CSIE 5310

VM Performance & Nested Virtualization

Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

Administration

- Midterm exam will be held next Wednesday 10/23 @ 14:20-17:20
  - Location: R101
  - Format: Hand-written; ***Open book!***
    - You can have a double-sided A4 paper with you for the exam
    - You **cannot** talk to others either online/in person; those who violate the rules will be seen as cheating, and you will get an F and be reported!
  - Questions: *conceptual questions* covering all materials in the past courses to the end of today

Agenda

- **VM Performance**
- Nested Virtualization

VM Performance

- VMs provide better security guarantees and resource utilization
  - Can workloads run in the VM as fast as on bare-metal machines?
  - Fidelity is the goal, but performance is critical for VM adoption
- We have thus covered optimizations from the software and hardware
  - Goal: reduce hypervisor intervention to improve performance
    - General strategy: reduce VM exits to the hypervisor

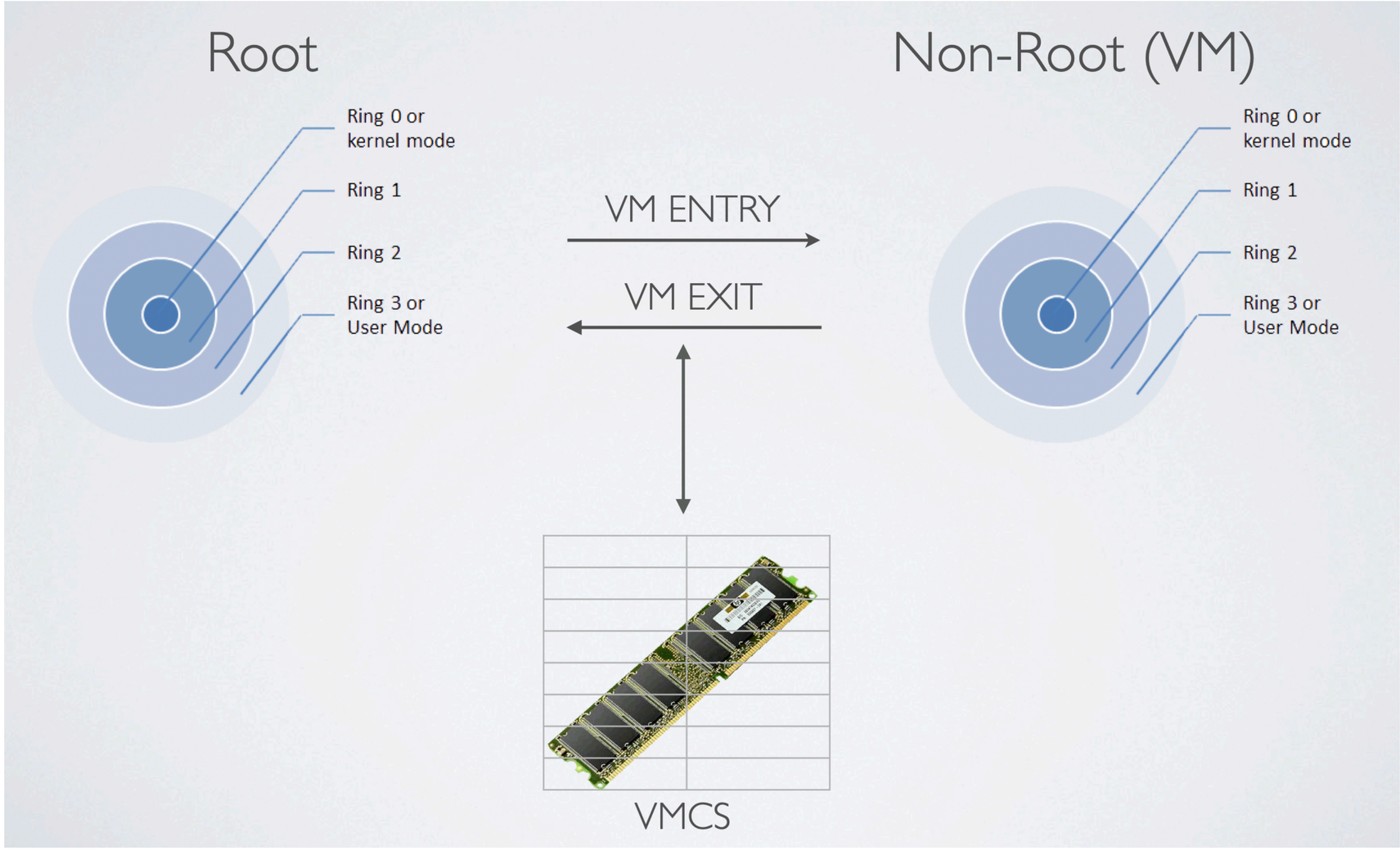
We have discussed

- CPU Virtualization
  - Optimizing virtualization for sensitive instructions
- Memory Virtualization
  - Optimizing shadow paging with nested paging
- I/O Virtualization (two goals)
  - Optimizing the cost of VM configuring and sending/receiving I/O data
  - Optimizing the cost of virtual interrupt delivery/handling

Review: Cost of CPU Virtualization

- Lack of hardware virtualization support; execute VMs in user mode to trap-and-emulate sensitive instructions
  - Cause a huge performance overhead
- Hardware support (Intel VT, Arm VE) allows most of the sensitive instructions to execute natively

Optimizing CPU Virtualization - Intel VT-x



From http://www.cs.columbia.edu/~cdall/pubs/KVMARM_talk.pdf

Optimizing Memory Virtualization

- Shadow page tables (SPT) vs Nested page tables (NPT)

	GPT Synchronization	Software Implementation Complexity	Cost of TLB miss
SPT	Needed	High	Low
NPT	Not Needed	Low	High

I/O Virtualization: Optimizing the cost of VM configuring and sending/receiving I/O data

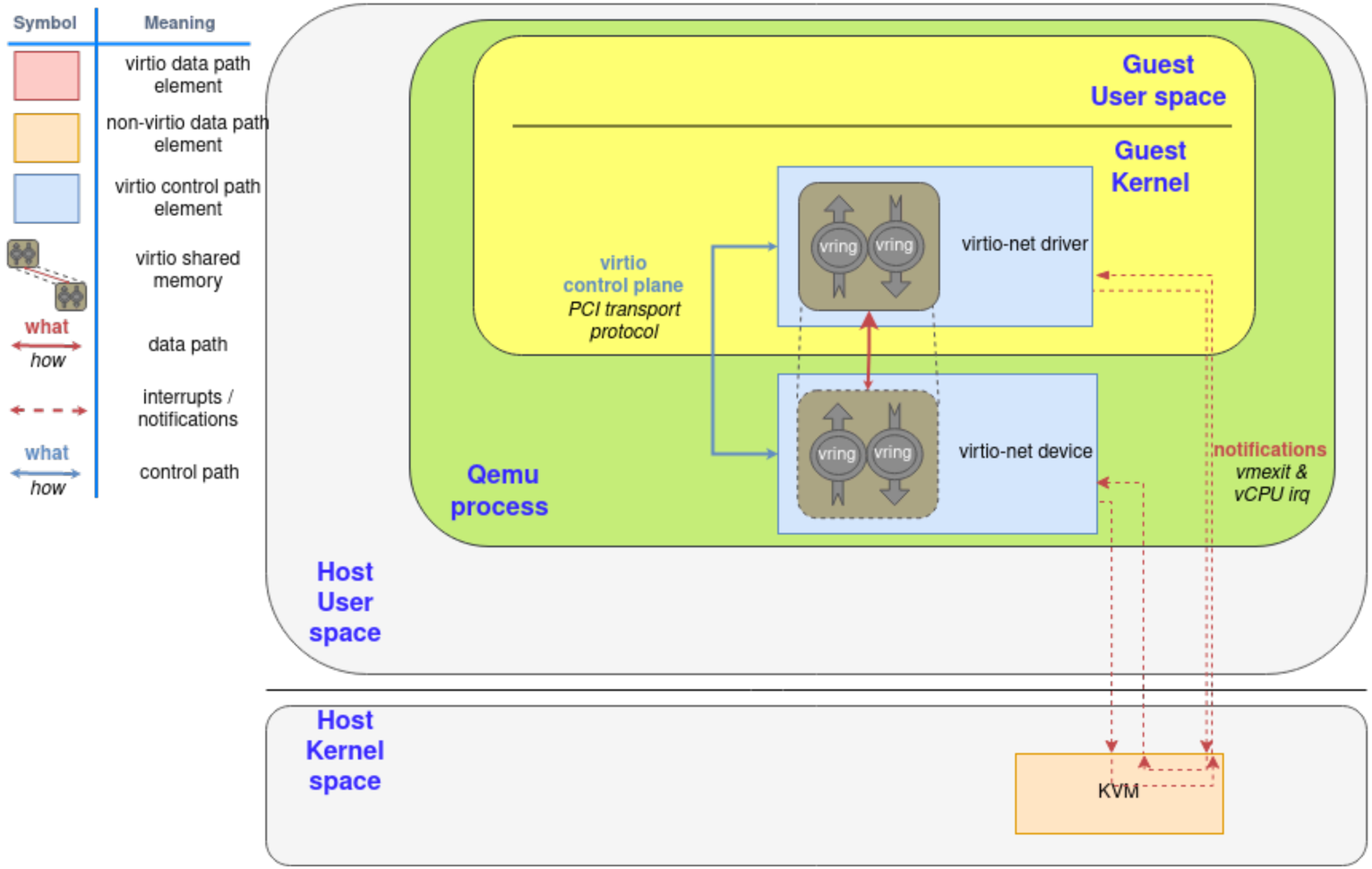
- I/O emulation is straightforward to implement but slow
  - Why? Lots of VM exits
- Reduce VM exits caused by I/O emulation
  - Use I/O paravirtualization or device passthrough

TCP Stream Performance: I/O emulation vs I/O paravirtualization

- Experimental setup:
  - Run Netperf TCP stream test within a Linux VM on Linux/KVM
  - Test VMs using fully emulated e1000 and virtio-net NIC
  - Test against the server running on the Linux/KVM host
  - Measure the throughput (Mb/s) of TCP sends from a VM

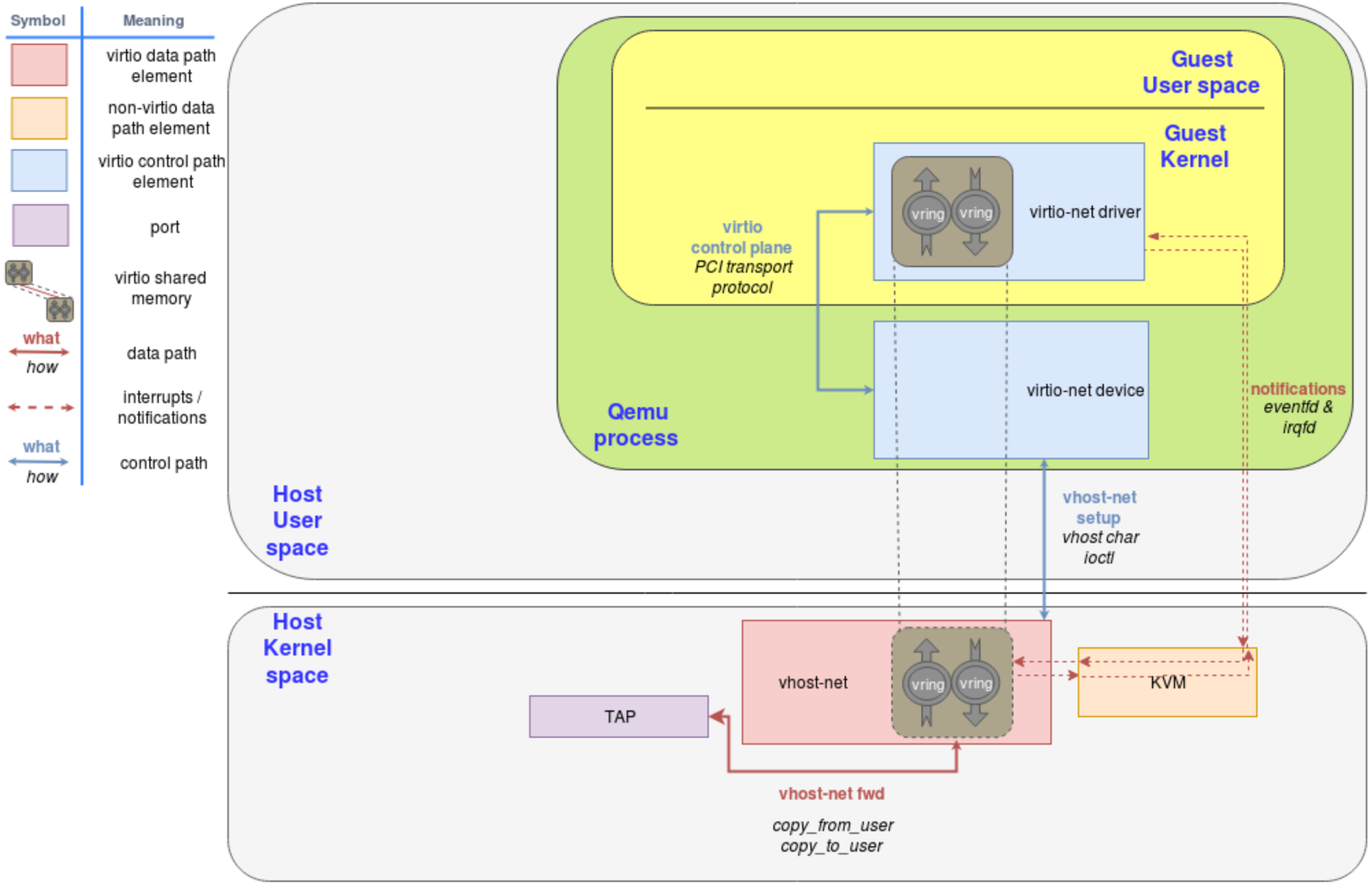
	Metric	e1000	Virtio-net	Ratio
Guest	throughput (Mbps)	239	5,230	22x
	exits per second	33,783	1,126	1/30x
TCP segments	per exit	1/9	25	225x
	per interrupt	1	118	118x
	per second	3,669	30,252	8x
	avg. size (bytes)	8,168	21,611	3x
	avg. processing time (cycles)	652,443	79,132	1/8x

I/O Paravirtualization: virtio



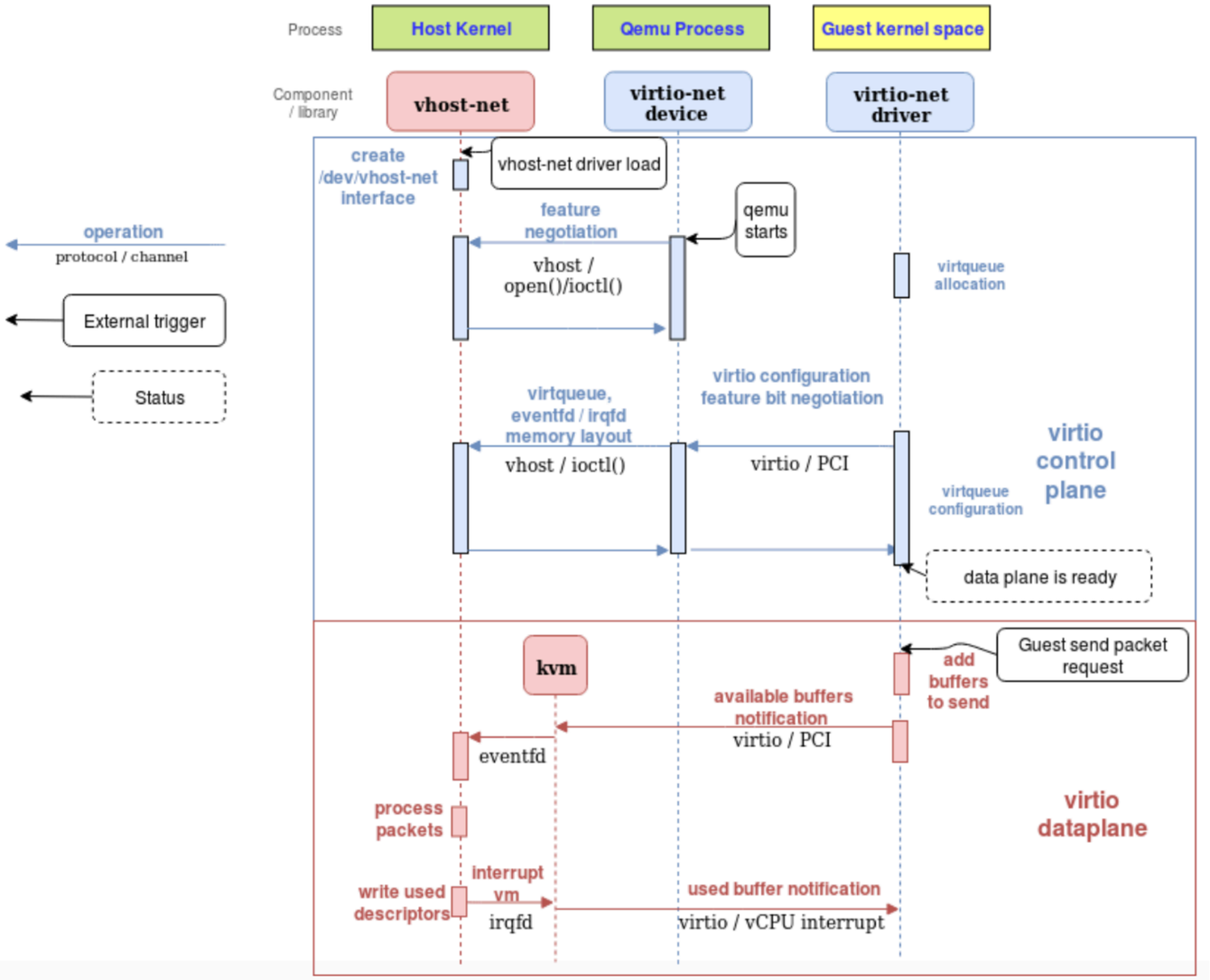
From: https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone

