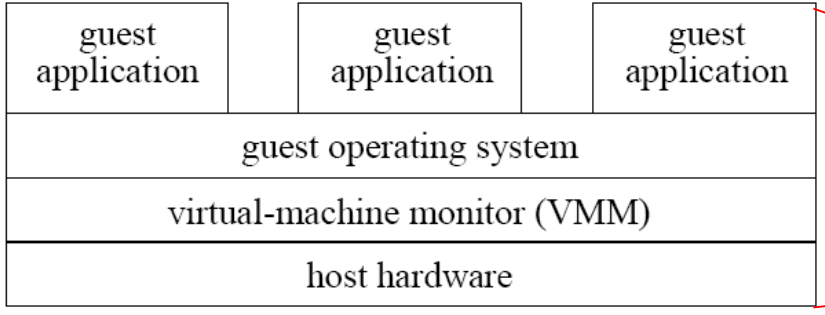




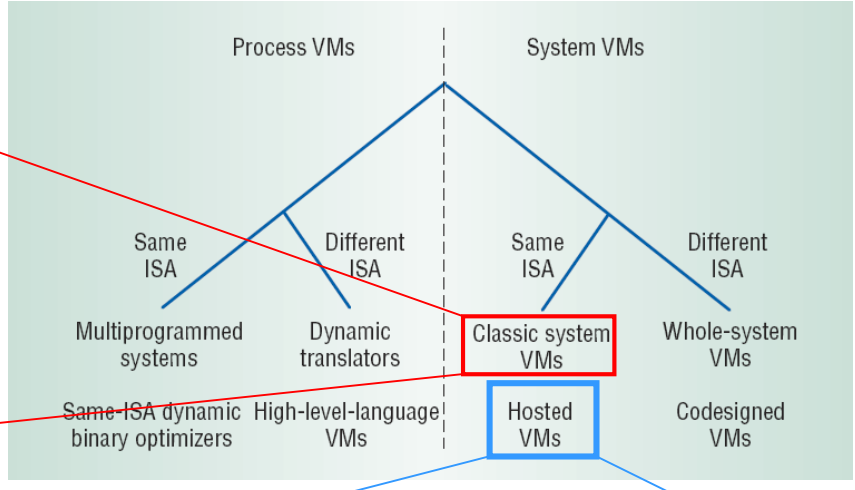
Design Space (Level vs. ISA)



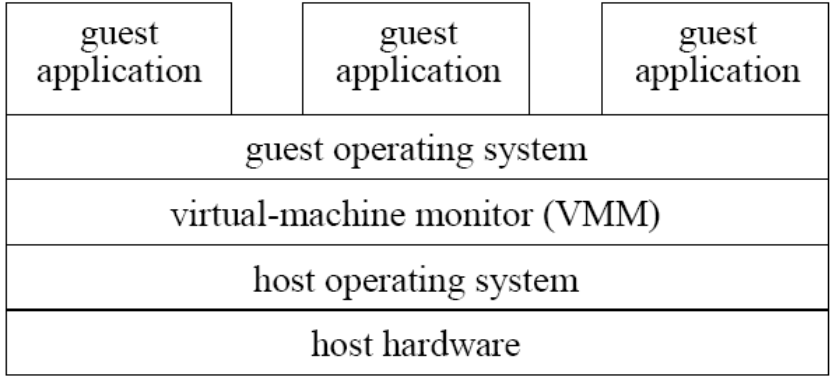
System VMMs



Type 1



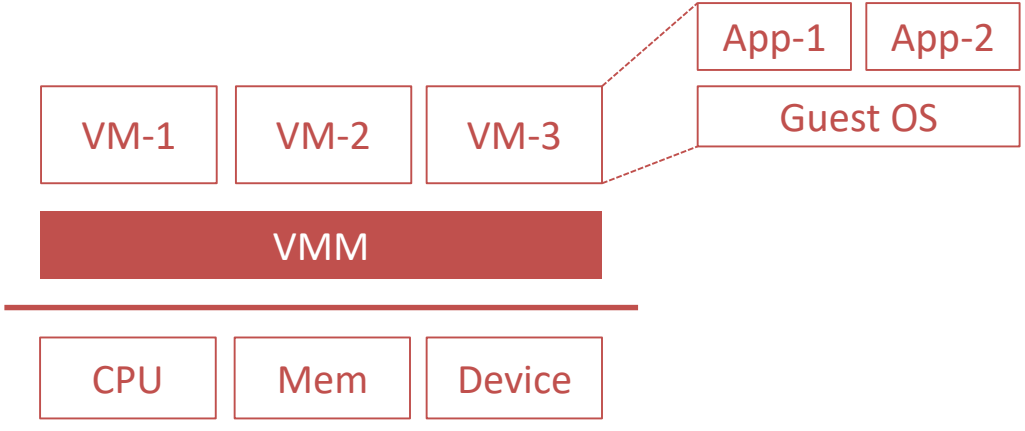
- **Type 1**: runs directly on hardware
  - High performance
  - e.g., Xen, VMware ESX Server
- **Type 2**: runs on Host OS
  - Ease of construction/installation
  - e.g., VMware Workstation



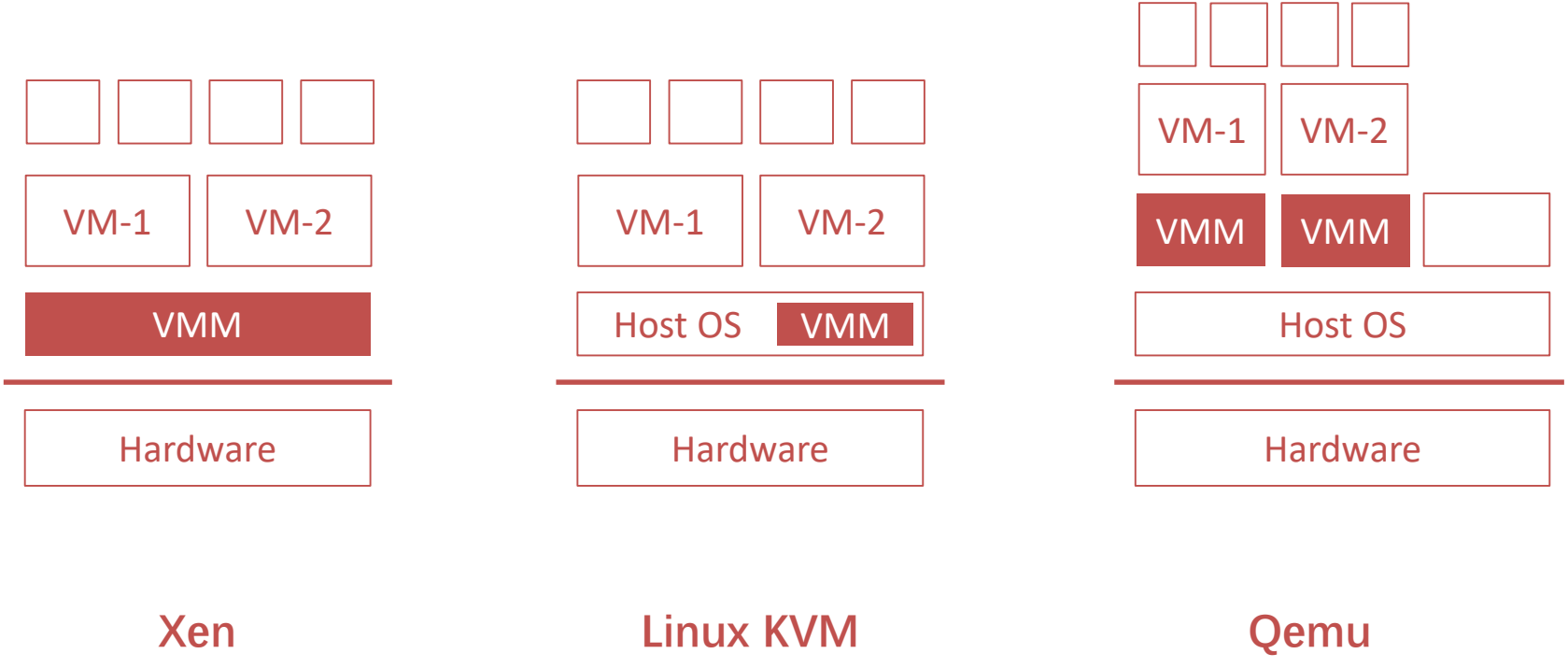
Type 2

VMM (Virtual Machine Monitor)

- Why do we need a VMM?
  - For managing VMs which running guest OS
  - VMM runs underlying VMs (higher privilege)
- Virtualize hardware
  - CPU
  - Memory
  - Device

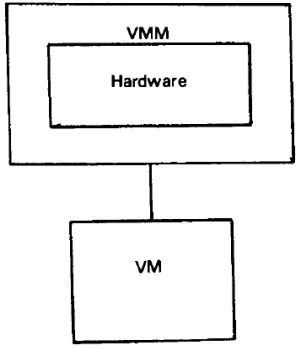


Different Architectures of VMM



Principles in 1974

Fig. 1. The virtual machine monitor.



"an *efficient, isolated duplicate* of the real machine"

- **Efficiency**
  - Innocuous instructions should execute directly on hardware
- **Resource control**
  - Executed programs may not affect the system resources
- **Equivalence**
  - The behavior of a program executing under the VMM should be the same as if the program were executed directly on the hardware (except possibly for timing and resource availability)

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

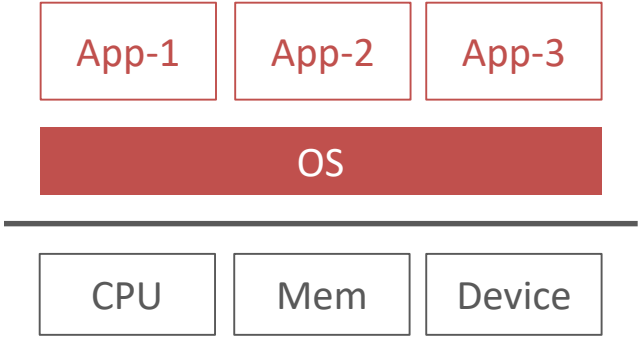
Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

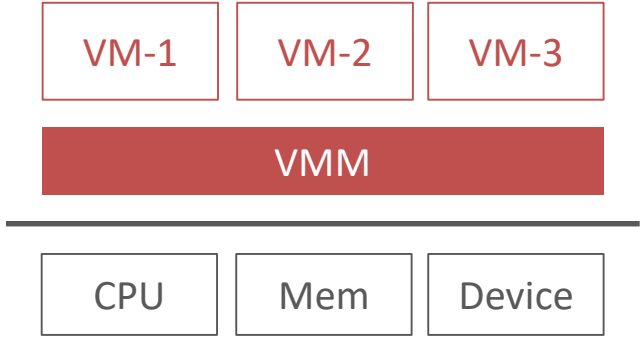
Communications of the ACM, vol 17, no 7, 1974, pp.412-421

CPU VIRTUALIZATION

What are the Differences between OS & VMM?