I/O Paravirtualization: virtio vhost net (1)



From: https://www.redhat.com/rhdc/managed-files/2019-09-12-virtio-networking-fig3.png

I/O Paravirtualization: virtio vhost net (2)



From: https://www.redhat.com/rhdc/managed-files/2019-09-12-virtio-networking-fig4.png

Device Passthrough

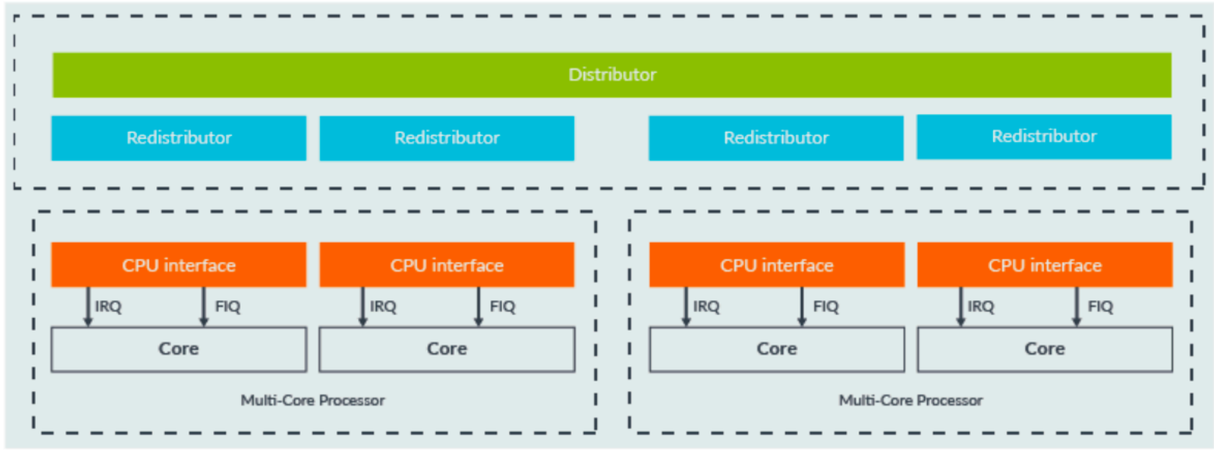
- Allow VMs to manage the physical I/O device directly with no hypervisor intervention
- Hardware support for safe and scalable device passthrough support
  - IOMMU and SRIOV
- Cost in virtual interrupt delivery
  - Passthrough devices cannot send interrupts directly to the VM without special hardware/ software support
  - The hypervisor handles physical interrupts from the hardware and injects virtual interrupts to the VM

I/O Virtualization: Optimizing the cost of virtual interrupt delivery/handling

- Allows VMs to handle hardware interrupts without VM exits
  - Ex: Intel VT-d's hardware support for posted interrupts
- Allow VMs to manage the interrupt controller without VM exits; ex: EOI interrupts without trapping
  - Requires hardware support from the interrupt controller

Review: Arm Interrupt Controller

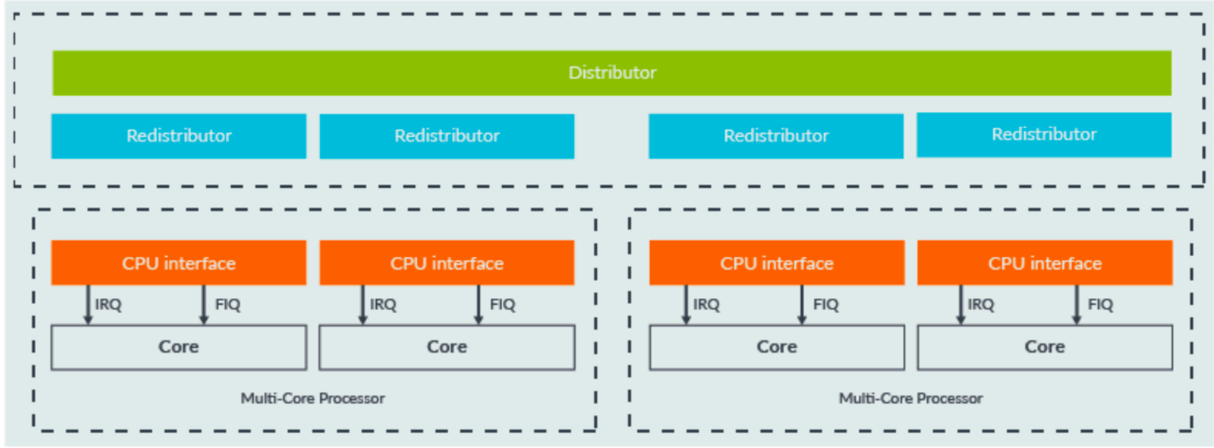
- Case Study: GICv3 consists of three groups of registers
  - Distributor interface
  - Redistributor interface
  - CPU interface
- Program access to GICv3 registers:
  - Access CPU interface via MSR/MRS instructions
  - Access Distributor and Redistribution via MMIO instruction



arm_generic_interrupt_controller_v3_and_v4_-_virtualization_guide_107627_0101_01_en.pdf

Arm Interrupt Controller: Virtualization Support (1)

- GICv3 provides virtualization support:
  - Hardware virtualization of the CPU interface -> VM accesses cause no trap
  - Allows hypervisors to program the GIC to inject virtual interrupts
- The GIC does not provide features for virtualizing the Distributor and Redistributor
  - KVM trap-and-emulate VM accesses to GIC Distributors and Redistributes

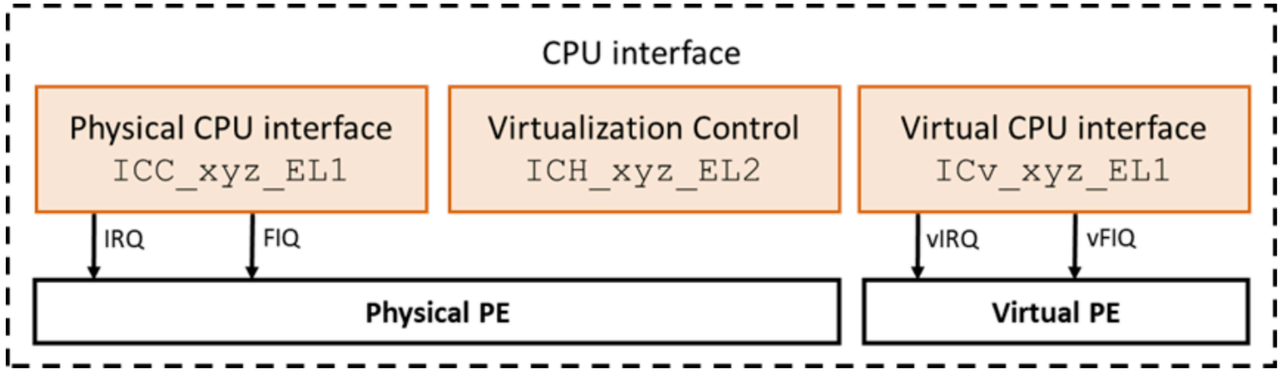


arm_generic_interrupt_controller_v3_and_v4_-_virtualization_guide_107627_0101_01_en.pdf

Arm Interrupt Controller: Virtualization Support (2)

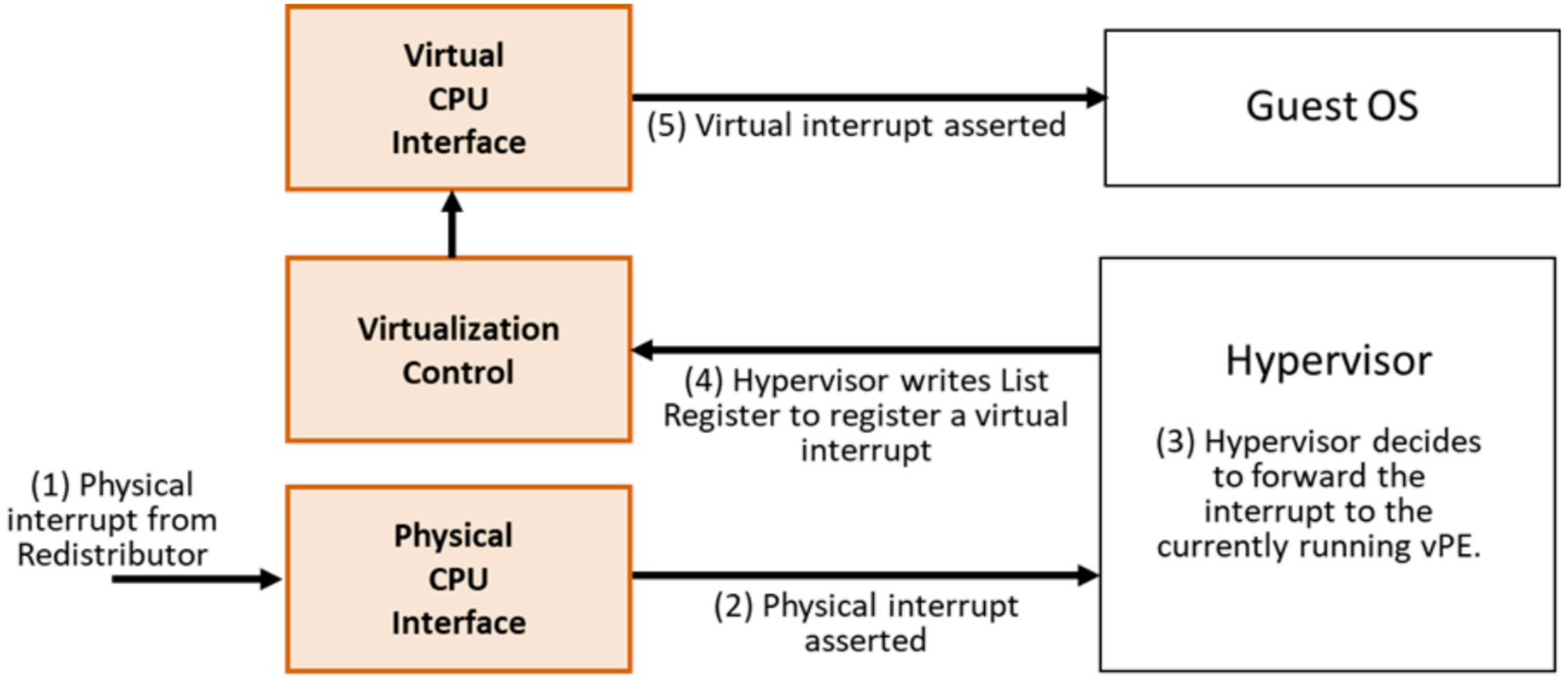
- GICv3 CPU interface registers are split into 3 groups:
  - *Physical CPU*: used by the hypervisor to handle physical interrupts
  - *Virtual CPU*: used by the software executing in a virtualized environment to handle virtual interrupts
    - Registers in this group have the same format as the ones in the Physical CPU interface
  - *Virtualization Control*: used by the hypervisor to control the virtualization features

Figure 3-1: CPU interface registers with virtualization



arm_generic_interrupt_controller_v3_and_v4_-_virtualization_guide_107627_0101_01_en.pdf

Arm Interrupt Controller: Interrupt Handling



arm_generic_interrupt_controller_v3_and_v4_-_virtualization_guide_107627_0101_01_en.pdf

Agenda

- **VM Performance**
  - **Microbenchmarks & Application Benchmarks**
- Nested Virtualization

VM Trap Cost

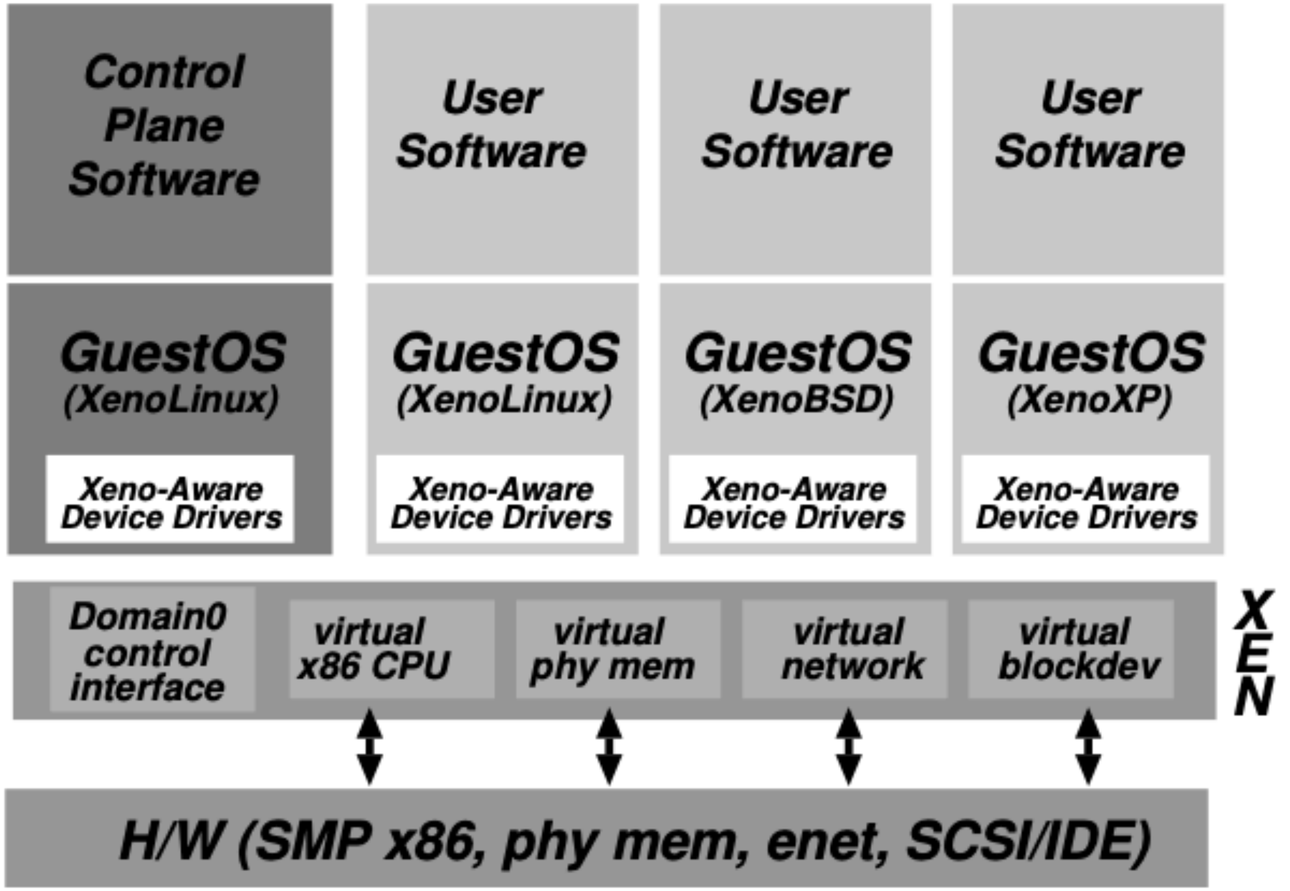
- Advanced hardware features are not always available
  - IOMMU and SRIOV may be missing in some hardware platforms
  - Virtio is widely used by VMs in the cloud such as Amazon AWS
- VM exits/traps cannot be entirely eliminated; knowing the trap cost of micro-level operations during virtualization is important

Microbenchmarks

- “ARM Virtualization: Performance and Architectural Implications” [1] presents a measurement study of Arm and x86 hypervisors
  - Used the state-of-art software/hardware: KVM and Xen using hardware virtualization features on Armv8 (VE) and x86 (Intel VT-x)
- Built microbenchmarks (kernel drivers) and modified the hypervisor to measure the cost for the hypervisor to handle VM exits

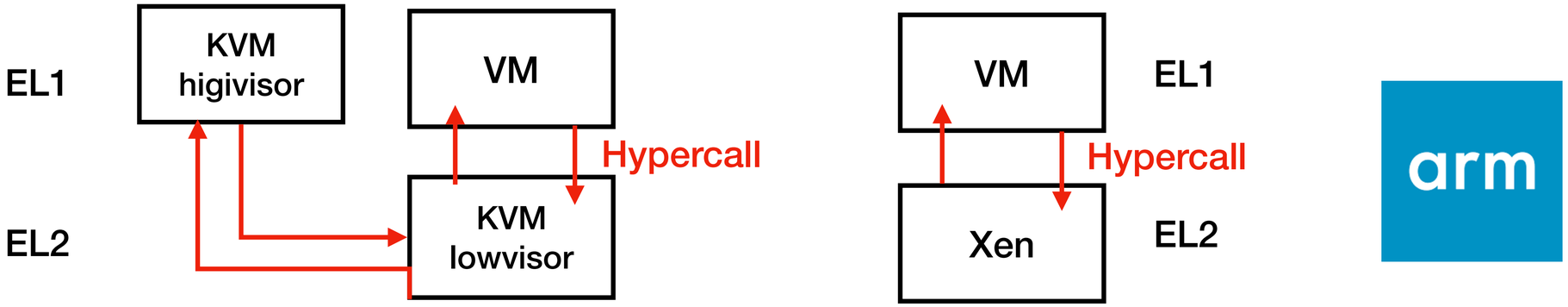
[1] ARM Virtualization: Performance and Architectural Implications Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos, ISCA 16

Xen's Architecture



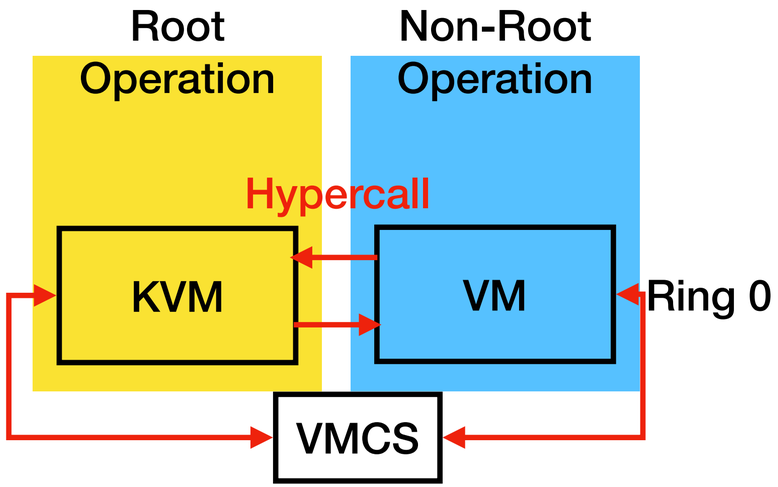
Microbenchmarks: Null hypercalls (1)

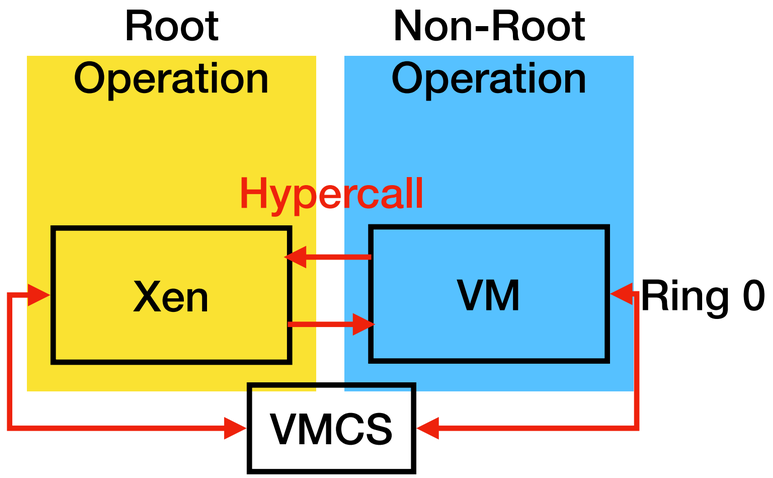
- First measure the basic transition cost of VM <—> hypervisor
  - Measure a the CPU cycles for KVM and Xen to handle a NULL hypercall



Microbenchmarks: Null hypercalls (2)

- First measure the basic transition cost of VM <—> hypervisor
  - Measure a the CPU cycles for KVM and Xen to handle a NULL hypercall





Microbenchmarks: Null hypercalls (3)

Microbenchmark	ARM		x86	
	KVM	Xen	KVM	Xen
Hypercall	6,500	376	1,300	1,228

Microbenchmarks: Null hypercalls (4)

Microbenchmark	ARM		x86	
	KVM	Xen	KVM	Xen
Hypercall	6,500	376	1,300	1,228

Why KVM is so slow?

Why similar?

Why x86 is much higher?

Microbenchmarks: Null hypercalls (5)

- Why is KVM Arm much slower than Xen Arm?
  - Xen handles hypercalls in EL2
    - What is/are the register state listed in the table needed to be context switched in Xen?
  - KVM handles hypercalls in EL1; resulting in high context switching cost

Register State	Save	Restore
GP Regs	152	184
FP Regs	282	310
EL1 System Regs	230	511
VGIC Regs	3,250	181
Timer Regs	104	106
EL2 Config Regs	92	107
EL2 Virtual Memory Regs	92	107

Breakdown of Hypercall cost in KVM/Arm

Microbenchmarks: Null hypercalls (6)

- Why is x86 slower than Arm?
  - x86 hardware context switches CPU states to/from VMCS
  - Arm hardware does not context switch CPU states — software decides what states to save or store
- Why is KVM similar to Xen on x86?
  - Hypercalls are handled by KVM and Xen in ring 0 in root operation

Microbenchmarks: other operations (1)

Microbenchmark	ARM		x86	
	KVM	Xen	KVM	Xen
Hypercall	6,500	376	1,300	1,228
Interrupt Controller Trap	7,370	1,356	2,384	1,734
Virtual IPI	11,557	5,978	5,230	5,562
Virtual IRQ Completion	71	71	1,556	1,464

KVM emulates virtual interrupt controller in highvisor in EL1; Xen does it in EL2 -> extra context switch overhead in KVM

Microbenchmarks: other operations (2)

Microbenchmark	ARM		x86	
	KVM	Xen	KVM	Xen
Hypercall	6,500	376	1,300	1,228
Interrupt Controller Trap	7,370	1,356	2,384	1,734
Virtual IPI	11,557	5,978	5,230	5,562
Virtual IRQ Completion	71	71	1,556	1,464

Arm allows VM to EOI interrupt without trapping to VMM

Application Benchmark Performance (1)

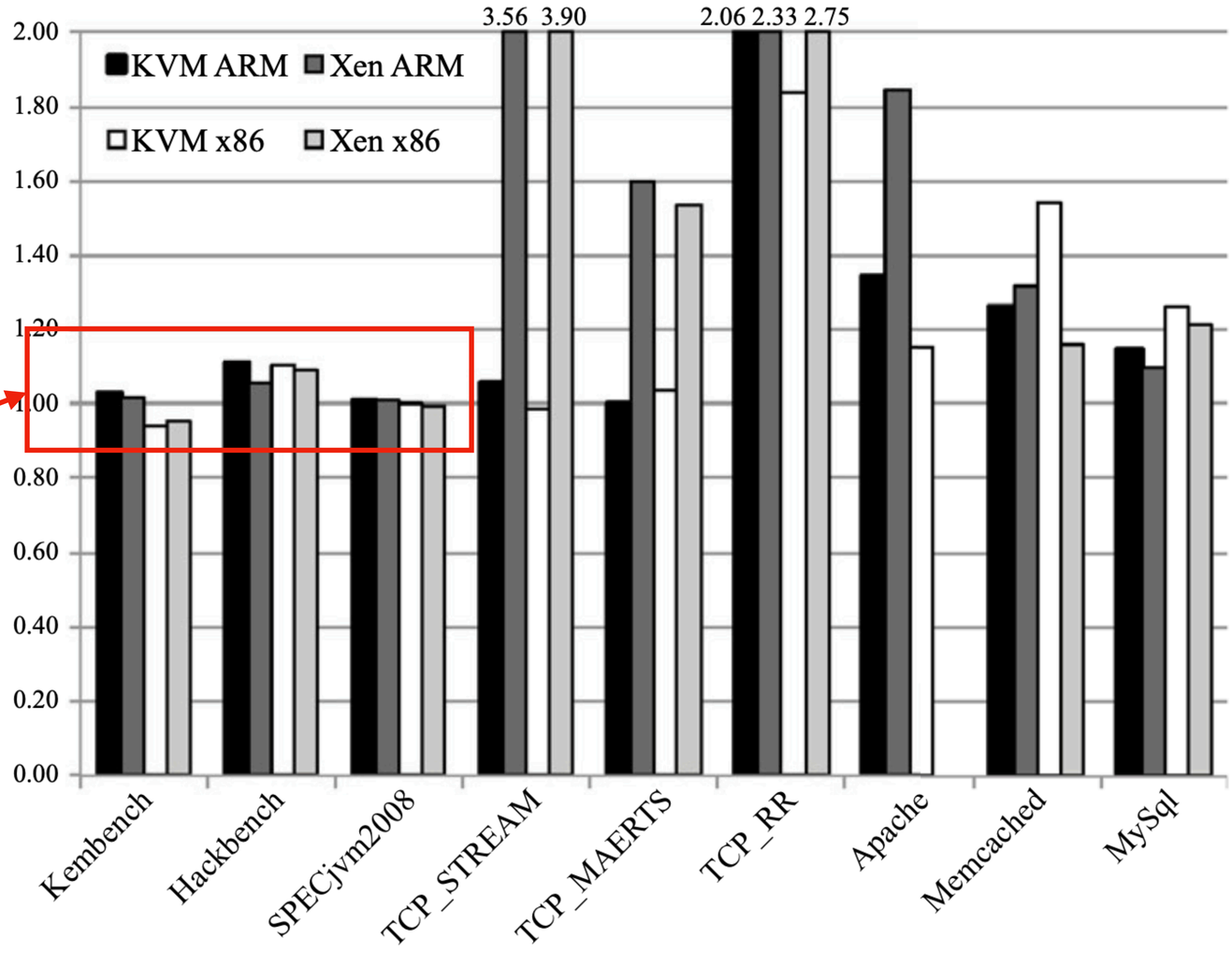
- Measure VM performance on KVM and Xen
  - Drive network I/O workloads from a client running on a remote machine
- KVM is configured with virtio devices: virtio-block and virtio-vhost-net
- Xen is configured with standard Xen PV I/O (paravirtual I/O)

Application Benchmark Performance (2)

Benchmark	Description
Kernbench	Kernel compilation by compiling the Linux 3.17.0 kernel using the allnoconfig for ARM using GCC 4.8.2.
Hackbench	`hackbench` [132] using unix domain sockets and 100 process groups running with 500 loops.
SPECjvm2008	`SPECjvm2008` [160] 2008 benchmark running several real life applications and benchmarks specifically chosen to benchmark the performance of the Java Runtime Environment. 15.02 release of the Linaro AArch64 port of OpenJDK was used run the benchmark.
Netperf	`netperf` v2.6.0 starting netserver on the server and running with its default parameters on the client in three modes: TCP_STREAM, TCP_MAERTS, and TCP_RR, measuring throughput transferring data from client to server, throughput transferring data from server to client, and latency, respectively.
Apache	`Apache` v2.4.7 Web server running `ApacheBench` v2.3 on the remote/local client, which measures the number of handled requests per second serving the index file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	`memcached` v1.4.14 using the `memtier` benchmark v1.2.3 with its default parameters.
MySql	`MySQL` v14.14 (distrib 5.5.41) running the `SysBench` v.0.4.12 OLTP benchmark using the default configuration with 200 parallel transactions.

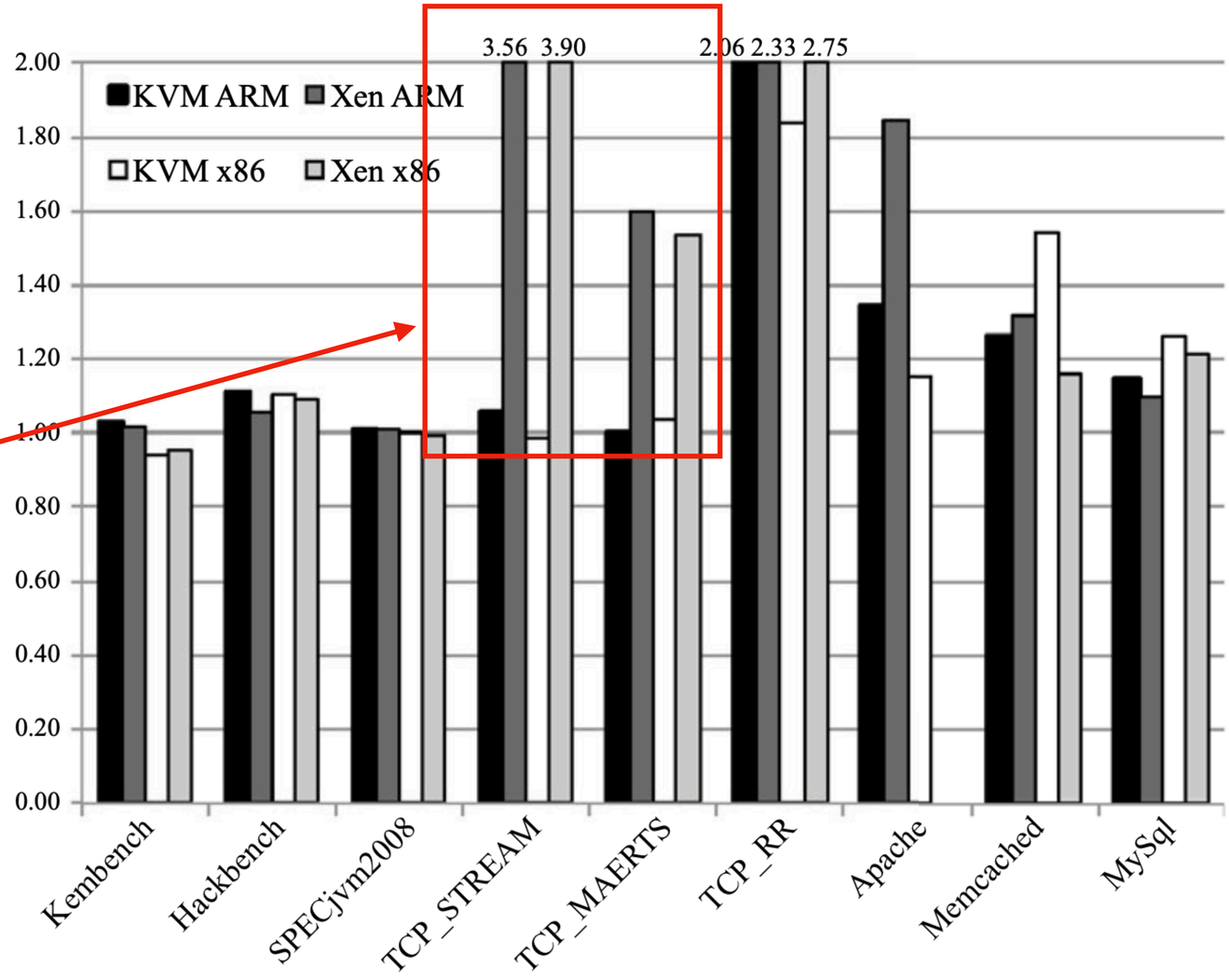
CPU bounded workloads

Small overhead overall. Workloads are CPU bound; not many VM traps

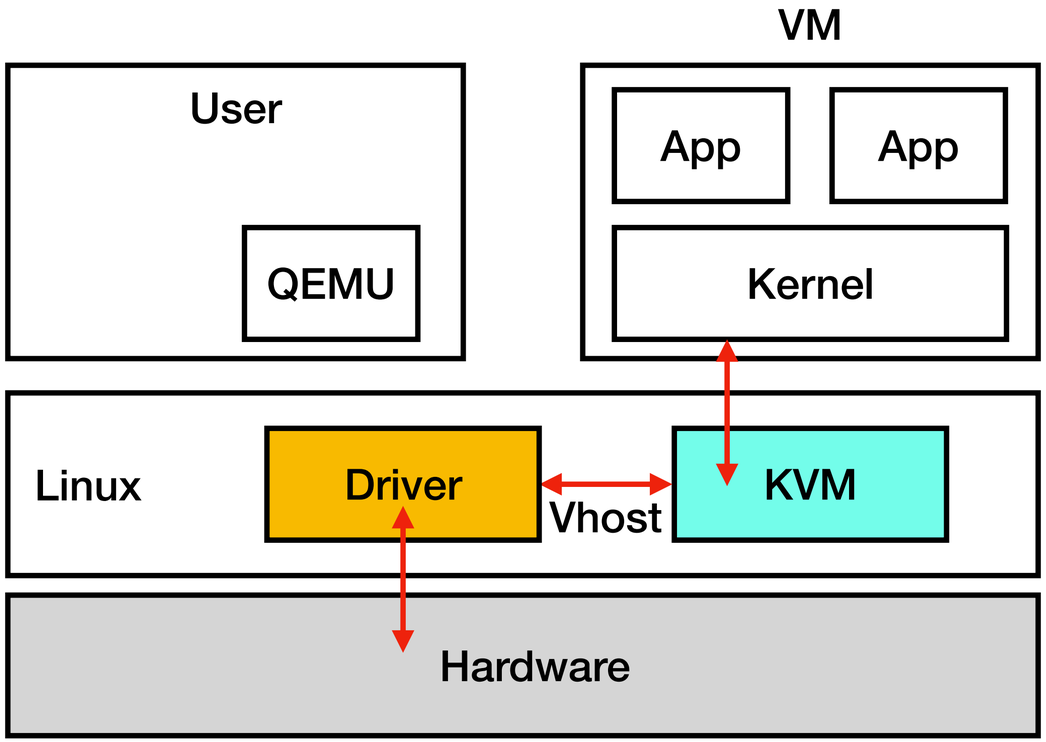


Network bandwidth - TCP_STREAM/MAERTS

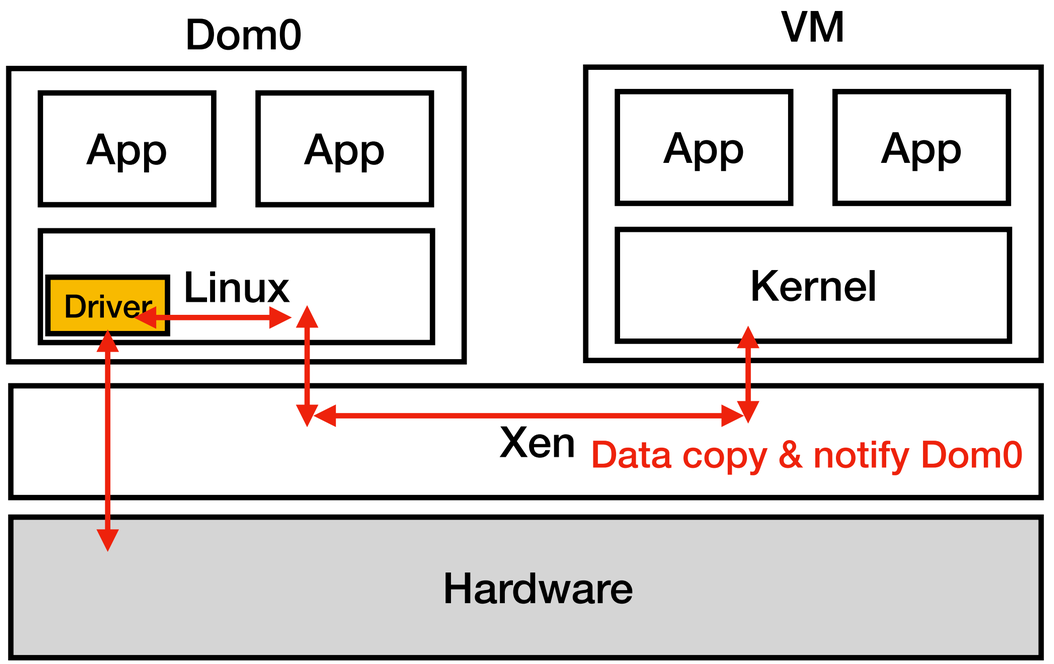
Xen has significant overhead over KVM



KVM I/O Model (w virtio/vhost)