- Similarities
  - Multiplex hardware
  - Higher privilege
- Differences
  - Different abstraction
  - VMM schedules VMs, OS schedules processes



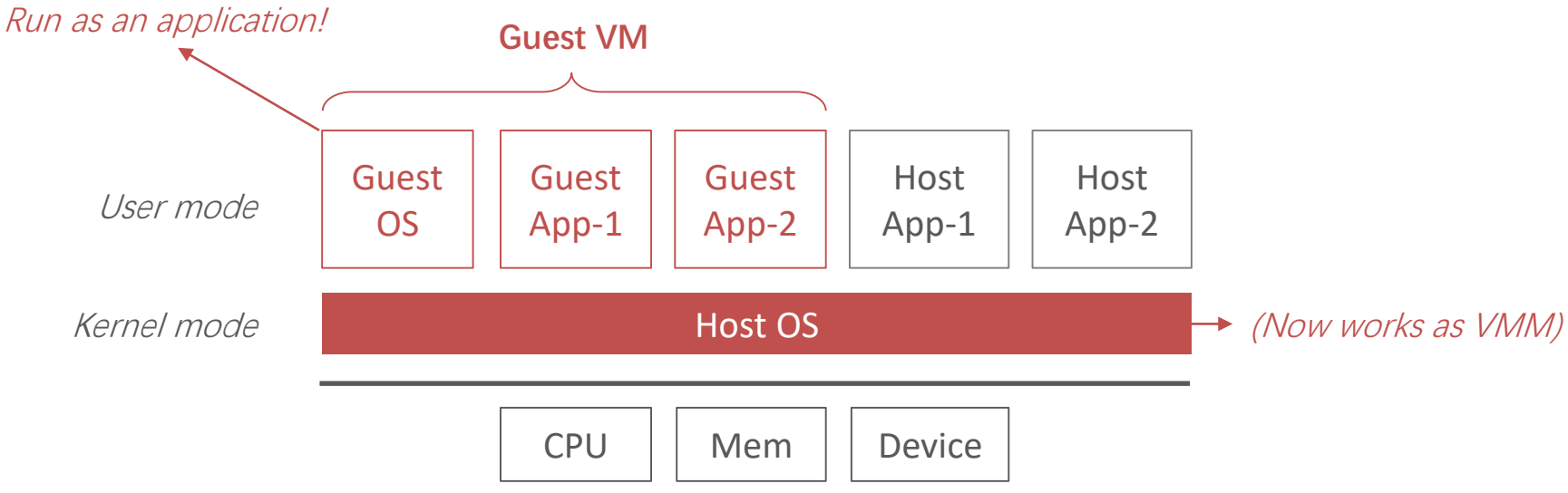


CPU Virtualization: Process and VM

- Each process thinks it has the entire CPU
  - Does not care other processes
- OS schedules the processes
  - OS splits the CPU time to time-slices
  - Schedule each process preemptively
  - Save context, find next, restore context

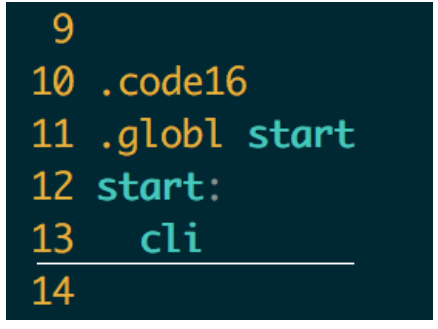
- Why not run a VM as a process?

First Try: OS on OS



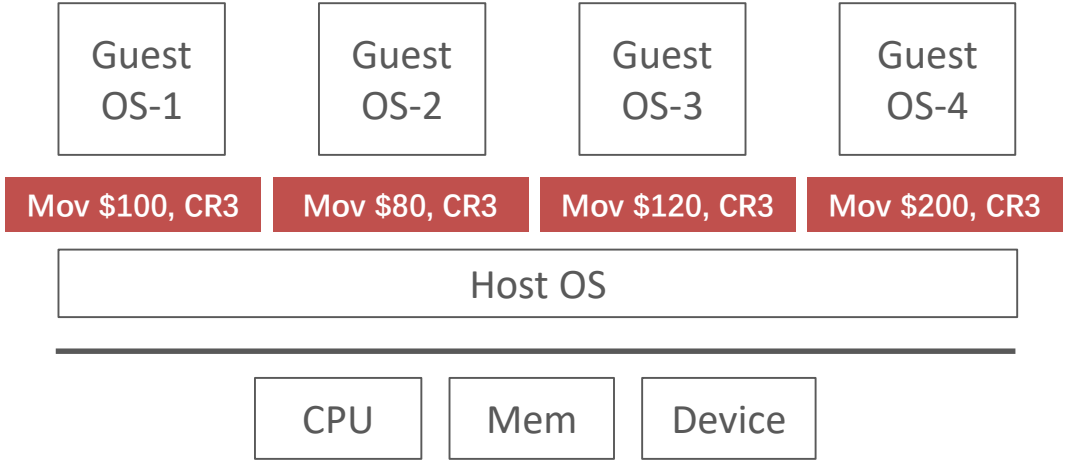
Run OS as an Application

- Stuck at the very first instruction
  - *cli* : disable interrupt
  - It's a privilege instruction
  - Cannot run in user mode!
- Similar instructions
  - E.g., change CR3, set IDT, etc.
  - These instructions will change machine states



xv6: bootasm.S

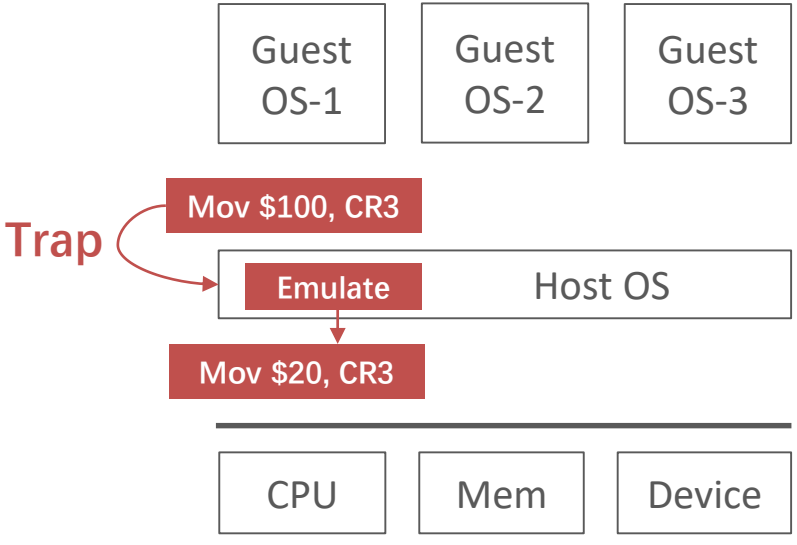
Challenges to Virtualized System ISA



- Guest OS executes privilege instructions
- Guest OS are not allowed to run the privilege instructions

Solution: Trap & Emulate

- **Trap**: running privilege instructions will trap to the VMM
- **Emulate**: those instructions are implemented as functions in the VMM
- System states are kept in VMM's memory, and are changed according



Now We Can Run OS as an Application

- Host OS virtualizes system states
  - Save guest system states in memory
- When a guest OS runs **cli**
  - CPU will trap to the host OS (trap)
  - Host OS will change the IF bit (emulate)
- How does a host OS deliver interrupt to a guest OS?
  - Similar as delivering a **signal** to an application
- Problem solved? No…

```
 9
10 .code16
11 .globl start
12 start:
13   cli
14
```

xv6: bootasm.S

Problems of Trap & Emulate

- Not all architectures are "**strictly virtualizable**"
- An ISA is strictly virtualizable if, when executed in a lesser privileged mode:
  - All instructions that access privileged state trap
  - All instructions either trap or execute identically
- Trap costs may be high

X86 is not Strictly Virtualizable

- Example: the ***popf*** instruction
  - *popf* takes a word off the stack and puts in into the flags register
    - One flag in that register is the interrupt enable flag (IF)
  - At system level the IF flag is updated by popf
  - At user level the IF flag is *not* updated, and CPU **silently** drops updates to the IF
- There **17** such instructions in X86
  - SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET, STR, MOV

How to Deal with the 17 Instructions?

1. **Instruction Interpretation**: emulate them by software
2. **Binary translation**: translate them to other instructions
3. **Para-virtualization**: replace them in the source code
4. **New hardware**: change the CPU to fix the behavior

Sol-1: Instruction Interpretation

- Emulate **Fetch/Decode/Execute** pipeline in software
  - Emulate all the system status using memory
    - E.g., using an array `GPR[8]` for general purpose registers
  - None guest instruction executes directly on hardware
- E.g., Bochs

Example: Virtualizing Interrupt Flag w/ Instruction Interpreter

```
void CPU_Run(void)
{
   while (1) {
      inst = Fetch(CPUState.PC);

      CPUState.PC += 4;

      switch (inst) {
      case ADD:
         CPUState.GPR[rd] = GPR[rn] + GPR[rm];
         break;
      …
      case CLI:
         CPU_CLI(); break;
      case STI:
         CPU_STI(); break;
      }

      if (CPUState.IRQ && CPUState.IE) {
         CPUState.IE = 0;
         CPU_Vector(EXC_INT);
      }
   }
}
```

```
void CPU_CLI(void)
{
   CPUState.IE = 0;
}


void CPU_STI(void)
{
   CPUState.IE = 1;
}


void CPU_Vector(int exc)
{
   CPUState.LR = CPUState.PC;
   CPUState.PC = disTab[exc];
}
```