Xen PV I/O Model

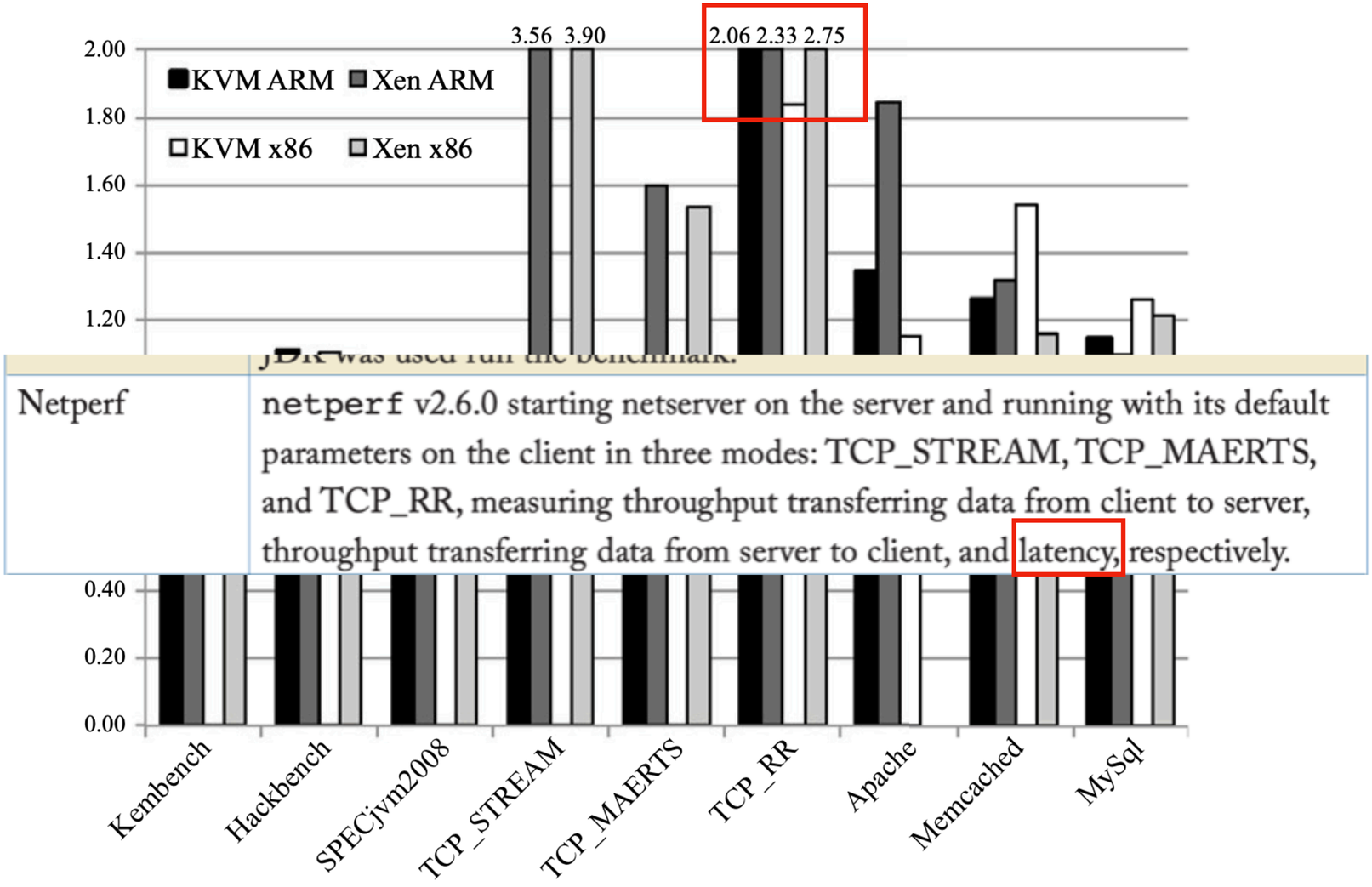


Dom0 by default cannot access VM memory

Networking: KVM virtio/vhost vs Xen PV I/O

	KVM	Xen
Trap (null hypercall)	Slow	Fast
I/O handling	By Linux in KVM	By Linux in Dom0

Network Latency - TCP_RR (1)

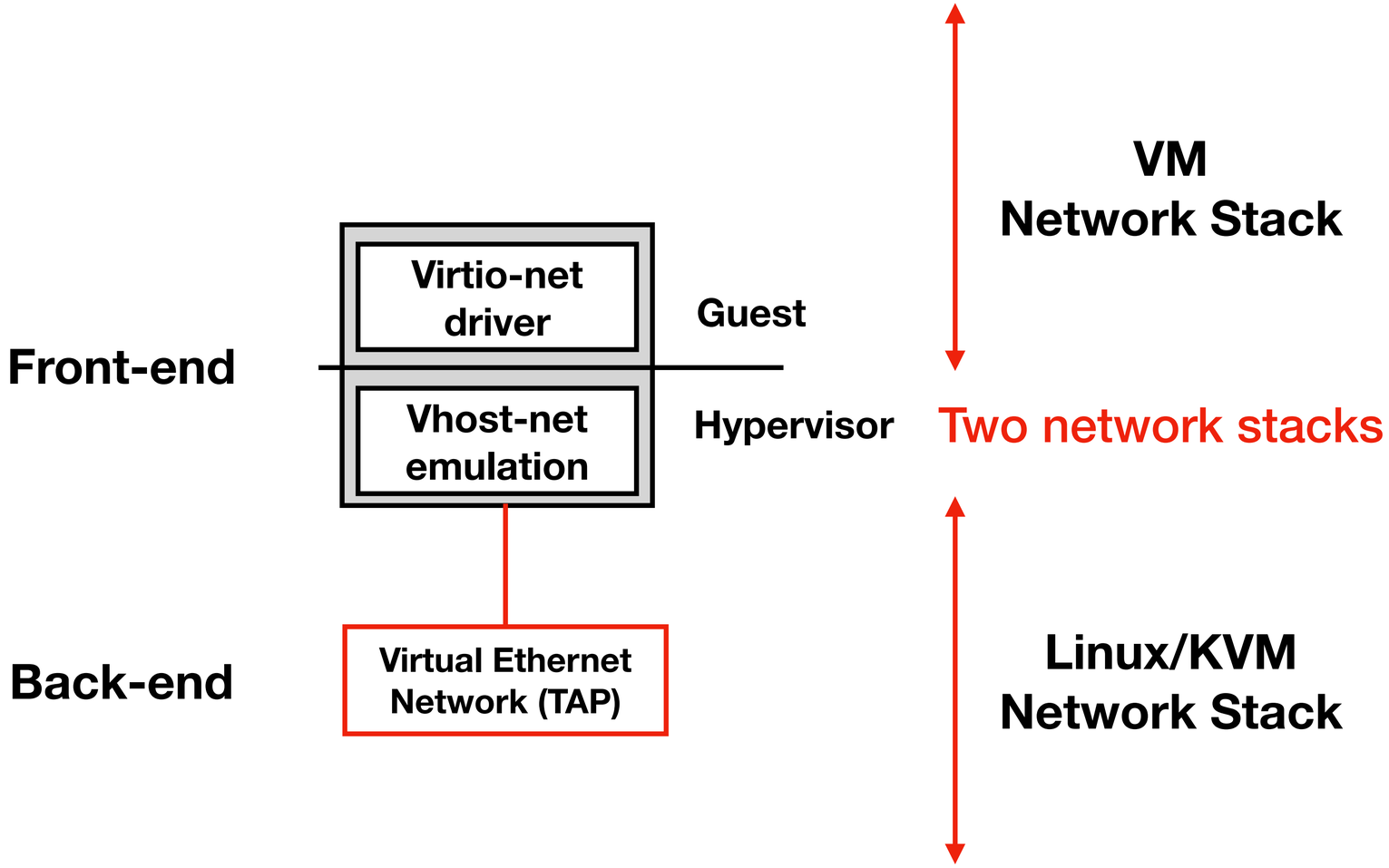


Netperf	`netperf` v2.6.0 starting netserver on the server and running with its default parameters on the client in three modes: TCP_STREAM, TCP_MAERTS, and TCP_RR, measuring throughput transferring data from client to server, throughput transferring data from server to client, and latency, respectively.

Network Latency - TCP_RR (2)

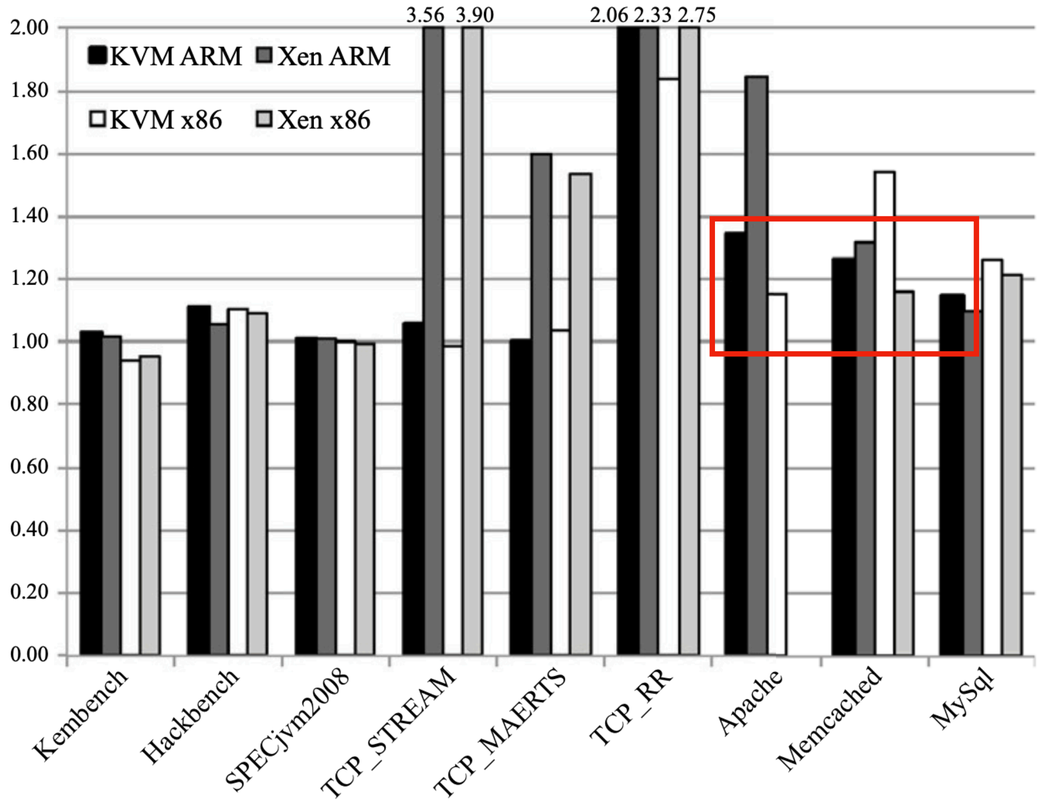
- TCP_RR measures the time it takes to:
  - Send 1 packet from the client to the server (in VM), and the server sends the packet back to the client
- Xen performs worse than KVM due to the same reason as earlier (rely on Dom0 to provide I/O virtualization)
- KVM also performs 2x worse compared to bare-metal; why?

Network Latency - TCP_RR (3)

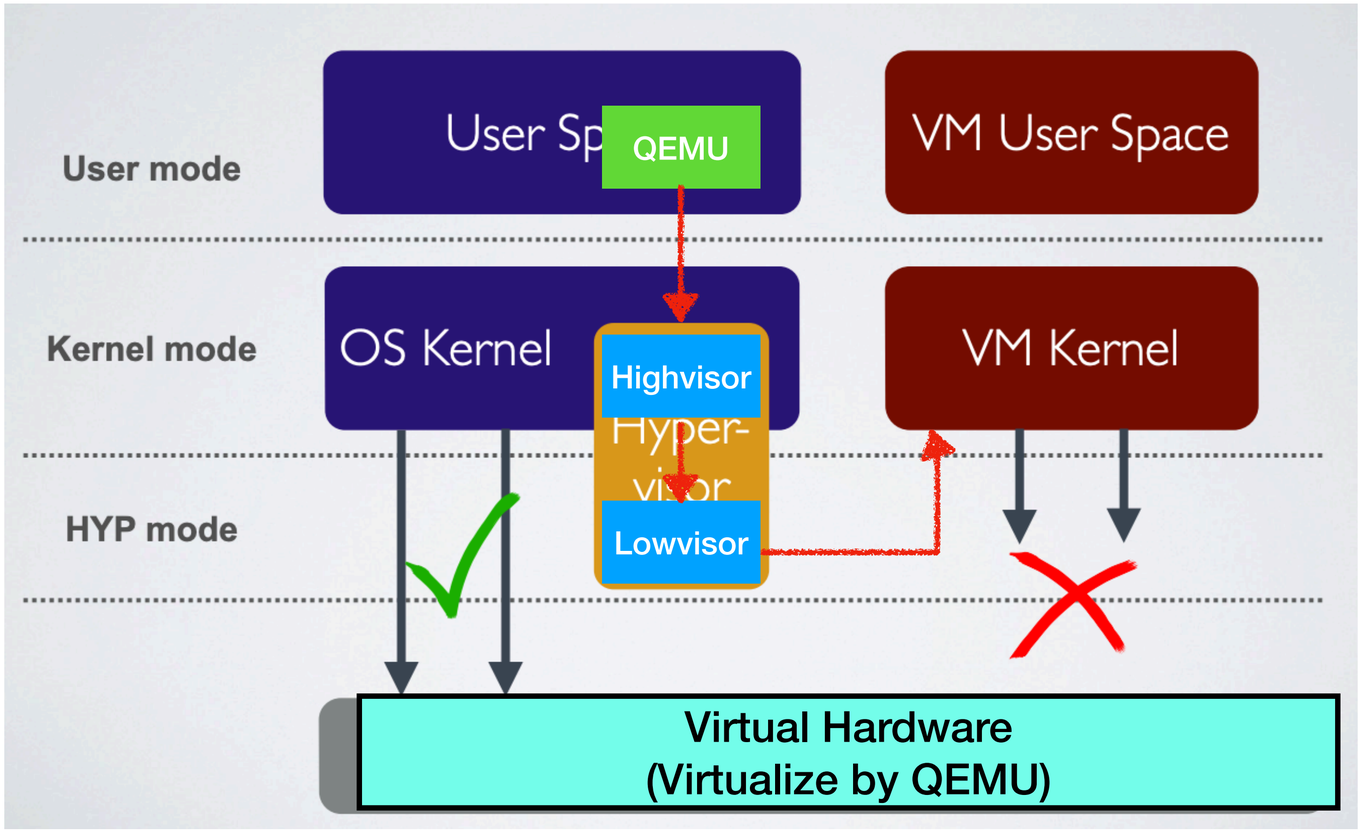


Overhead in Server Network Applications

- Overhead in handling interrupts and delivering virtual interrupts
- KVM and Xen cannot scale interrupt handling
  - Virtual interrupts both were injected into a single vCPU
  - Cannot split workloads across vCPUs to fully utilize all CPU resources

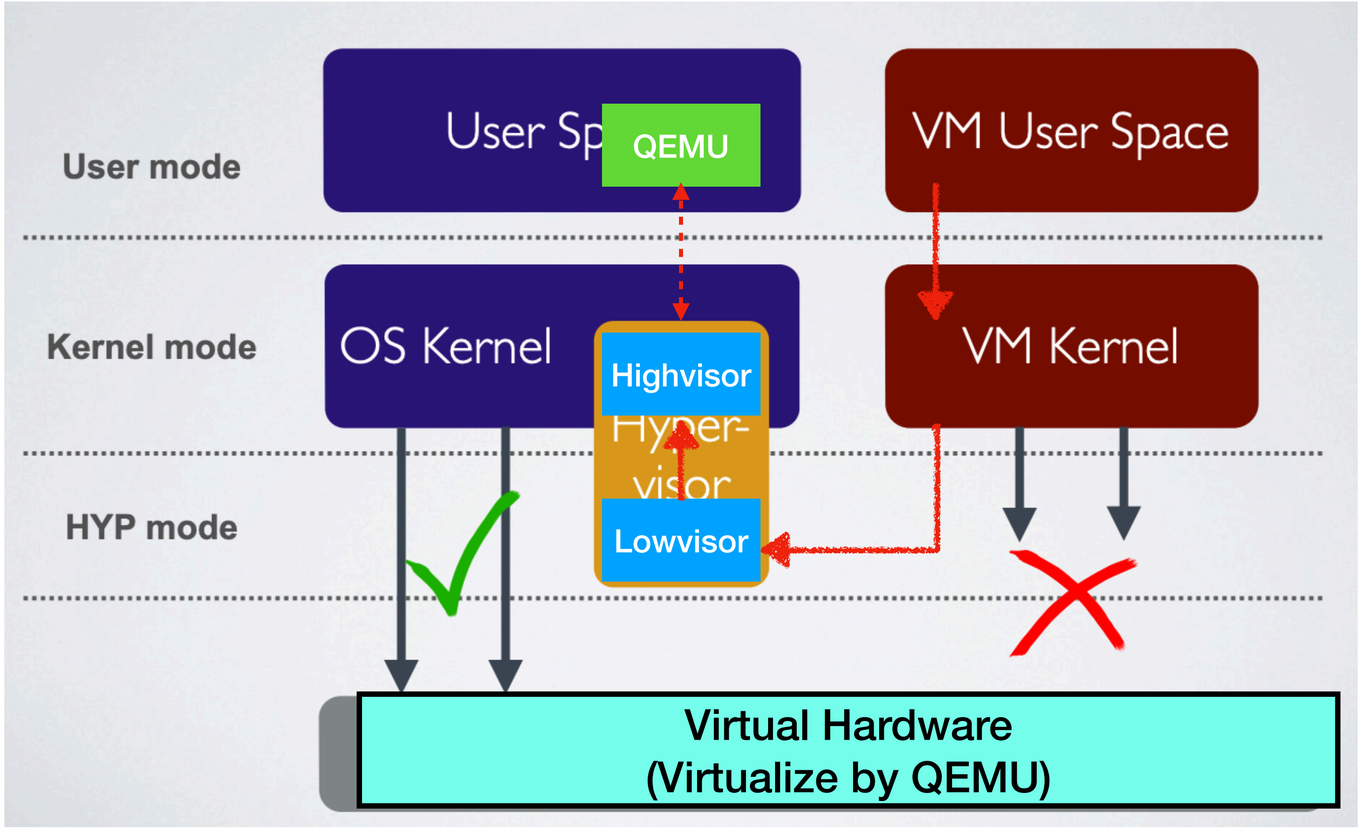


Review: KVM for Arm - VM Enter



Modified from http://www.cs.columbia.edu/~cdall/pubs/KVMARM_talk.pdf

Review: KVM for Arm - VM Exit



Modified from http://www.cs.columbia.edu/~cdall/pubs/KVMARM_talk.pdf

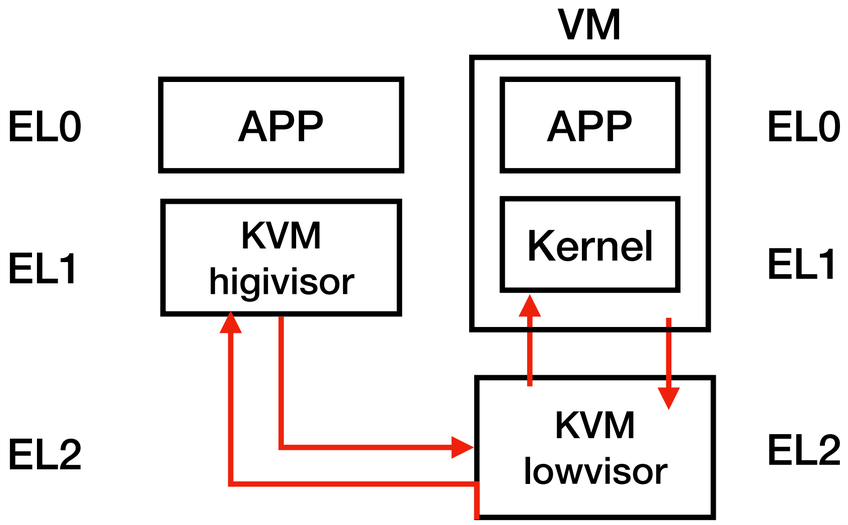
Architectural Updates for Arm VE (1)

- KVM on Arm initially adopts split-mode virtualization
  - Linux cannot run directly in EL2
  - Results in a high cost in context switches between EL1 and EL2
- Arm 8.1 introduces *Virtualization Host Extensions (VHE)*
  - Allows Linux to run entirely in EL2 without modification

Architectural Updates for Arm VE (2)

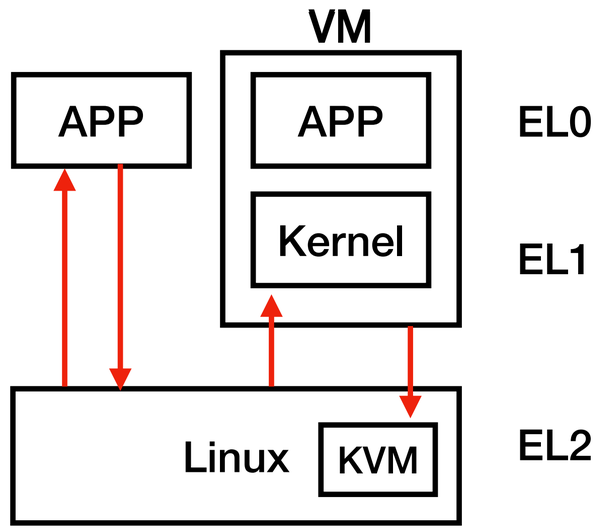
- Updates in Arm VHE:
  - Extra set of EL2 registers
    - Arm VE provides one TTBR in EL2, but Linux uses two TTBRs, why?
  - Hardware register access redirection of EL1 to EL2 registers
    - The existing kernel that uses EL1 registers can run directly in EL2
  - Make EL2 page table formats compatible with EL1's

KVM with/without VHE support



KVM without VHE

arm



KVM with VHE

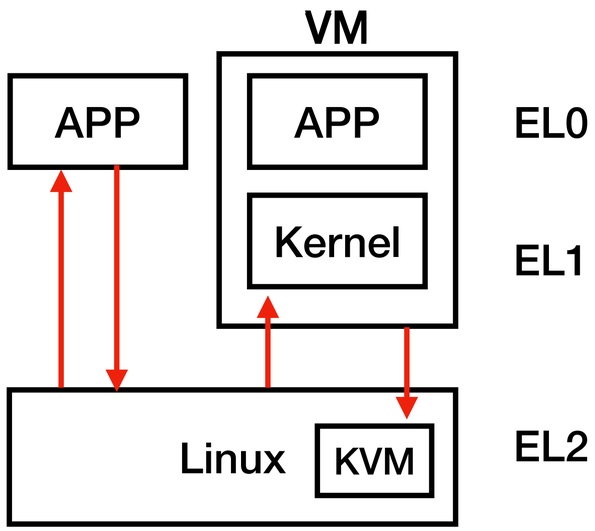
Running KVM in EL2 entirely

- Previous work "*Optimizing the Design and Implementation of the Linux ARM Hypervisor*" (Usenix ATC 2017) tried to manually port Linux/KVM to run entirely in EL2 on an Arm v8 hardware, which VHE is unavailable!

Why only slightly better than Arm?

Microbenchmark	ARM	ARM EL2	ARM EL2 OPT
Hypercall	6,413	6,277	752
I/O Kernel	8,034	7,908	1,604
I/O User	10,012	10,186	7,630
Virtual IPI	13,121	12,562	2,526

Running KVM in EL2 entirely



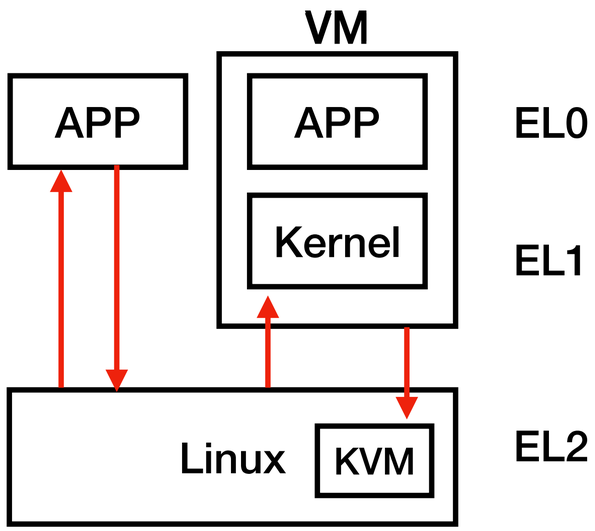
KVM with VHE

arm

Register State	Save	Restore
GP Regs	152	184
FP Regs	282	310
EL1 System Regs	230	511
VGIC Regs	3,250	181
Timer Regs	104	106
EL2 Config Regs	92	107
EL2 Virtual Memory Regs	92	107

The cost here can mostly be removed; why?

Running KVM in EL2 entirely



KVM with VHE

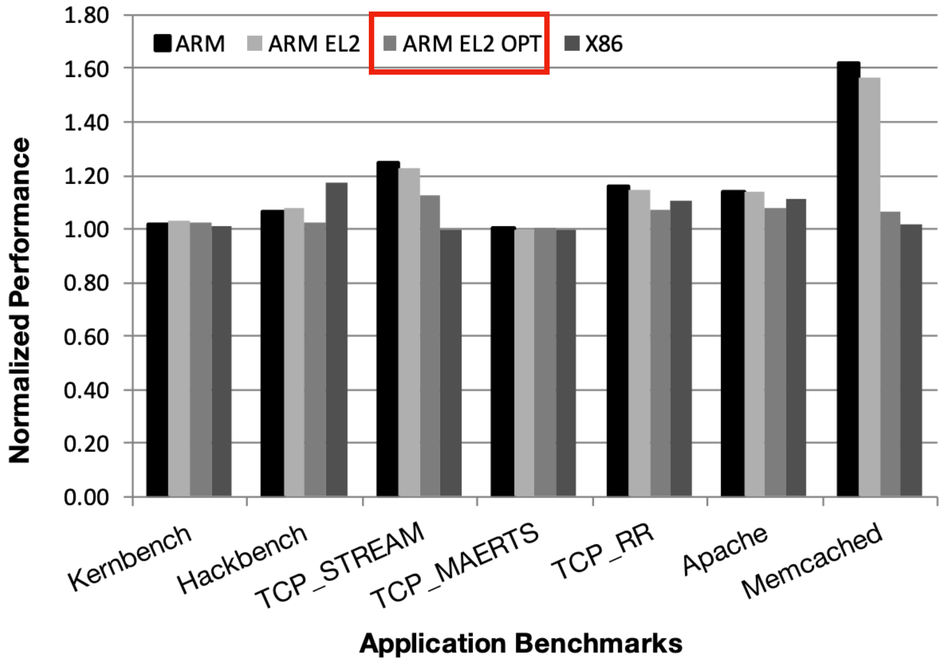
arm

Register State	Save	Restore
GP Regs	152	184
FP Regs	282	310
EL1 System Regs	230	511
VGIC Regs	3,250	181
Timer Regs	104	106
EL2 Config Regs	92	107
EL2 Virtual Memory Regs	92	107

Recall that KVM did context switch because EL1 is shared between the VM and KVM highvisor

Running KVM in EL2 entirely

- Optimizing EL2 switches leads to significant improvements in applications
  - [ATC17] measured performance with device passthrough



Summary: Virtualization Performance

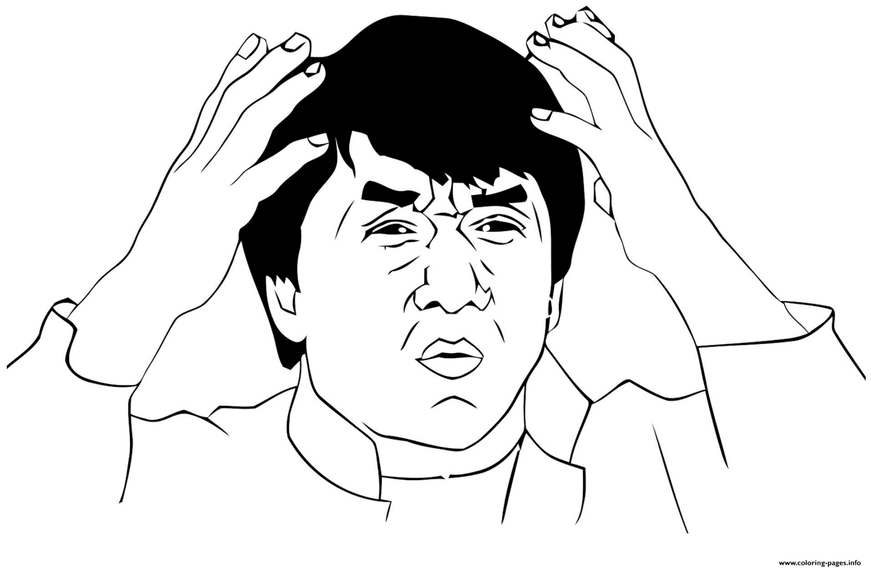
- Software and hardware approaches have been introduced to improve virtual machine performance; two strategies:
  - Reducing traps
  - Reducing the overhead involved in each trap
- More challenges ahead:
  - Support for more devices; what about GPUs?
  - More advanced hardware features; offloading virtio to hardware?

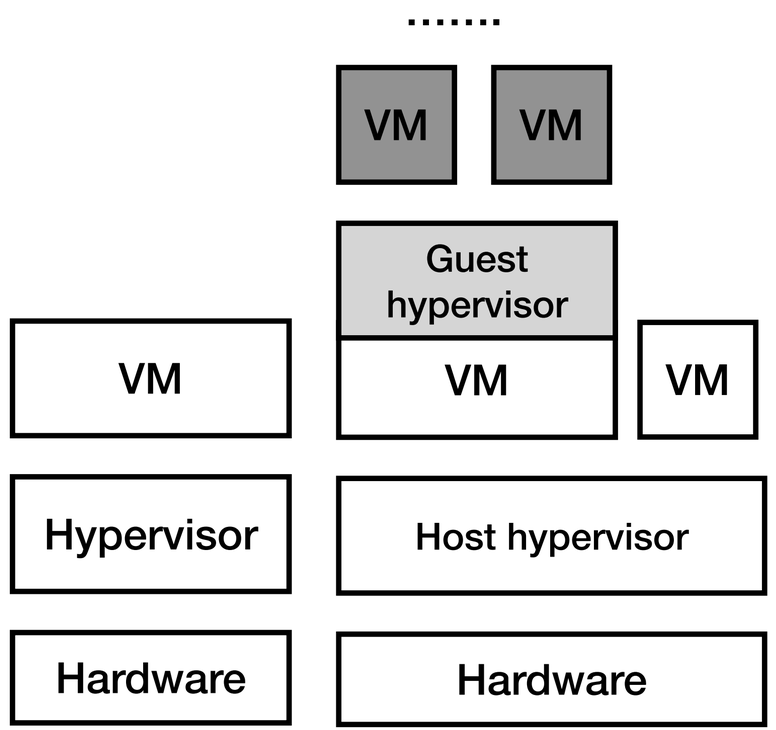
Agenda

- VM Performance
  - Microbenchmarks & Application Benchmarks
- **Nested Virtualization**

Nested Virtualization

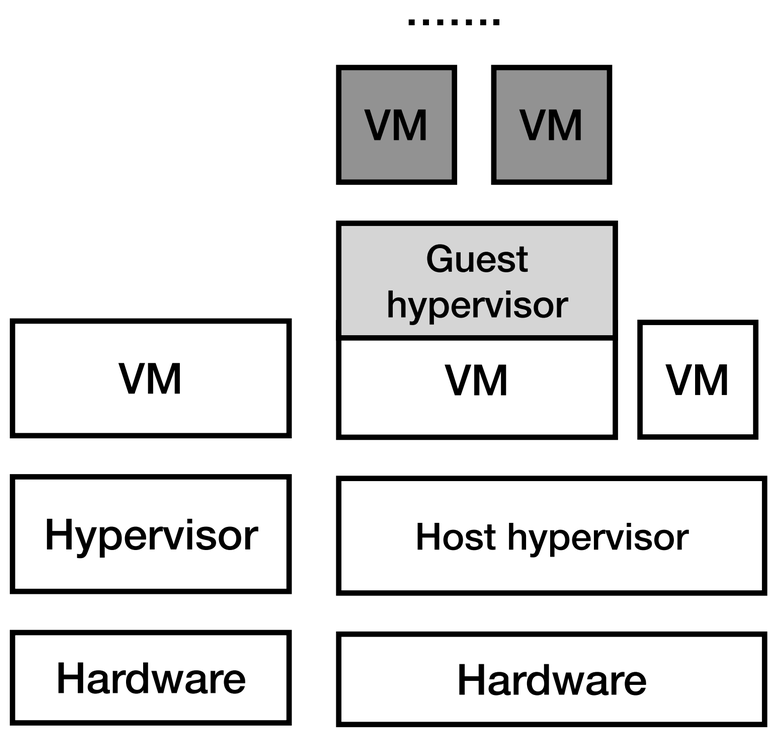
- Run hypervisors on top of a hypervisor with good performance





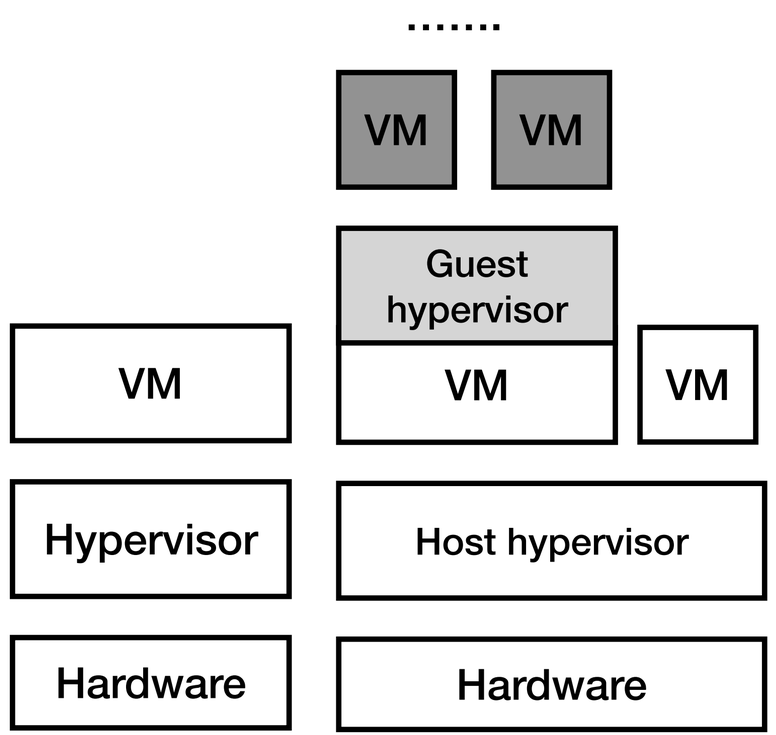
Nested Virtualization: Motivation

- Run your custom hypervisors in the public cloud setting
- Security (e.g. hypervisor-level rootkit)
- Simplifying hypervisor development