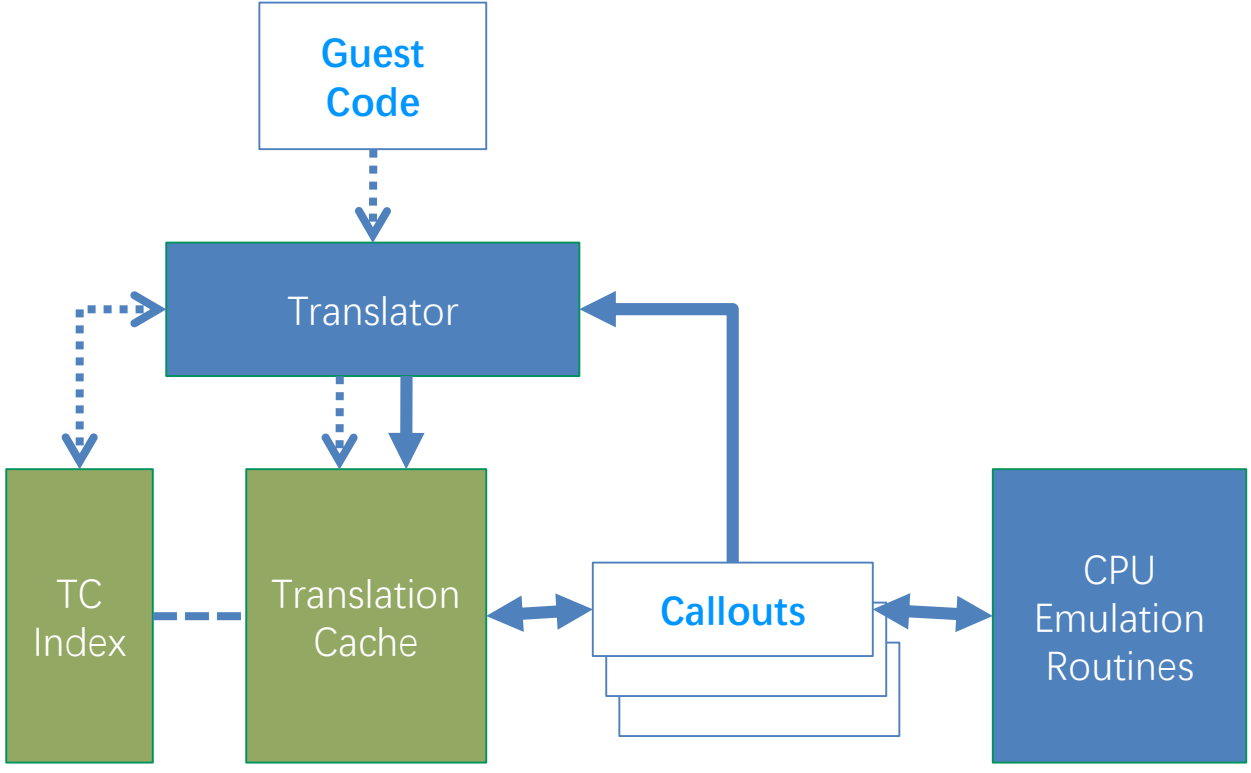
Instruction Interpretation

- Positives
  - Easy to implement & minimal complexity
- Negatives
  - **Very slow!**

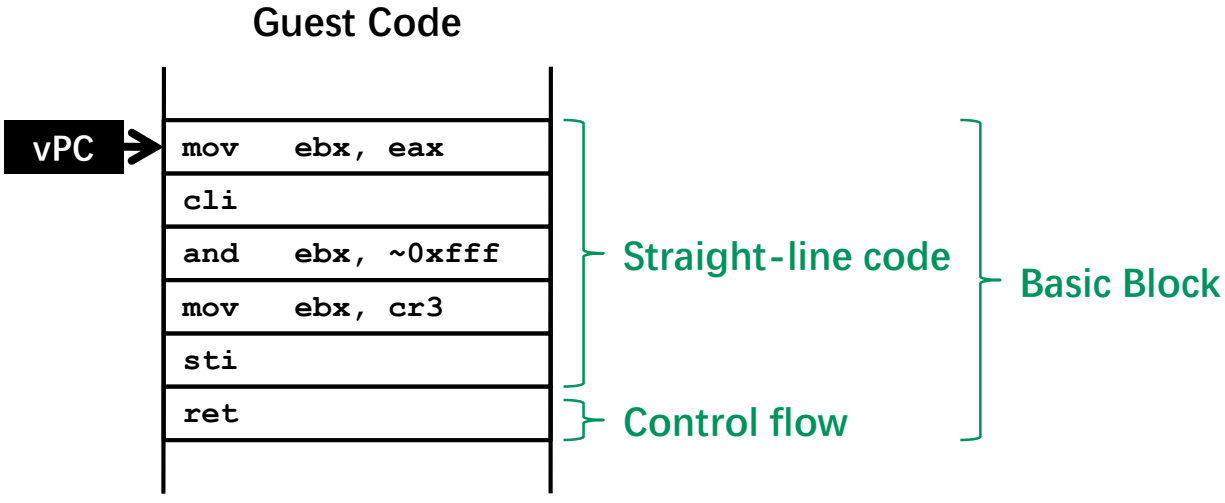
Sol-2: Binary Translator

- Translate before execution
  - Translation unit is basic block (why?)
  - Each basic block -> code cache
  - Translate the 17 instructions to function calls
    - Implemented by the VMM
- E.g., VMware, Qemu

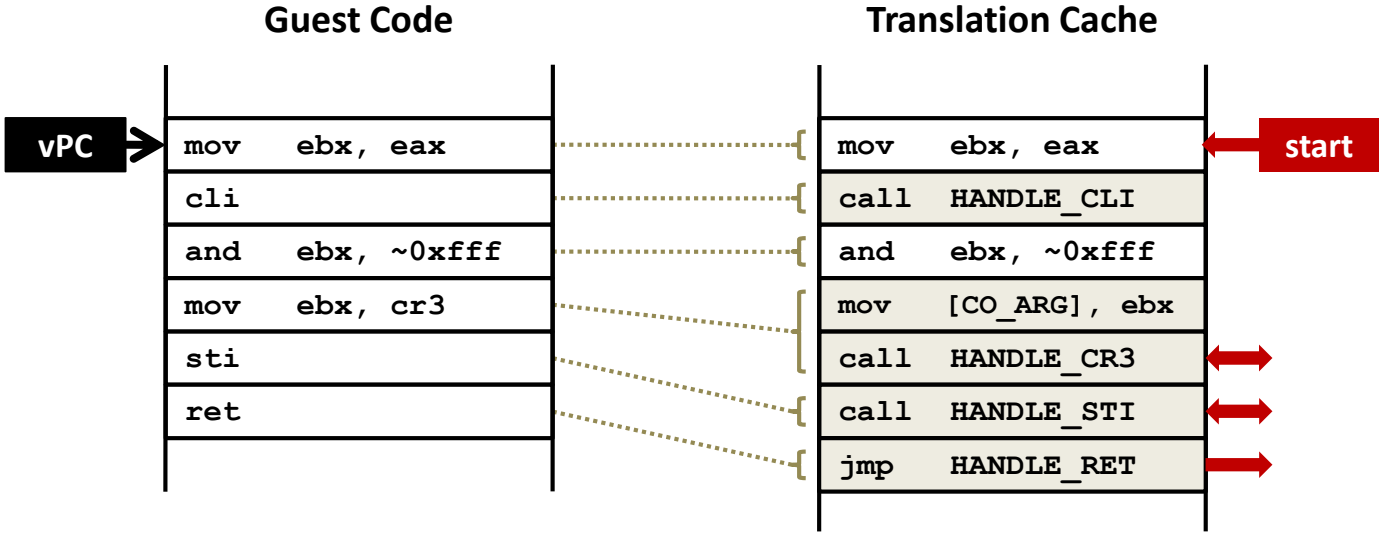
Architecture of a Binary Translator



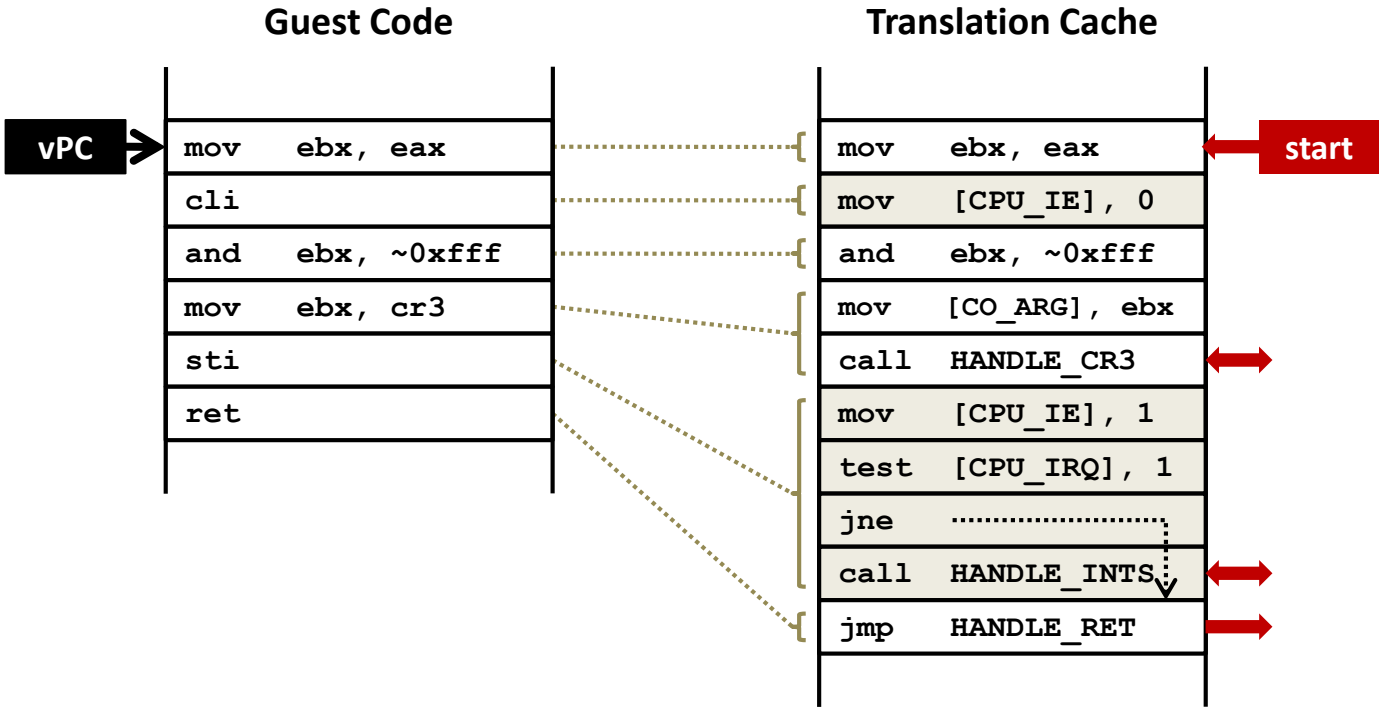
Basic Blocks



Binary Translation



Binary Translation



Basic Binary Translator

```
void BT_Run(void)
{
   CPUState.PC = _start;
   BT_Continue();
}

void BT_Continue(void)
{
   void *tcpc;

   tcpc = BTFindBB(CPUState.PC);

   if (!tcpc) {
      tcpc = BTTranslate(CPUState.PC);
   }

   RestoreRegsAndJump(tcpc);
}
```

```
void *BTTranslate(uint32 pc)
{
   void *start = TCTop;
   uint32 TCPC = pc;

   while (1) {
      inst = Fetch(TCPC);
      TCPC += 4;

      if (IsPrivileged(inst)) {
         EmitCallout();
      } else if (IsControlFlow(inst)) {
         EmitEndBB();
         break;
      } else {
         /* ident translation */
         EmitInst(inst);
      }
   }

   return start;
}
```