Nested Virtualization: Terminology

- Host hypervisor: the hypervisor that runs on bare-metal
- Guest hypervisor: the hypervisor that runs in VM(s)
- Nested VMs: VMs that run on a guest hypervisor



The Turtles Project (1)

- First attempt made by IBM Research Haifa
- Won the best paper award in OSDI 2010 [1]
- Efficient nested virtualization for Intel x86 environments based on KVM
  - Leverage Intel VT-x (VMX) but ***without*** the “hardware” nested virtualization support
- Supports multiple guest hypervisors and VMs
  - VMware, Windows, …

[1] The Turtles Project: Design and Implementation of Nested Virtualization, OSDI 10

The Turtles Project (2)

IBM

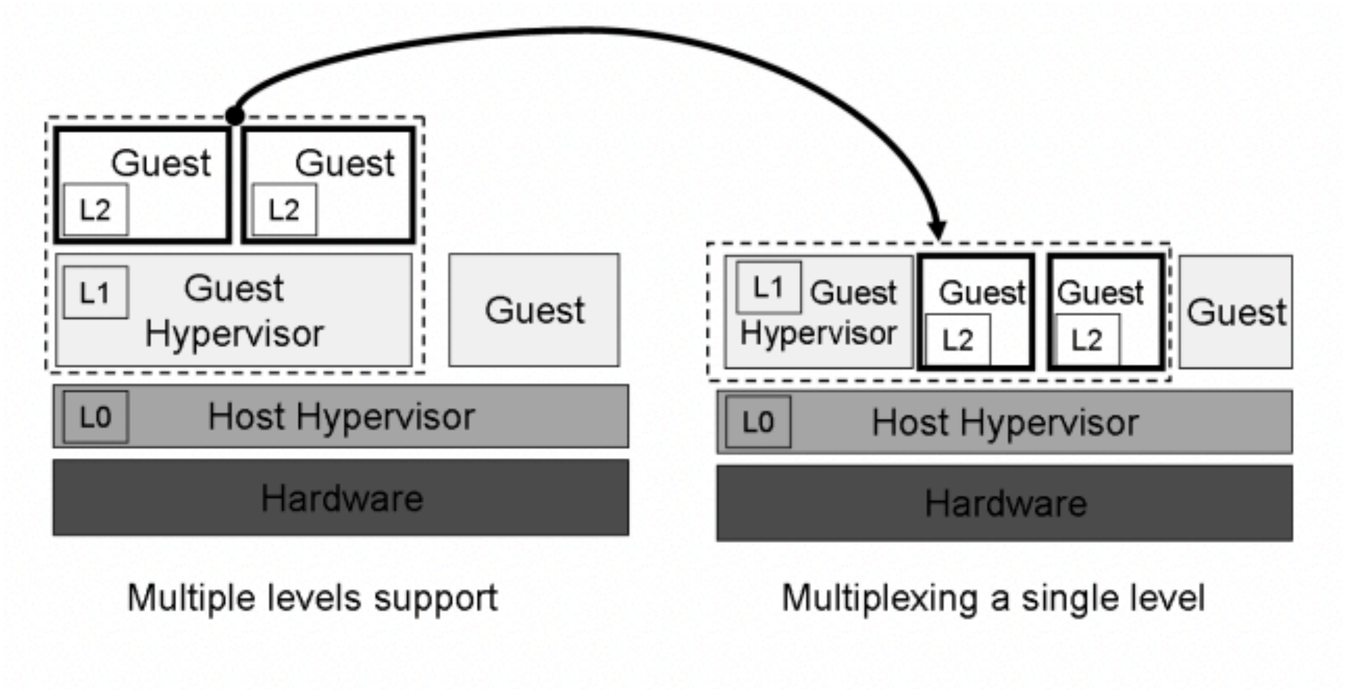
- Proposed different techniques to support:
  - Nested CPU virtualization
  - Nested memory virtualization
  - Nested I/O virtualization
- Nested virtualization without hardware support is slow:
  - The Turtles project proposed performance optimizations for nested VM

Key technique from the Turtles project

- Issue: cannot directly expose Intel VMX hardware to the guest hypervisors for good performance; how? Only one VMX hardware is available
  - The host hypervisor (that runs in ***root operation***) controls the hardware VMX and virtualizes VMX for the guest hypervisors
  - Each guest (that run in ***non-root operation***) hypervisor virtualizes VMX for the hypervisor above it

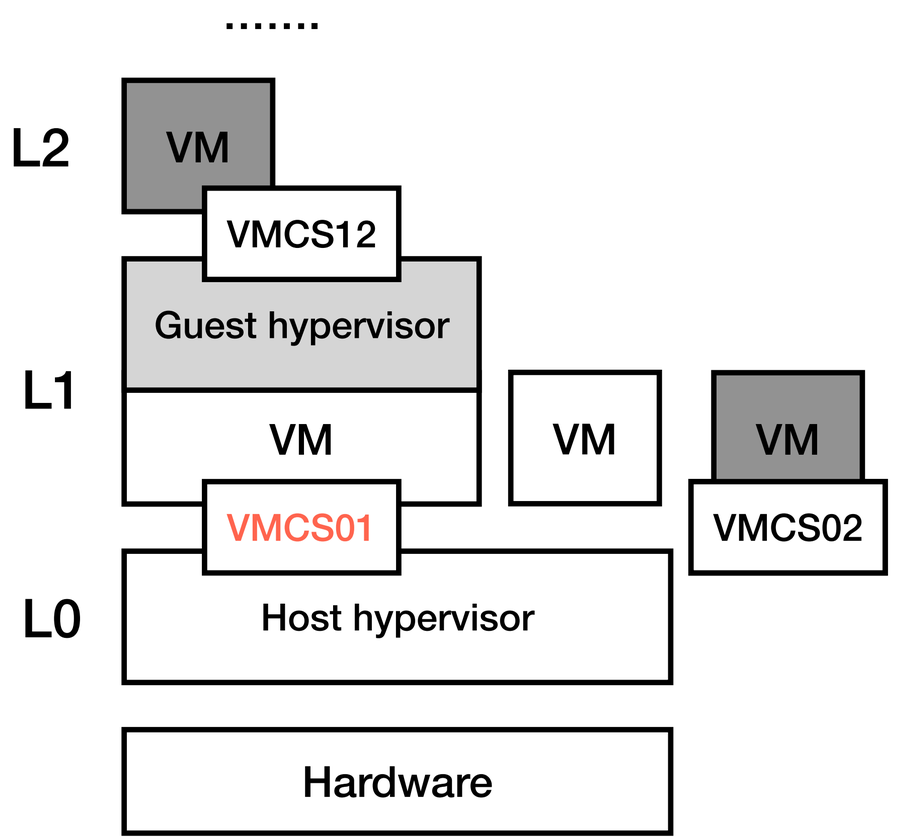
Key Technique from Turtles

- The host hypervisor (L0) multiplexes the hardware between L1, L2, …, Ln



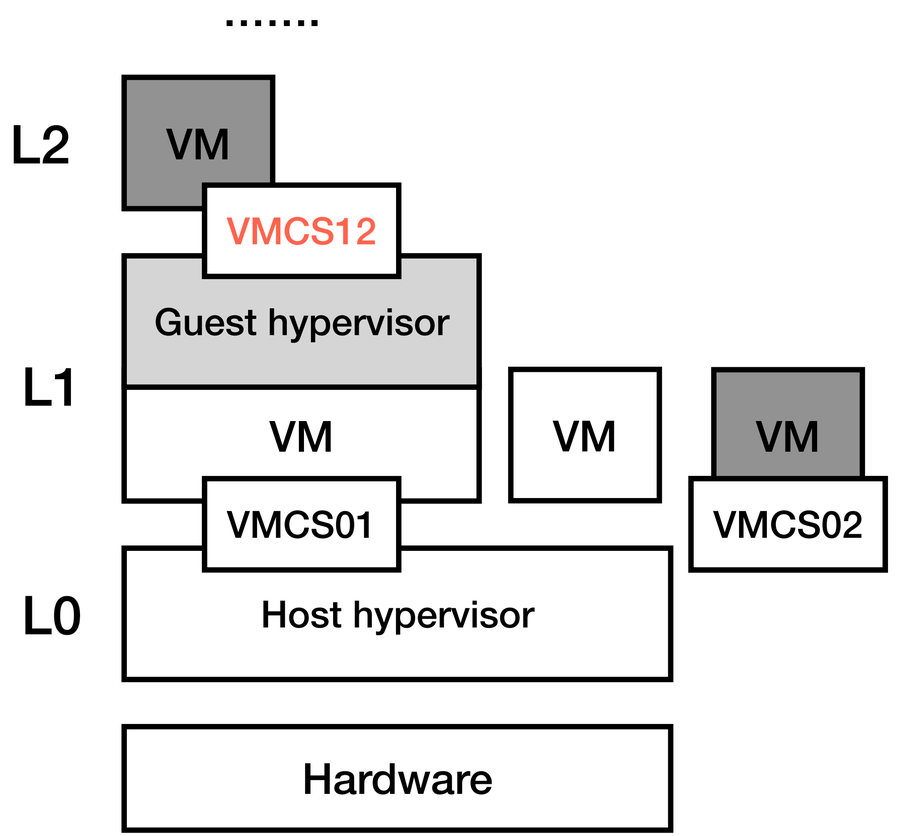
Nested VMX virtualization

- **L0 runs L1 with VMCS01**
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- vmlaunch traps to L0
- L0 prepares VMCS02 for L2
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- L2 causes a trap to L0
- L0 handles trap itself or forwards it to L1
- ….
- L0 resumes L2
- Repeat



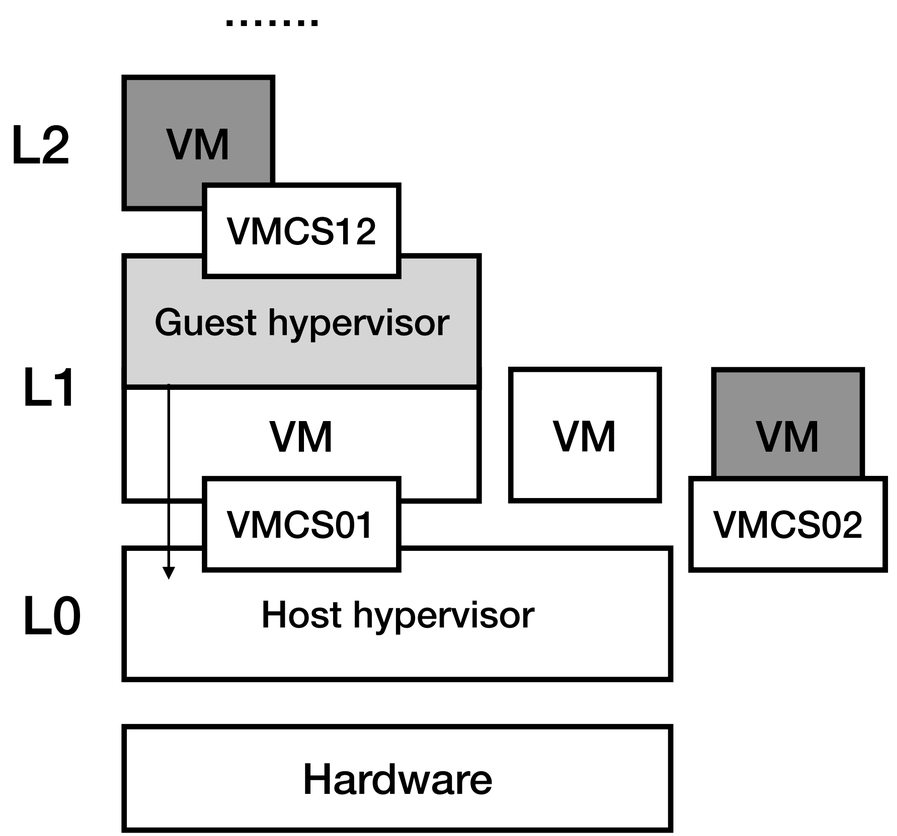
Nested VMX virtualization

- L0 runs L1 with VMCS01
- **L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2**
- vmlaunch traps to L0
- L0 prepares VMCS02 for L2
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- L2 causes a trap to L0
- L0 handles trap itself or forwards it to L1
- ….
- L0 resumes L2
- Repeat



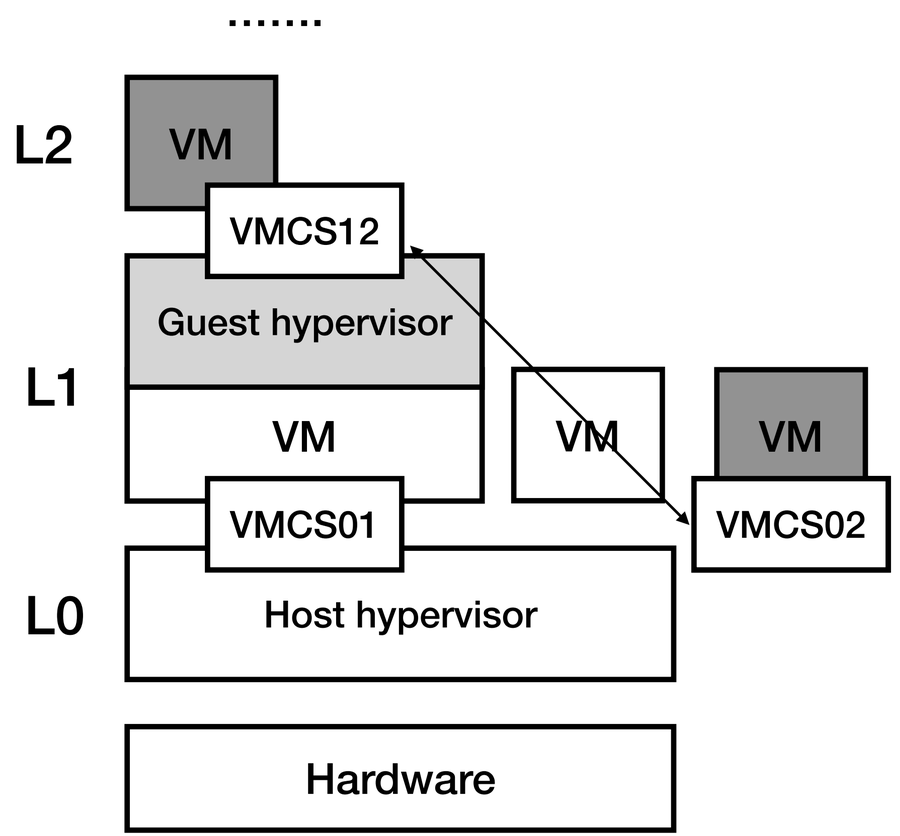
Nested VMX virtualization

- L0 runs L1 with VMCS01
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- **vmlaunch traps to L0**
- L0 prepares VMCS02 for L2
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- L2 causes a trap to L0
- L0 handles trap itself or forwards it to L1
- ….
- L0 resumes L2
- Repeat



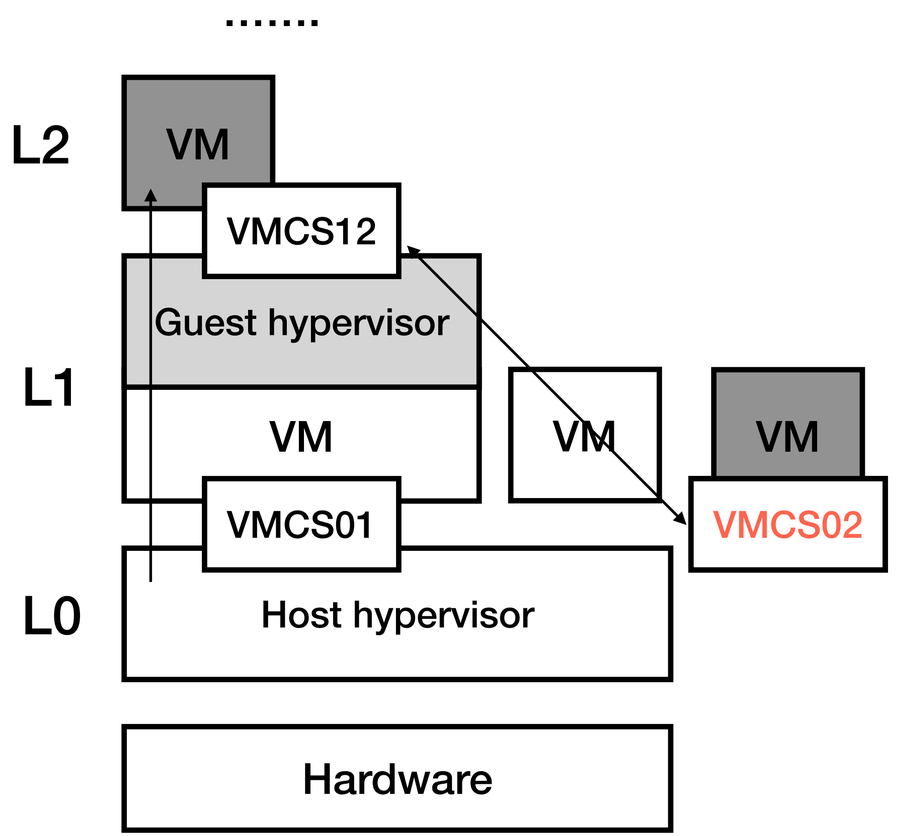
Nested VMX virtualization

- L0 runs L1 with VMCS01
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- vmlaunch traps to L0
- **L0 prepares VMCS02 for L2**
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- L2 causes a trap to L0
- L0 handles trap itself or forwards it to L1
- ….
- L0 resumes L2
- Repeat