Basic Binary Translator – Part 2

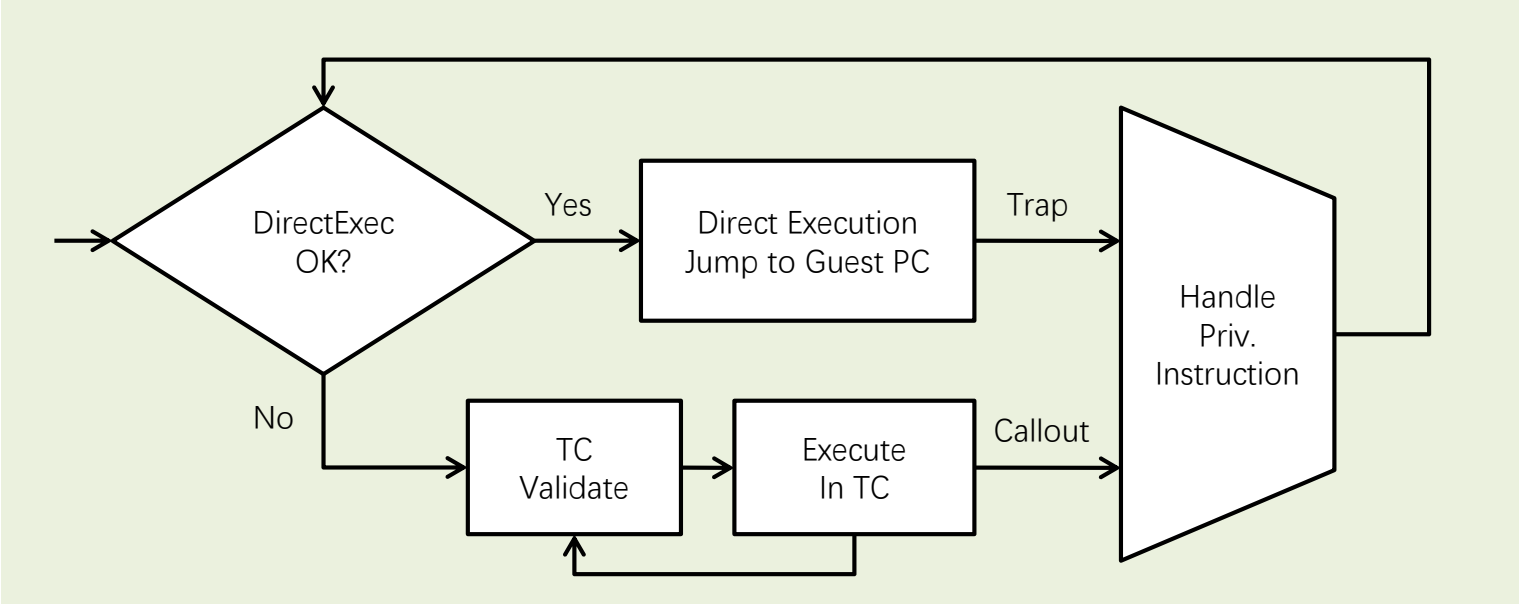
```
void BT_CalloutSTI (BTSavedRegs regs)
{
   CPUState.PC = BTFindPC(regs.tcpc);
   CPUState.GPR[] = regs.GPR[];

   CPU_STI();

   CPUState.PC += 4;

   if (CPUState.IRQ
         && CPUState.IE) {
      CPUVector();
      BT_Continue();
      /* NOT_REACHED */
   }

   return;
}
```

Issues with Binary Translation

- PC synchronization on interrupts
  - Now interrupt will only happen at basic block boundary
  - But on real machine, interrupt may happen at any instruction

- Carefully handle self-modifying code (SMC)
  - Notified on writes to translated guest code

Hybrid Approach



- Binary translation for the kernel
- Direct execution (trap & emulate) for the applications

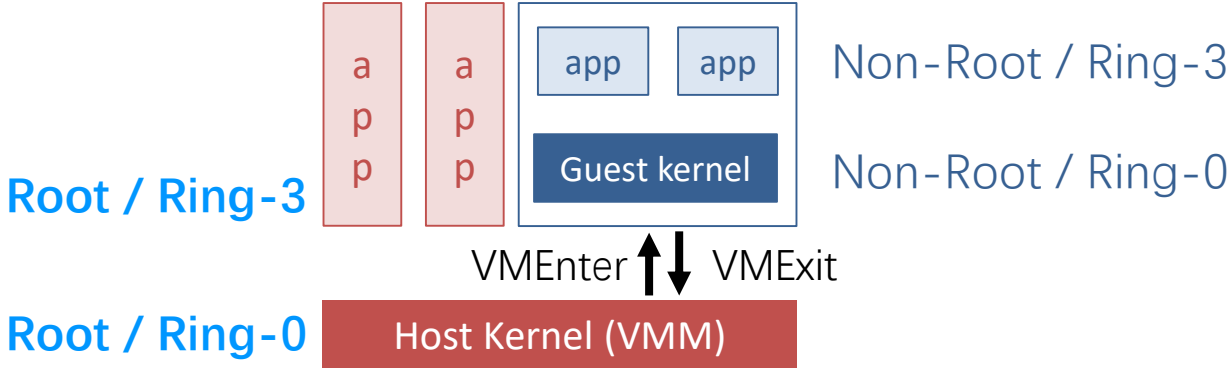
Sol-3: Para-virtualization

- Modify OS and let it cooperate with the VMM
  - Change sensitive instructions to calls to the VMM
    - Also known as **hypercall**
  - Hypercall can be seen as trap
- E.g., Xen
  - Widely used by industry like Amazon's EC2

Sol-4: Hardware Supported CPU Virtualization

- VMX **root** operation:
  - Full privileged, intended for Virtual Machine Monitor
- VMX **non-root** operation:
  - Not fully privileged, intended for guest software

Both forms of operation support all four privilege levels from 0 to 3

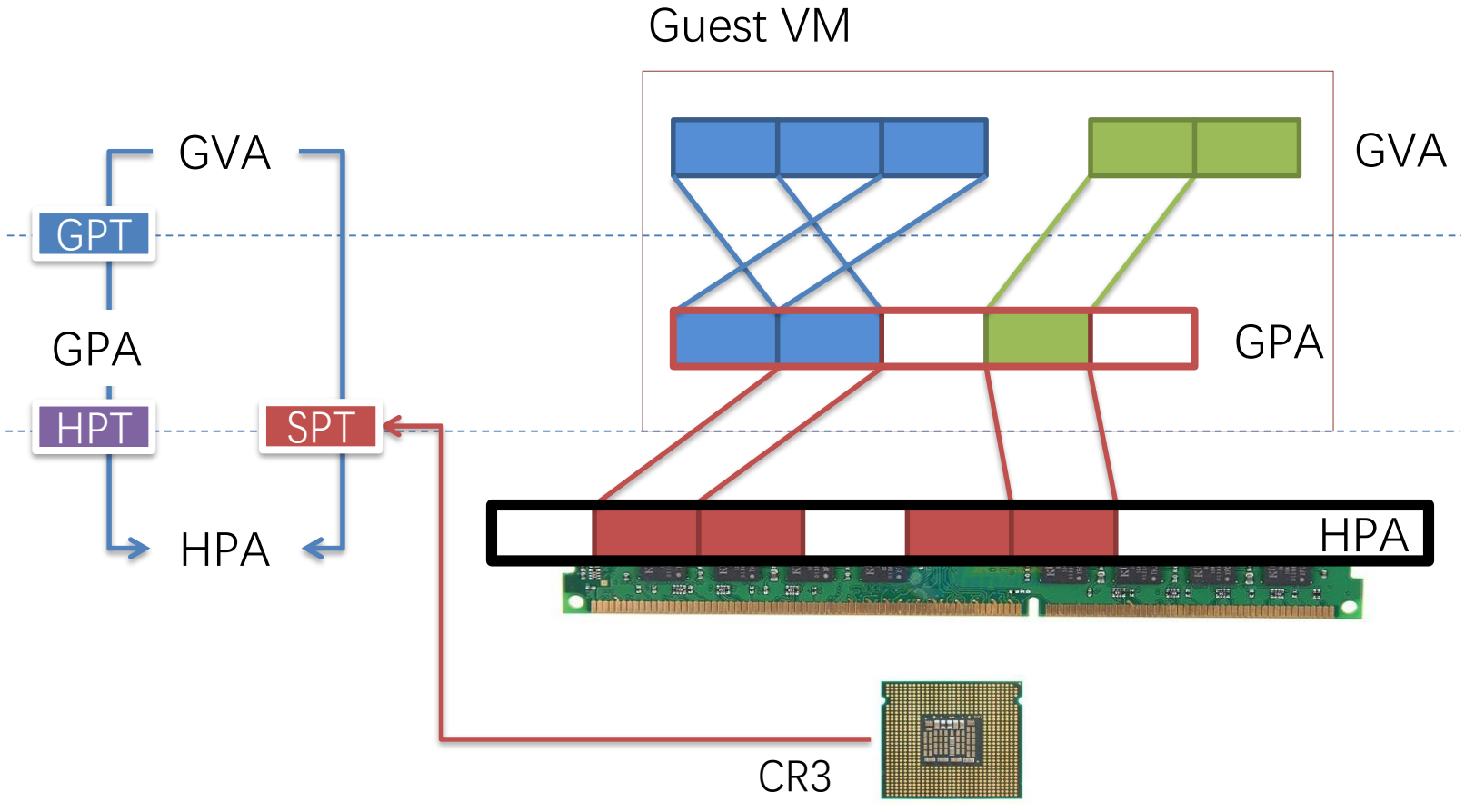


MEMORY VIRTUALIZATION

Virtualizing the Page Tables

- Terminology: 3 types of address now
  - **GVA->GPA->HPA** (Guest virtual. Guest physical. Host physical)
  - Guest VM's page table contains GPA
- Setting CR3 to point to guest page table would not work
  - E.g., a processes in VM might access host physical address 0~1GB, which might not belong to that guest VM
  - Solution-1: **shadow paging**
  - Solution-2: **direct paging**
  - Solution-3: **new hardware**

Solution-1: Shadow Paging



Two Page Tables Become One

1. VMM intercepts guest OS setting the virtual CR3
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow page table

Setup Shadow Page Table

```
set_cr3 (guest_page_table):
    for GVA in 0 to 220
        if guest_page_table[GVA] & PTE_P:
            GPA = guest_page_table[GVA] >> 12
            HPA = host_page_table[GPA] >> 12
            shadow_page_table[GVA] = (HPA<<12)|PTE_P
        else
            shadow_page_table = 0
     CR3 = PHYSICAL_ADDR(shadow_page_table)
```

Question

- Assume that:
  - There are 10 VMs running on a machine
  - Each VM contains 10 applications
- **Q**: how many shadow page tables in total?
  - Shadow page tables are per application
  - Guest page tables are per application
  - Host page tables are per VM

What if Guest OS Modifies its Own Page Table?

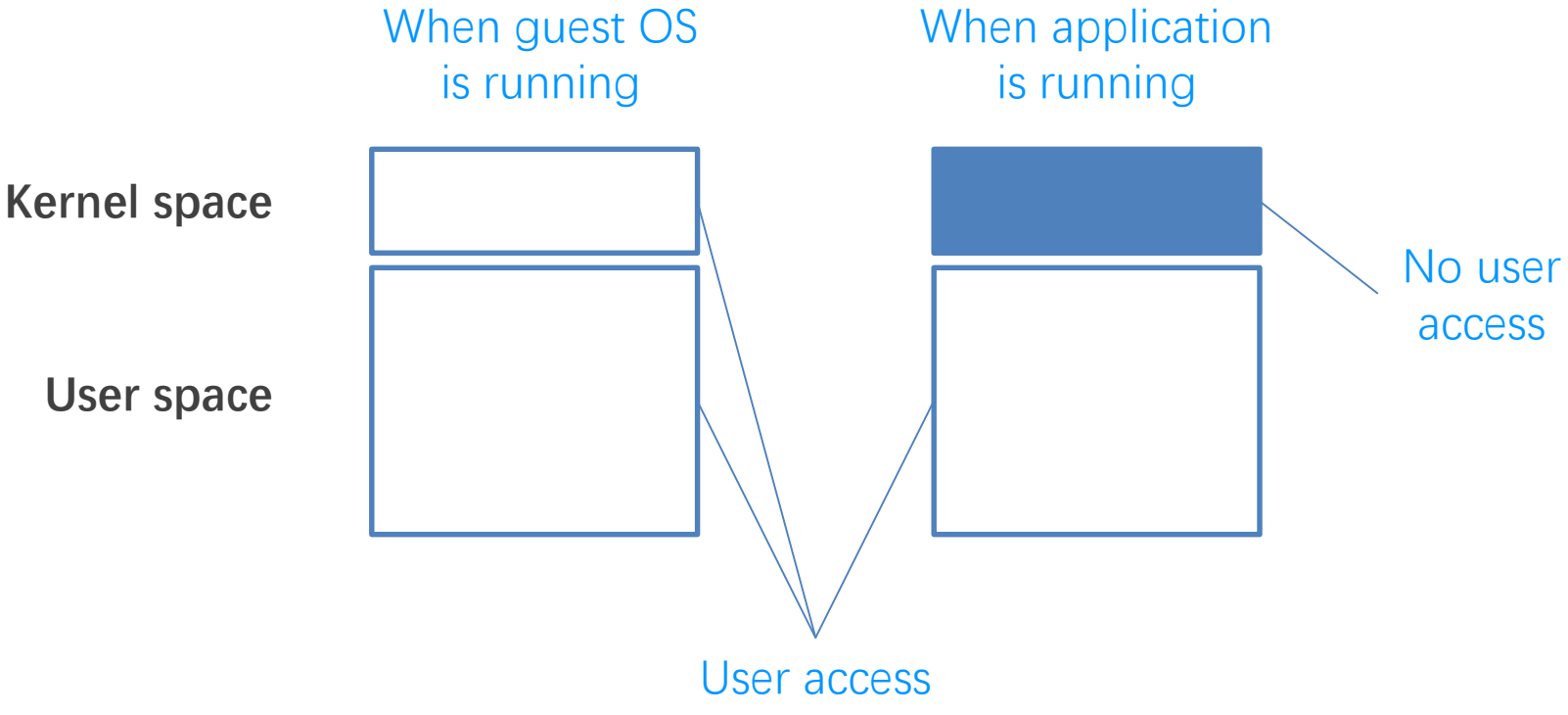
- It will have no effect!
  - Since CR3 is now pointing to the shadow page table
  - Need to synchronize the shadow page table with guest page table
- Solution
  - **VMM needs to intercept when guest OS modifies page table, and updates the shadow page table accordingly**

1. Mark the guest table pages as read-only (in the shadow page table)
2. If guest OS tries to modify its page tables, it triggers page fault
3. VMM handles the page fault by updating shadow page table

What if a Guest App Access its Kernel Memory?

- Remember that now the kernel is running in user mode
  - It means any application may also access guest kernel's memory
  - How do we selectively allow / deny access to kernel-only pages?
- **One solution:** split a shadow page table to two tables
  - Two shadow page tables, one for user, one for kernel
  - When guest OS switches to user mode, VMM will switch the shadow page table as well, vice versa
  - Recall trap & emulate

Two Memory Views of Guest VM



The Same Question

- Assume that:
  - There are 10 VMs running on a machine
  - Each VM contains 10 applications
- **Q**: now, how many shadow page tables in total?

Sol-2: Direct Paging (Para-virtualization)

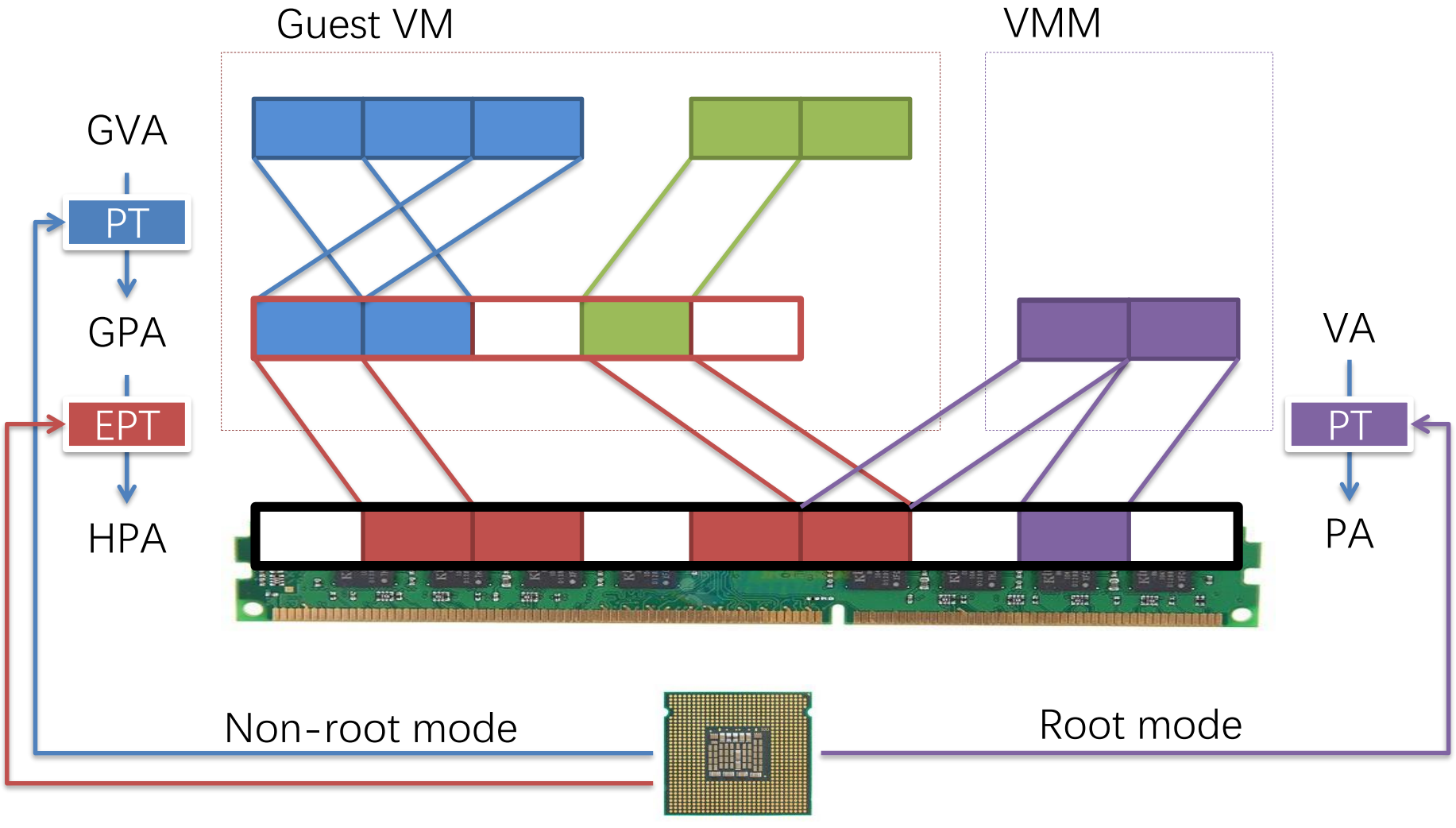
- Modify the guest OS
  - No GPA is needed, just GVA and HPA
  - Guest OS directly manages its HPA space
  - Use hypercall to let the VMM update the page table
  - The hardware CR3 will point to guest page table
- VMM will check all the page table operations
  - The guest page tables are read-only to the guest

Sol-2: Direct Paging (Para-virtualization)