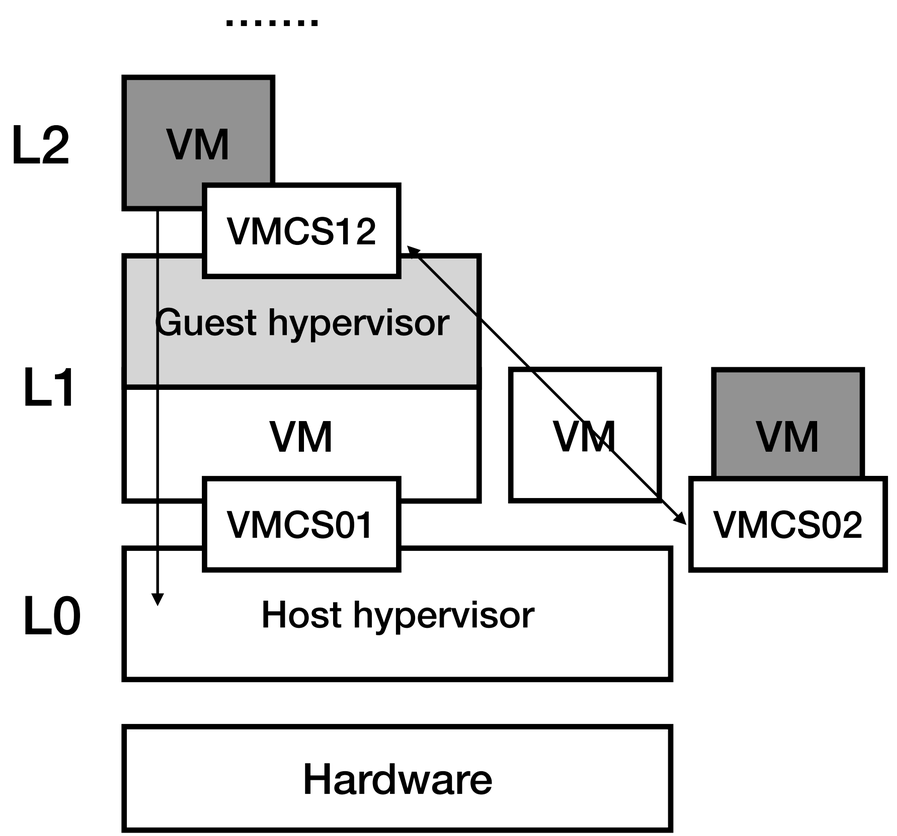
Nested VMX virtualization

- L0 runs L1 with VMCS01
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- vmlaunch traps to L0
- L0 prepares VMCS02 for L2
- **L0 loads VMCS02 to hardware and runs vmlaunch to enter L2**
- L2 causes a trap to L0
- L0 handles trap itself or forwards it to L1
- ….
- L0 resumes L2
- Repeat



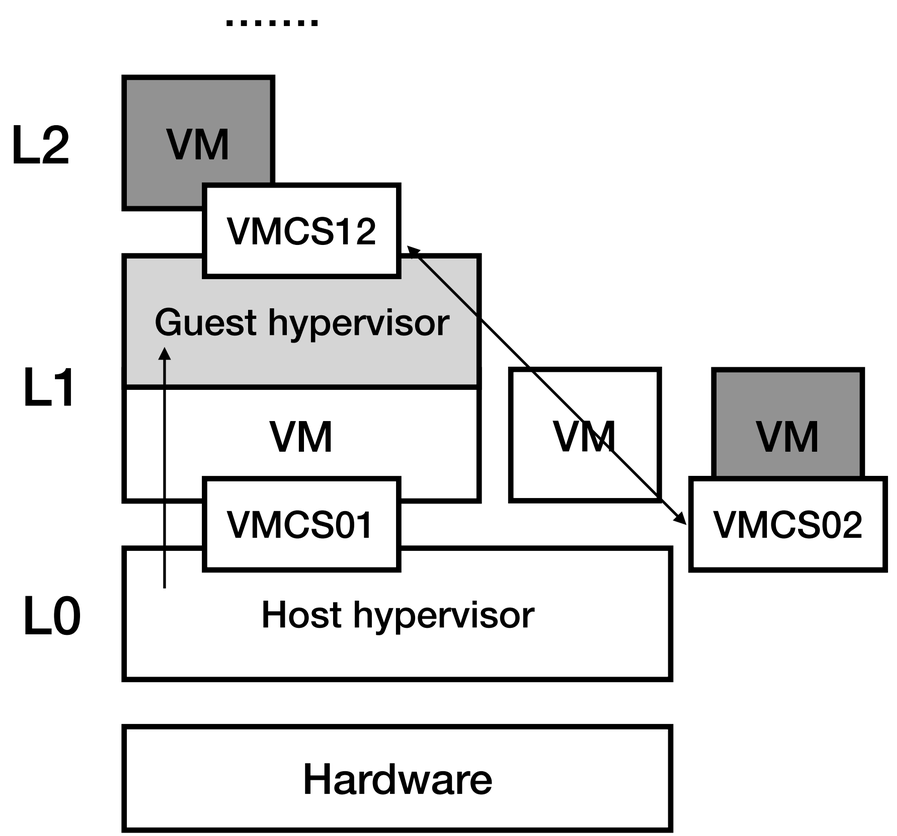
Nested VMX virtualization

- L0 runs L1 with VMCS01
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- vmlaunch traps to L0
- L0 prepares VMCS02 for L2
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- **L2 causes a trap to L0**
- L0 handles trap itself or forwards it to L1
- ….
- L0 resumes L2
- Repeat



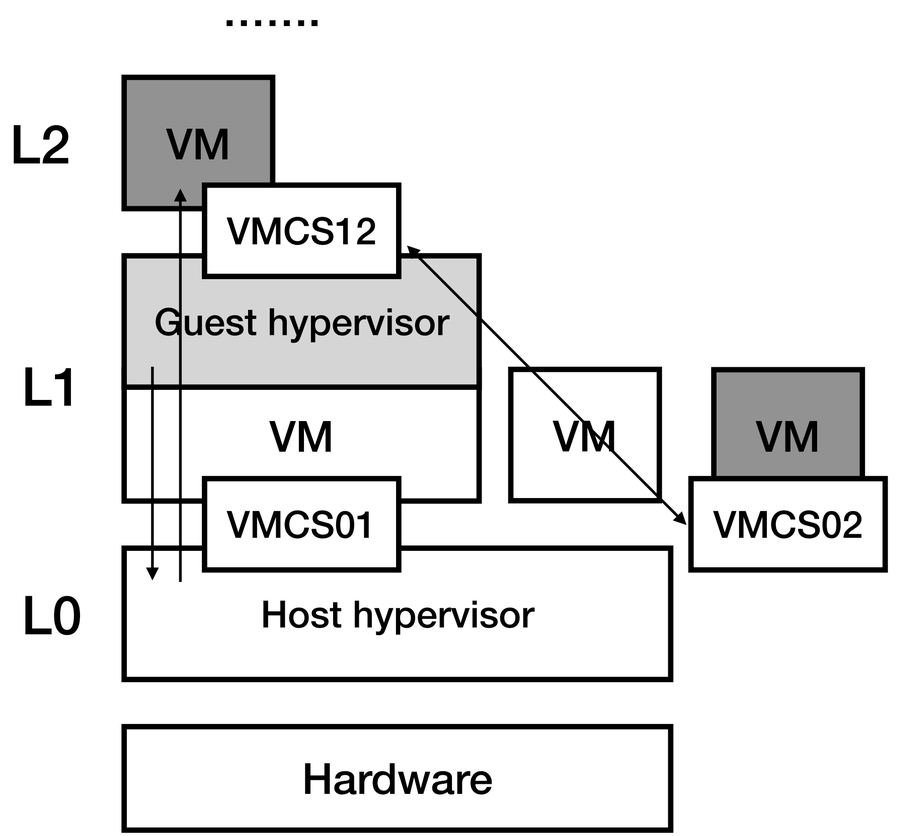
Nested VMX virtualization

- L0 runs L1 with VMCS01
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- vmlaunch traps to L0
- L0 prepares VMCS02 for L2
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- L2 causes a trap to L0
- **L0 handles trap itself or forwards it to L1**
- ….
- L0 resumes L2
- Repeat



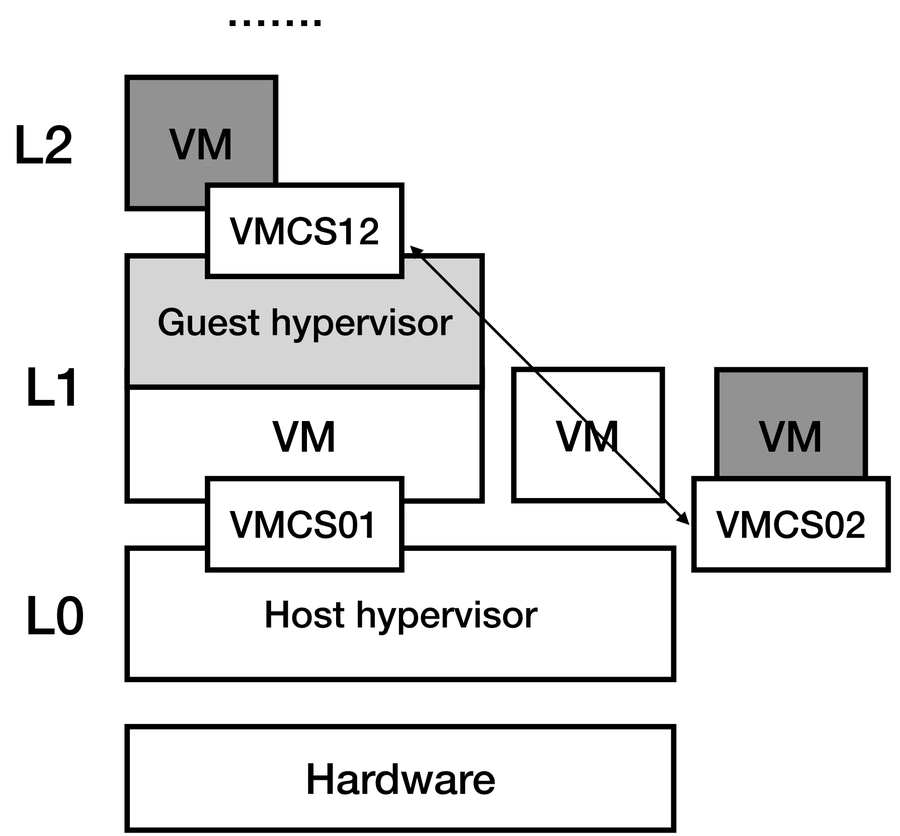
Nested VMX virtualization

- L0 runs L1 with VMCS01
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- vmlaunch traps to L0
- L0 prepares VMCS02 for L2
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- L2 causes a trap to L0
- L0 handles trap itself or forwards it to L1
- ....
- **L0 resumes L2**
- Repeat



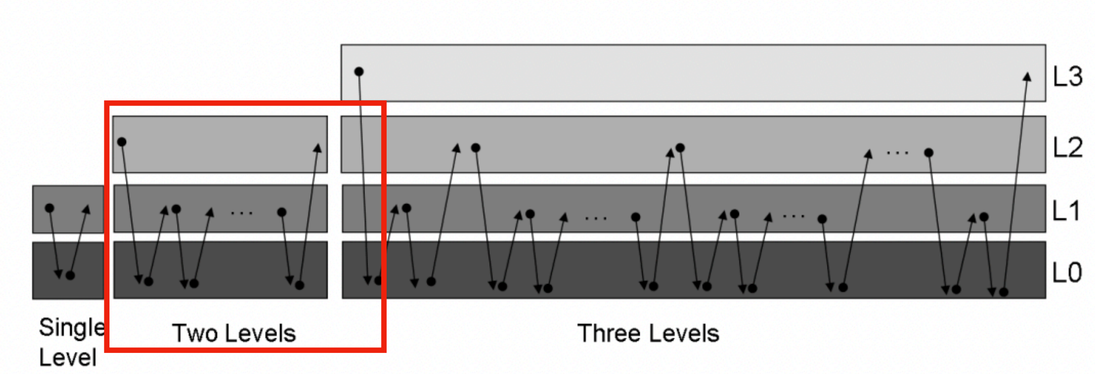
Nested VMX virtualization

- L0 runs L1 with VMCS01
- L1 prepares VMCS12 for its L2 VM and executes *vmlaunch* to enter L2
- vmlaunch traps to L0
- L0 prepares VMCS02 for L2
- L0 loads VMCS02 to hardware and runs vmlaunch to enter L2
- L2 causes a trap to L0
- L0 handles trap itself or forwards it to L1
- ….
- L0 resumes L2
- **Repeat**

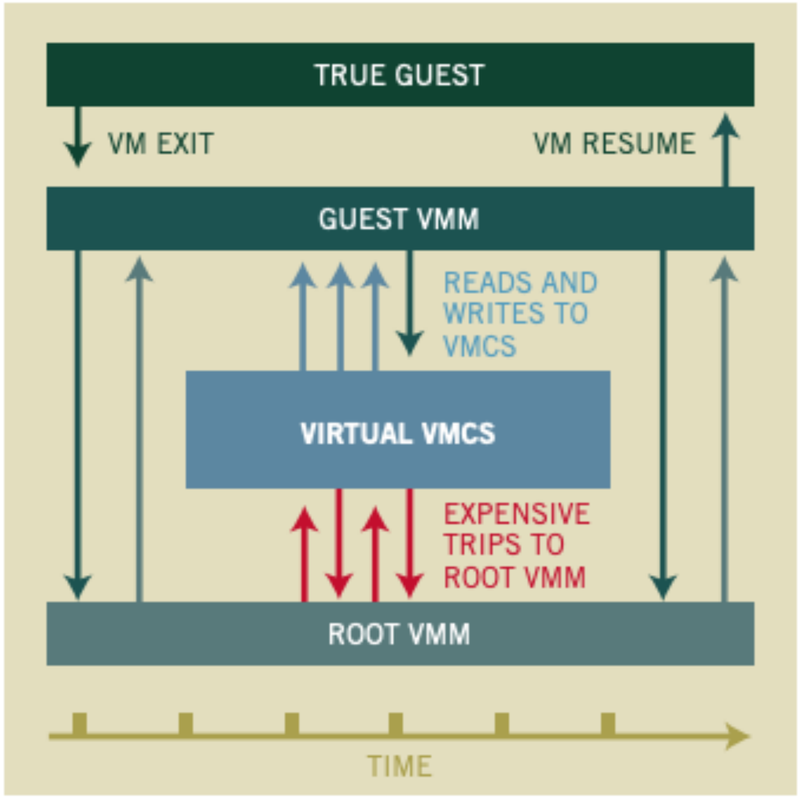


Problem: Exit Multiplication (1)

- A hypervisor does many things to handle a single VM exit: reads/writes VMCS, disables interrupts, etc. —> so does L1 do it for an L2's exit
  - Hypervisor uses *vmread* and *vmwrite* instructions to access VMCS on VMX
- L1 operations trap to L0 and lead to ***exit multiplication***
  - A single L2 exit can cause 40-50 L1 exits



Problem: Exit Multiplication (2)



Nested Virtualization Support on Intel VT-x

https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf

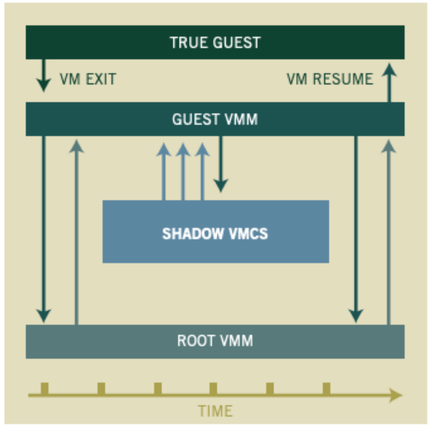
Problem: Exit Multiplication - Optimization (1)

IBM

- Goal: reduce the number of VM exits
- Reads or writes to VMCS can be performed by vmread/vmwrite without side-effects
- Turtles replace vmread/vmwrite from the guest hypervisors with memory load/store to a virtual VMCS in memory to avoid trapping
  - Host hypervisor batch updates the hardware VMCS during VM exits
  - Similar to the optimization for CPU virtualization (in lecture 2) using the register file

Problem: Exit Multiplication - Optimization (2)

- Intel VT provides hardware support for nested virtualization
  - VMCS Shadowing allows a guest hypervisor to access the Shadow VMCS using vmread/ vmwrite without trapping to the host (root) hypervisor
- Support nested virtualization on Intel x86 hardware with 6-14% overhead