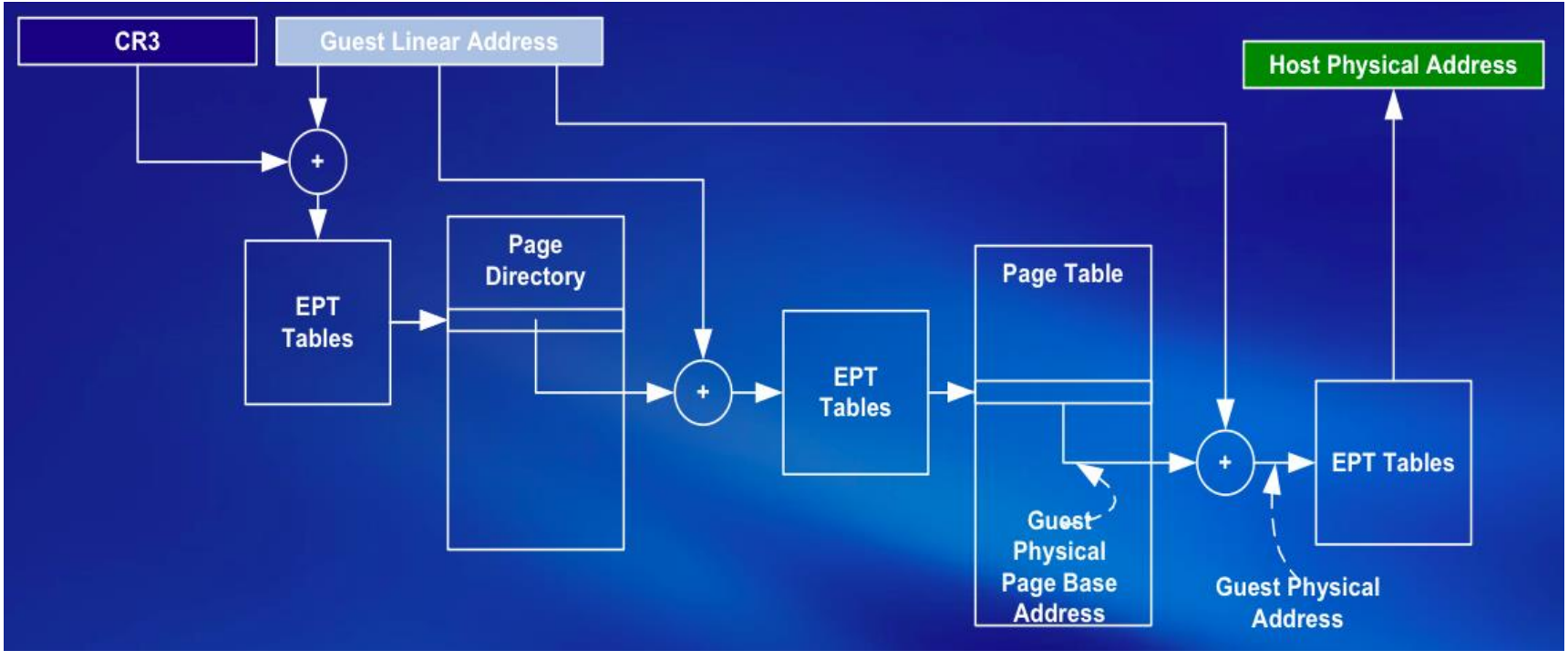
- Positive
  - Easy to implement and more clear architecture
  - Better performance: guest can batch to reduce trap
- Negatives
  - Not transparent to the guest OS
  - The guest now knows much info, e.g., HPA
    - May use such info to trigger *rowhammer* attacks

Sol-3: Hardware Supported Memory Virtualization

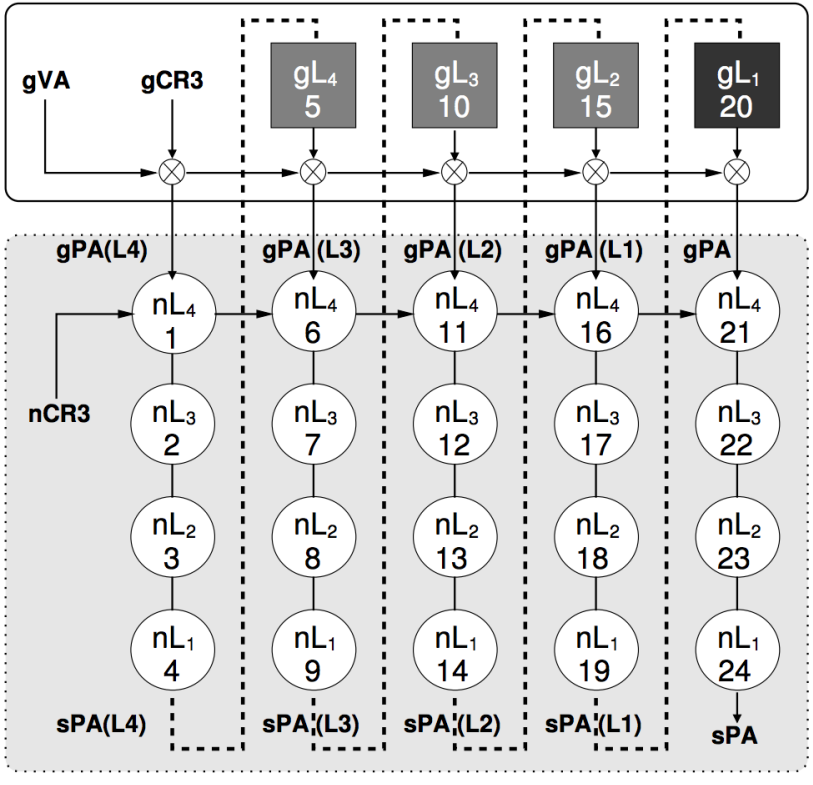
- Hardware implementation
  - Intel's EPT (Extended Page Table)
  - AMD's NPT (Nested Page Table)
- Another table
  - EPT for translation from **GPA to HPA**
  - EPT is controlled by the hypervisor
  - EPT is per-VM


Guest VM
VMM
GVA
PT
GPA
EPT
HPA
VA
PT
PA
Non-root mode
Root mode

EPT Translation: Details



EPT Increases Memory Access



sPA == HPA
nCR3 == hCR3
nL_1 == hL_1

One memory access from the guest VM may lead up to ***20 memory accesses***!

I/O VIRTUALIZATION

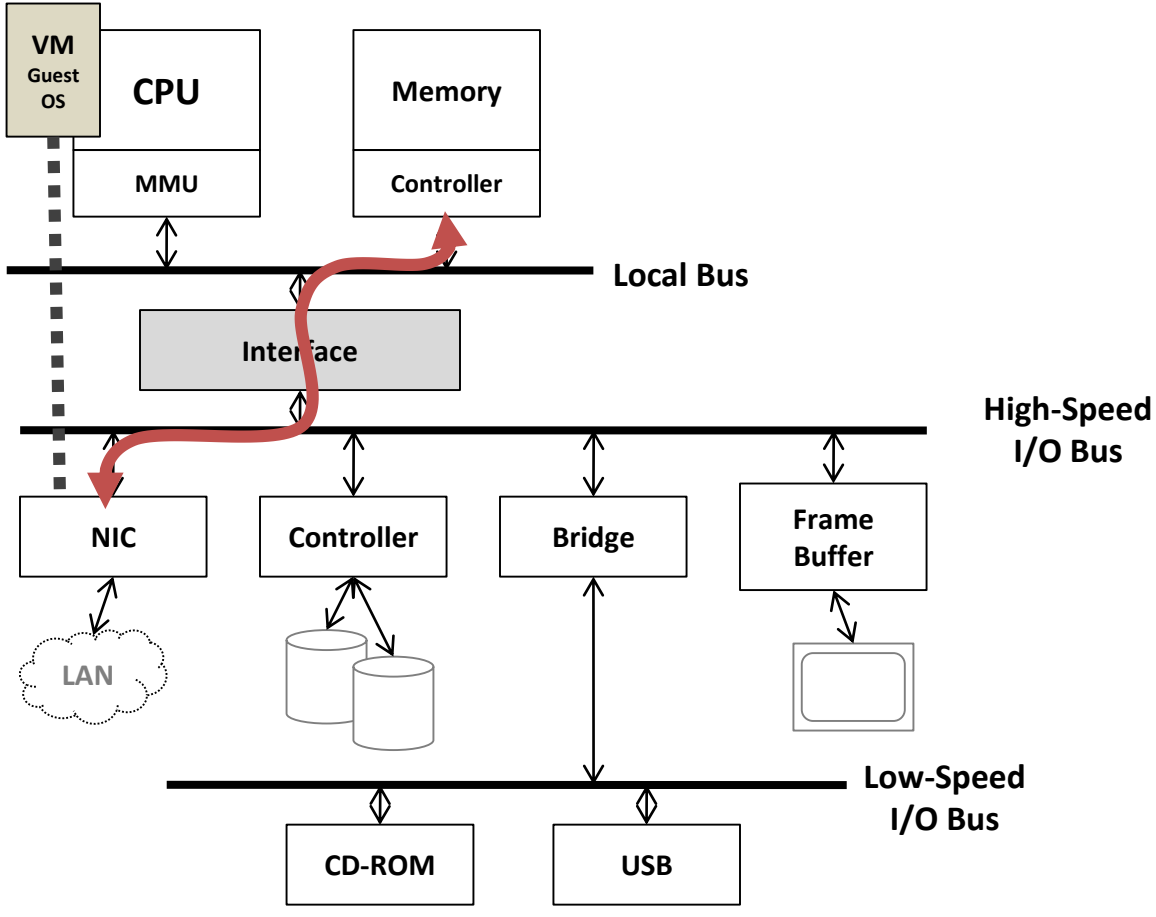
I/O Virtualization

- Goal
  - Multiplexing device to guest VMs
- Challenges
  - Each guest OS has its own driver
  - How can one device be controlled by multiple drivers?
  - What if one guest OS tries to format its disk?

Solutions for I/O Virtualization

1. **Direct access**: VM owns a device exclusively
2. **Device emulation**: VMM emulates device in software
3. **Para-virtualized**: split the drivers to guest and host
4. **Hardware assisted**: self-virtualization device

Sol-1: Direct Access Device Virtualization

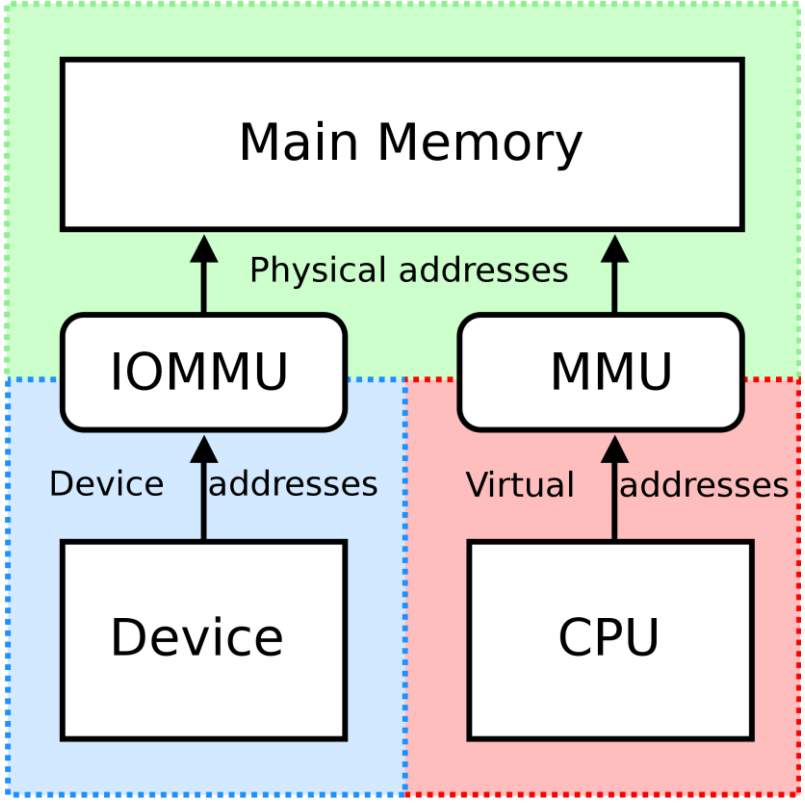


IOMMU: Page Tables for Devices

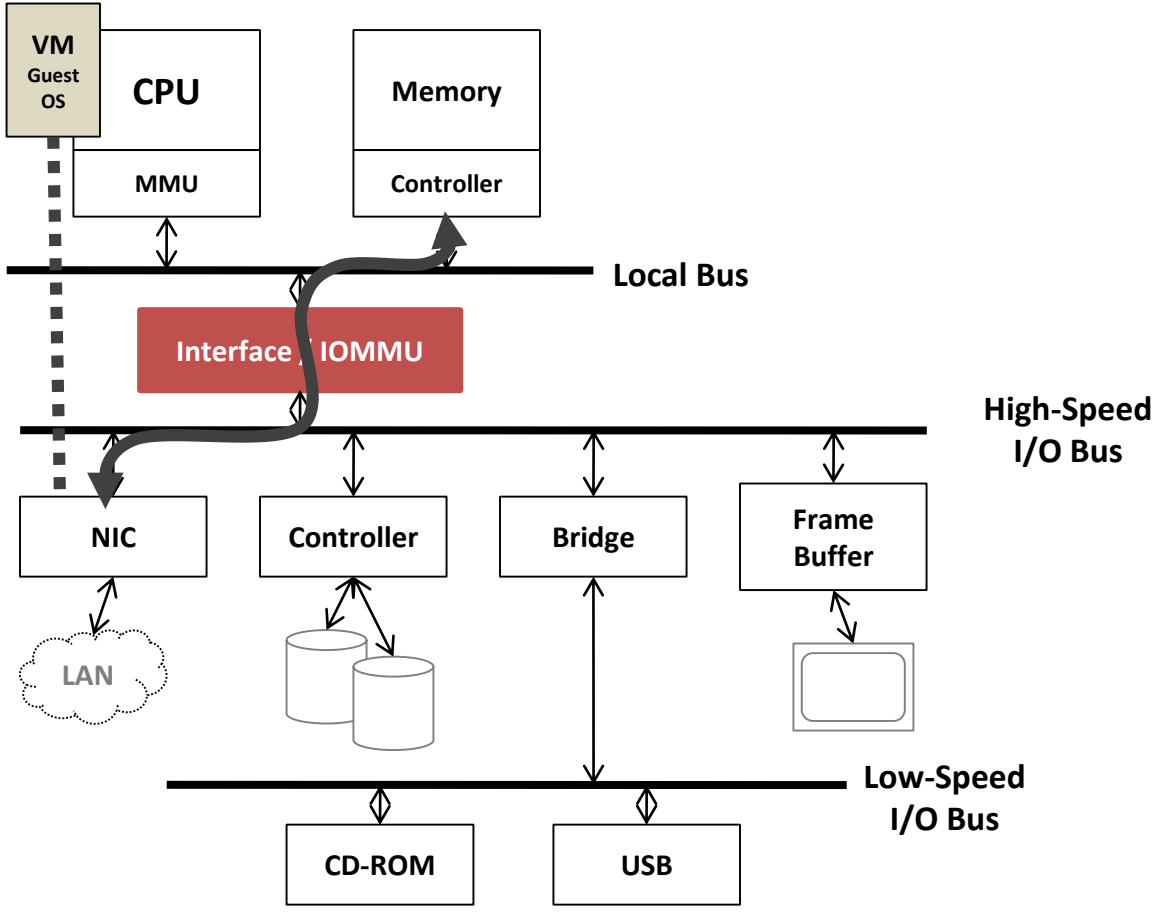
- Allow guest OS direct access to underlying device
  - Guest just reuses its own device driver
- **Q:** What if a VM asks device to access memory of other VMs?
  - It is possible because device accesses HPA in DMA
  - Thus a device can access any memory
- Solution: Page tables for devices
  - Another MMU: IOMMU for devices
  - Q: what addresses will IOMMU translate?

IOMMU for Direct Access

- VT-d architecture defines a multi-level page-table structure for DMA address translation
- Similar to IA-32 processor page-tables, enabling software to manage memory at 4 KB or larger page granularity



Direct Access Device Virtualization



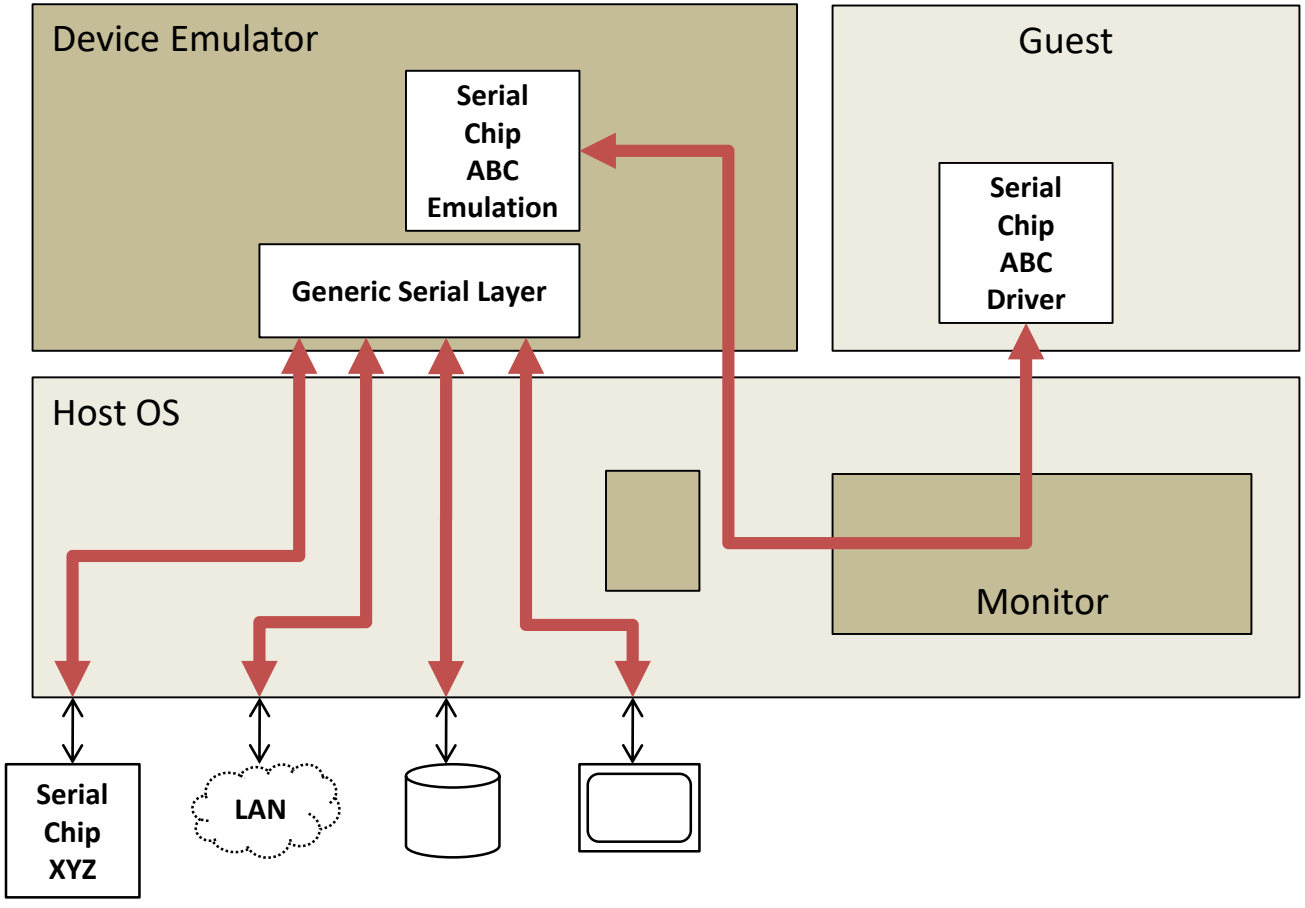
Direct Access Device Virtualization

- Positives
  - Fast, since the VM uses device just as native machine
  - Simplify monitor: limited device drivers needed
- Negatives
  - Hardware interface visible to guest (bad for migration)
  - Interposition is hard by definition (no way to trap & emulate)
  - Now you need much more devices! (image 100 VMs)

Sol-2: Emulating Devices

- Emulate a device
  - Implement device logic in pure software
  - Can even emulate non-existing devices
- For example
  - VMware emulates a 8139 network card for VMs
  - Since 8139 is widely used, almost every OS has its driver
  - So no need to write a new driver