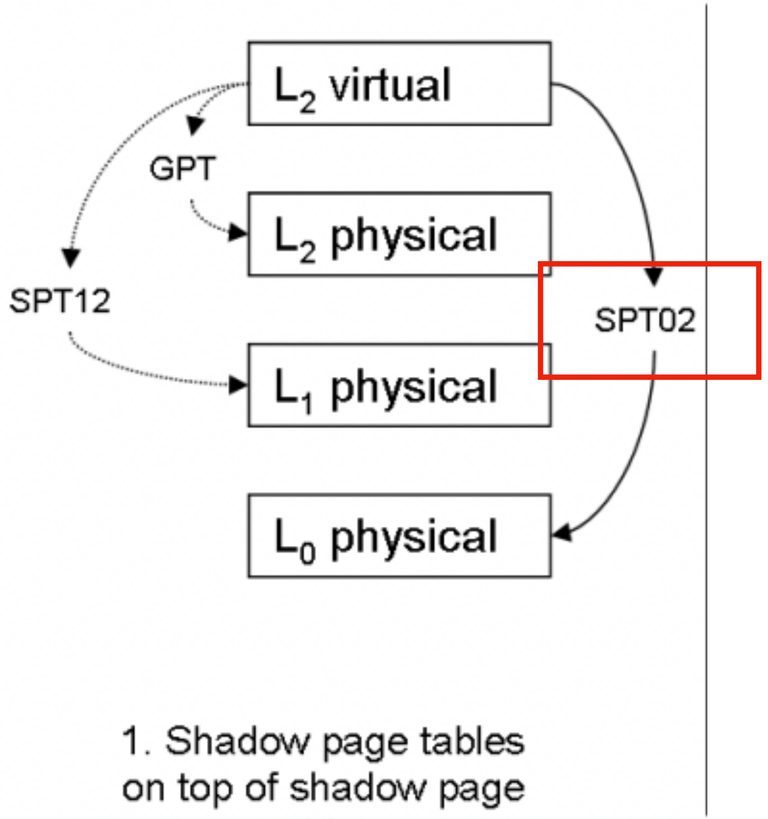
https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf

Nested Memory Virtualization (1)

- Goal: Wanted to leverage hardware extended page tables (EPT) if possible
- Turtles introduced *multi-dimensional paging* for nested memory virtualization
  - Intuition: the host hypervisor compresses page tables to match with EPT/NPT hardware
  - Recall that the MMU walks two levels of page tables: GPT & EPT/NPT
- Consider a case we have L0, L1, and L2
  - Must translate the address space from L2 -> L1 -> L0
  - L2 GPT, L1 GPT, L1 EPT, and L0 EPT

Nested Memory Virtualization (2)

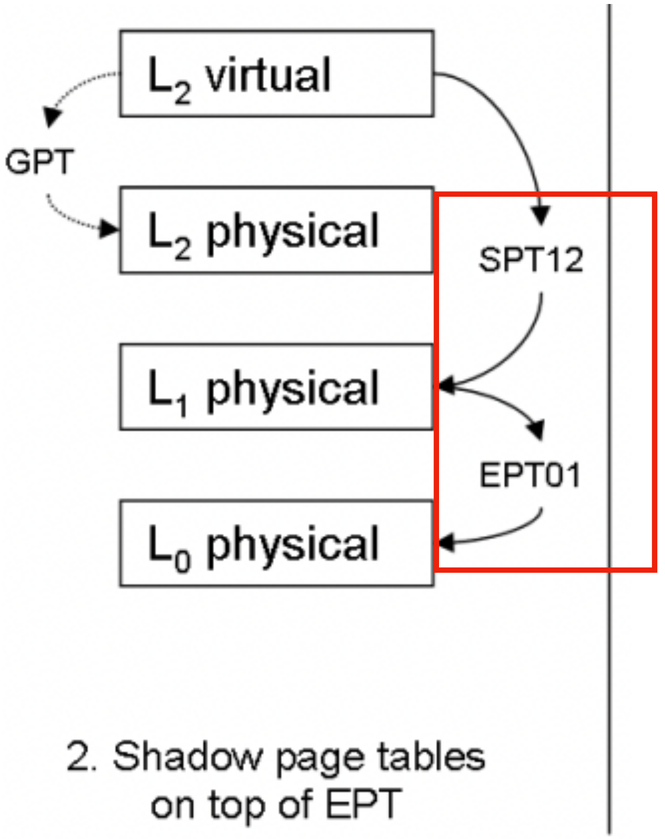
- First approach: shadow-on-shadow
  - Useful when EPT hardware is unavailable
  - L1 creates shadow page tables (SPT) for L2 to translate L2-gVA to L1-gPA
  - L0 creates SPTs for L2 to translate L2-gVA to L0-hPA
  - Managing two SPTs is too slow for nested VM due to exit multiplication!



1. Shadow page tables on top of shadow page tables

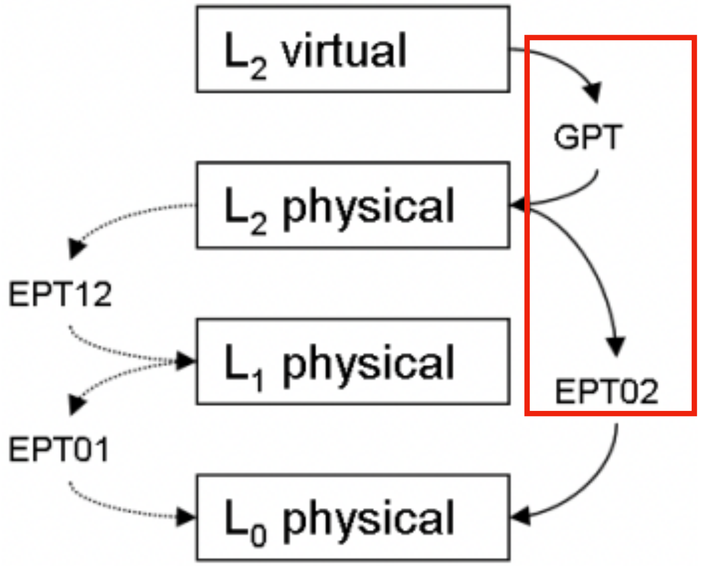
Nested Memory Virtualization (3)

- Second approach: shadow-on-EPT
  - Used when EPT hardware is available
  - L1 creates SPTs for L2 to translate L2-gVA to L1-gPA
  - L0 creates EPTs for L1 to translate L1-gPA to L0-hPA
  - Still suffers from exit multiplication



Nested Memory Virtualization (4)

- Third approach: ***multi-dimensional paging***
  - L1 creates EPTs for L2 to translate L2-gPA to L1-gPA (EPT12)
  - L1 uses EPTs built by L0 (EPT01)
  - L0 creates EPT for ***L2*** to translate L2-gPA to L0-hPA (EPT02)
    - L0 compresses EPTs (EPT12 & EPT01) in the EPT it builds for L2
  - Results in a lot fewer exits and achieves good performance



3. multi-dimensional paging
(EPT on top of EPT)

Nested I/O virtualization (1)

- The turtles project supports all classic types of virtual I/O for nested VMs
- Allows the nested VM to directly access hardware device for performance
  - The paper calls it multi-level device assignment (last in the table below)

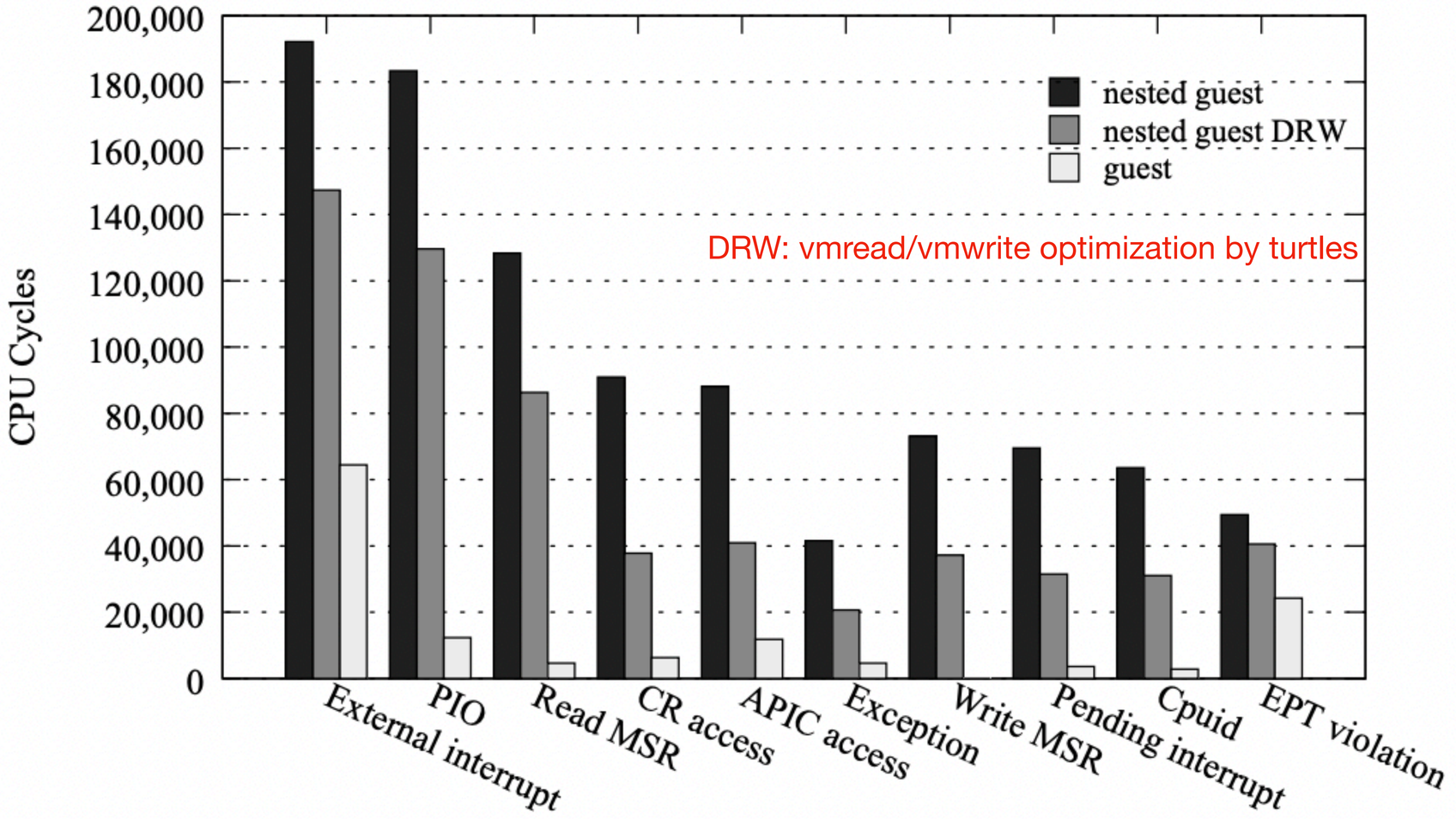
I/O virtualization method between L_0 & L_1	I/O virtualization method between L_1 & L_2
Emulation	Emulation
Para-virtual	Emulation
Para-virtual	Para-virtual
Device assignment	Para-virtual
Device assignment	Device assignment

Table 1: I/O combinations for a nested guest

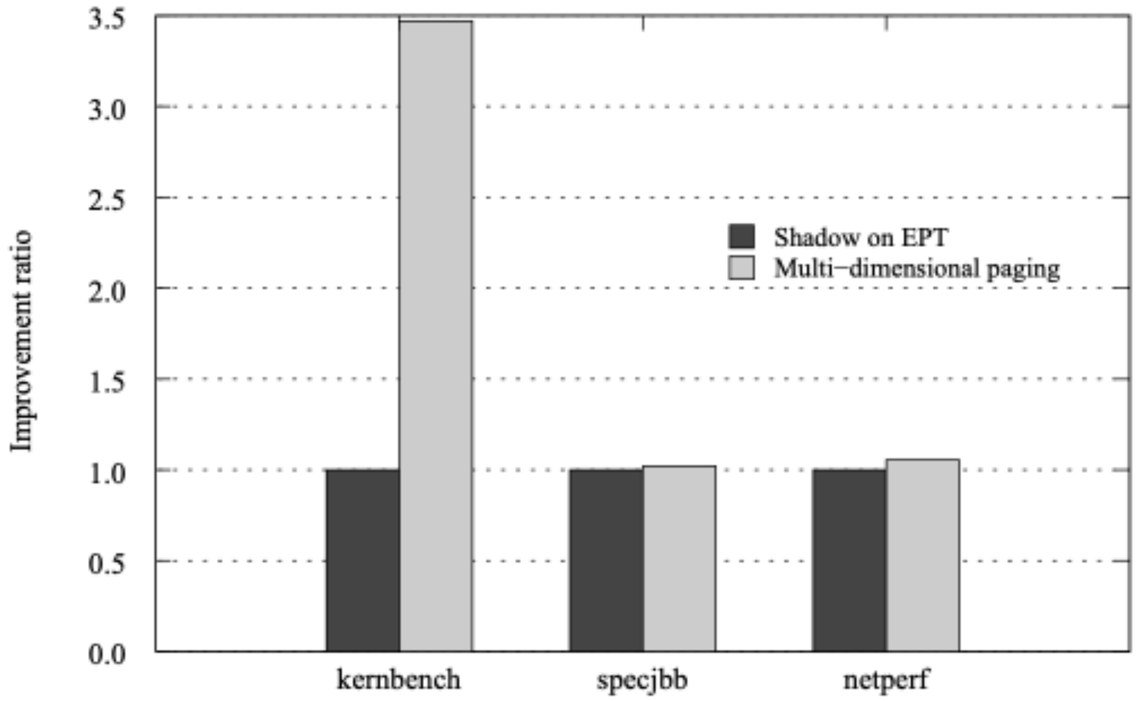
Nested I/O virtualization (2)

- It is unsafe to allow guest VMs to fully control hardware I/O devices
- The host hypervisor leverages the IOMMU to ensure VMs cannot perform malicious DMA
  - Virtualize the IOMMU for guest hypervisors
  - Compress multiple IOMMU translations onto the available hardware I/O page tables — similar to MMU page table compression!
- Allow the physical device to DMA with nested VM's memory directly

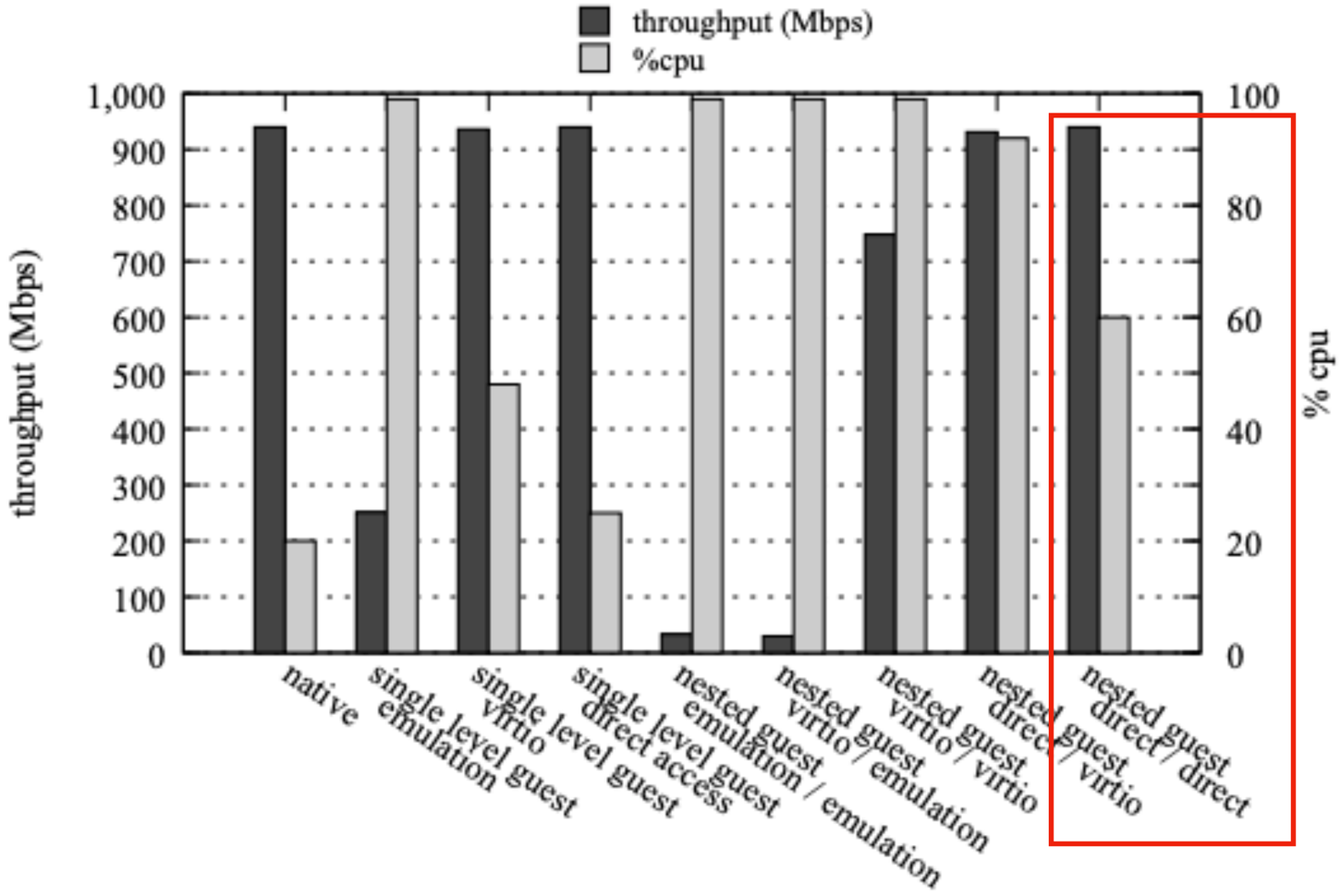
Turtles: Performance Evaluation (KVM on KVM)



Turtles: Performance Evaluation (KVM on KVM)



Turtles: Performance Evaluation (KVM on KVM)



CPU utilization is much higher than native! WHY?

Nested Virtualization for Arm

- Researchers at Columbia University first implemented nested virtualization for KVM Arm in 2017 and evaluated its performance [1]
  - KVM Arm baseline nested virtualization support performs very slow
  - Proposed architectural changes (idea similar to shadow VMCS) that adopted by Arm v8.4

[1] NEVE: Nested Virtualization Extensions for ARM, SOSP 17