Example: Emulating Serial Port

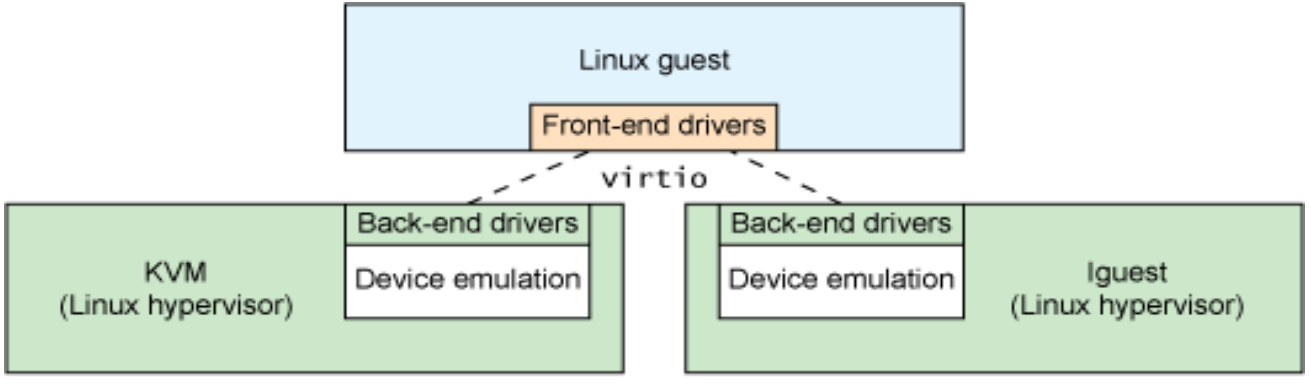


Emulated Devices

- Positives
  - Platform stability (good for migration)
  - Allows interposition
  - No special hardware support is needed
- Negatives
  - Can be slow (it’s software emulated)

Sol-3: Para-Virtualized Devices

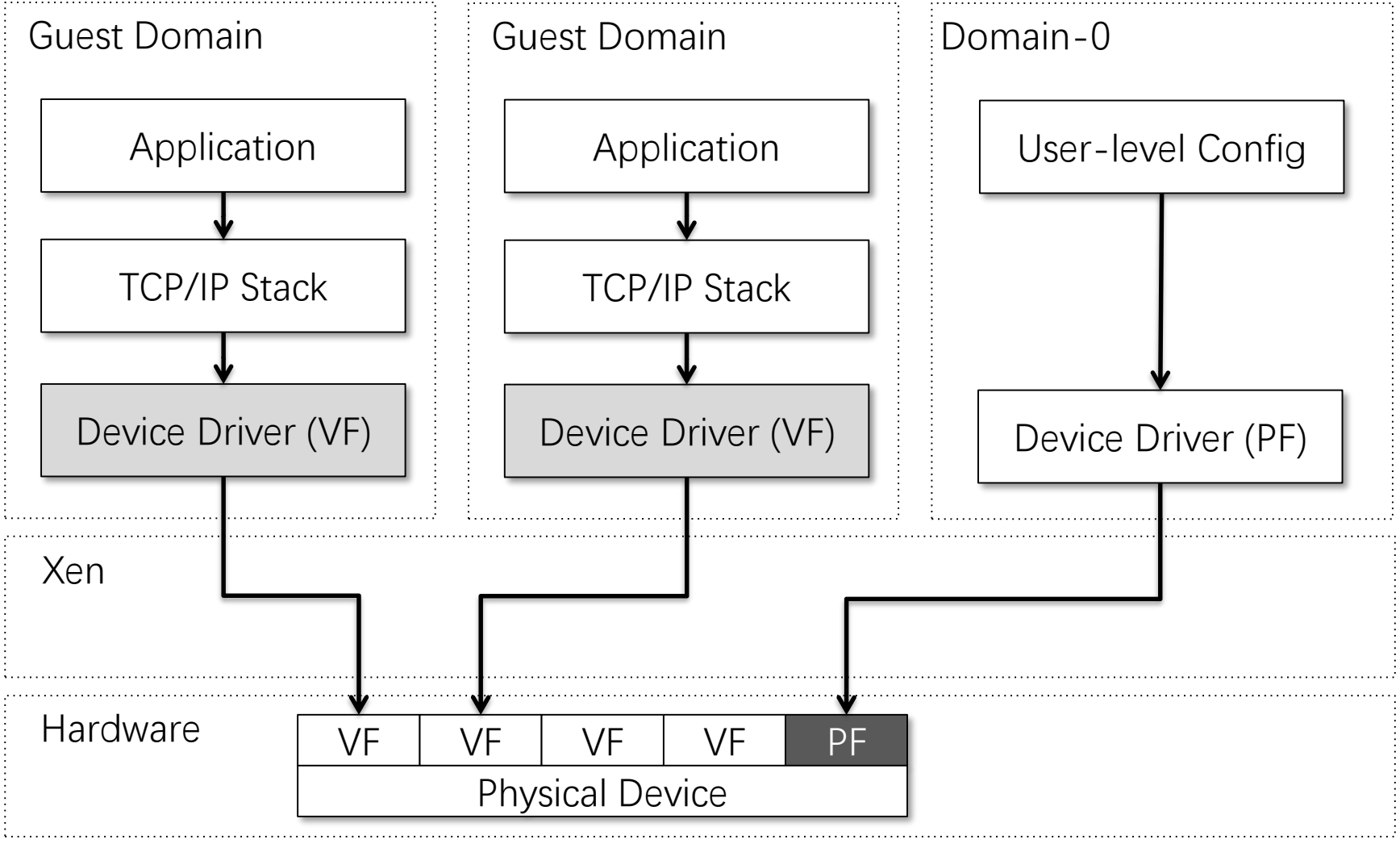
- VMM offers new types of device
  - The guest OS will run a new driver (front-end driver)
  - The VMM will run a back-end driver for each front-end
  - The VMM will finally run device driver to drive the device



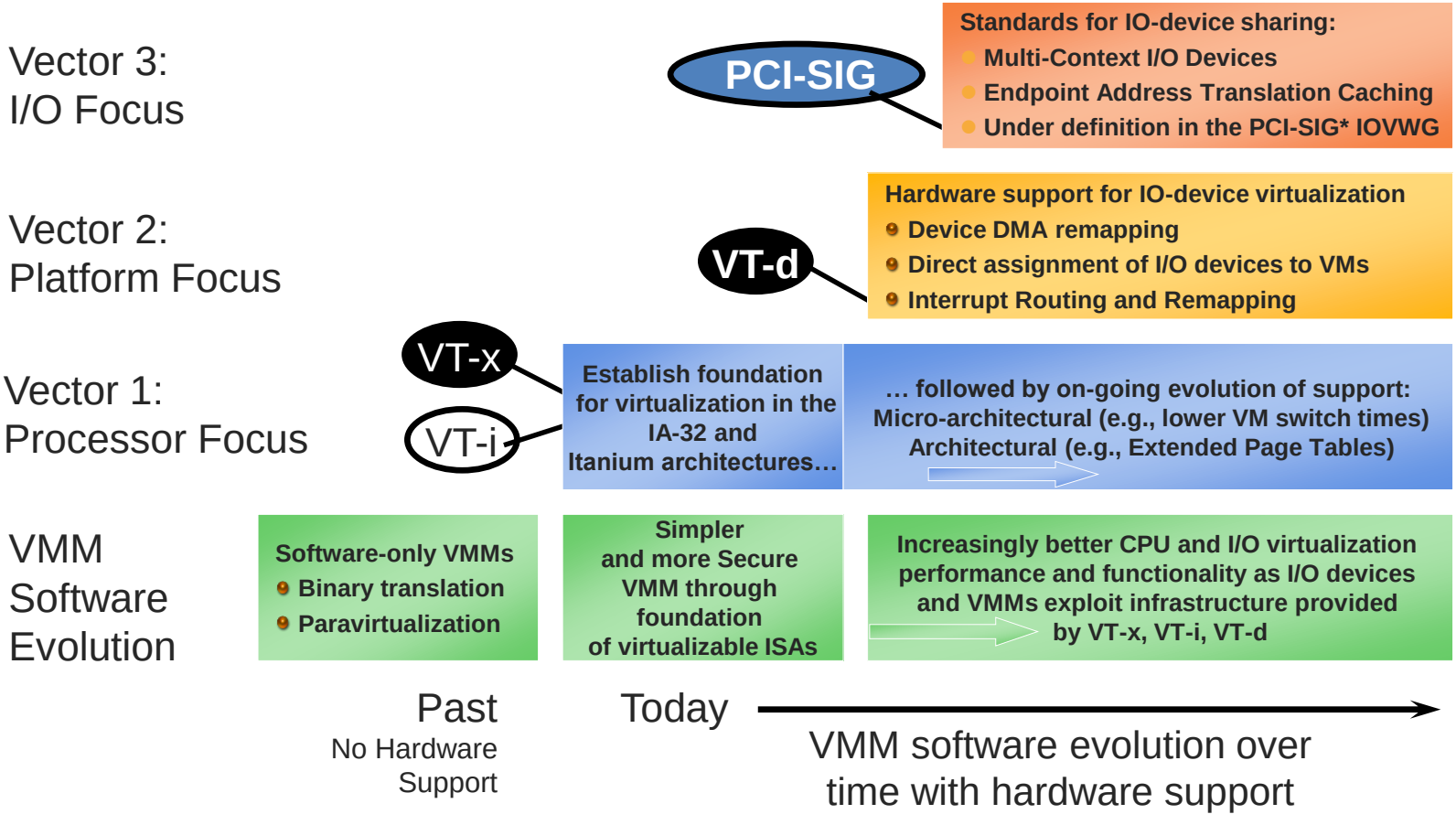
VirtIO Architecture by KVM

Sol-4: Hardware Support for I/O Virtualization

- VMM
  - An SR-IOV-capable device can be configured to appear in the PCI configuration space as multiple functions
- VM
  - The VMM assigns one or more VFs to a VM by mapping the actual configuration space of the VFs to the configuration space presented to the virtual machine by the VMM


Guest Domain
Application
TCP/IP Stack
Device Driver (VF)
Guest Domain
Application
TCP/IP Stack
Device Driver (VF)
Domain-0
User-level Config
Device Driver (PF)
Xen
Hardware
VF
VF
VF
VF
PF
Physical Device

Intel Virtualization Technology Evolution



Virtualization Technologies

Virtualization	Software Solution	Hardware Solution

CPU	• Trap & Emulate
	• Instruction interpretation
	• Binary translation
		• VT-x
		  • Root / non-root mode
		  • VMCS

	• Para-virtualization: Replace 17 insns	
Memory	• Shadow page table • Separating page tables for U/K	• EPT
	• Para-virtualization: Direct paging	
Device	• Direct I/O • Device emulation	• IOMMU • SR-IOV
	• Para-virtualization: Front-end & back-end driver (e.g., virtio)	