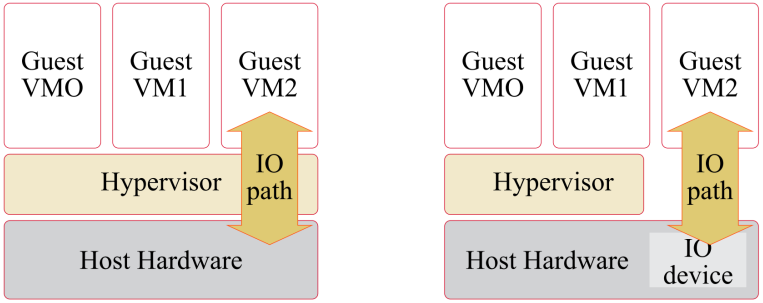
x86-64: I/O Virtualization with Hardware Support

Hardware and Software Support For Virtualization

Chapter 6.4

Direct Device Assignation

- How to **completely** avoid the overheads of I/O virtualization?
  - Forego I/O interposition: i.e., allow the guest-OS to **access directly** to physical devices directly (neither another VM nor the hypervisor can access them).
  - Hypervisor will assign statically the device to a single VM



- Such **naïve approach** will suffer from **(1)** scalability issues (e.g., limited physical devices per server while #VM>>) and **(2)** lack of control (e.g., **rogue guest-OS** can perform DMA operations in memory zones assigned to hypervisor/other VM)

- **Hardware support** is required to work around these issues.

(2.a) IOMMU

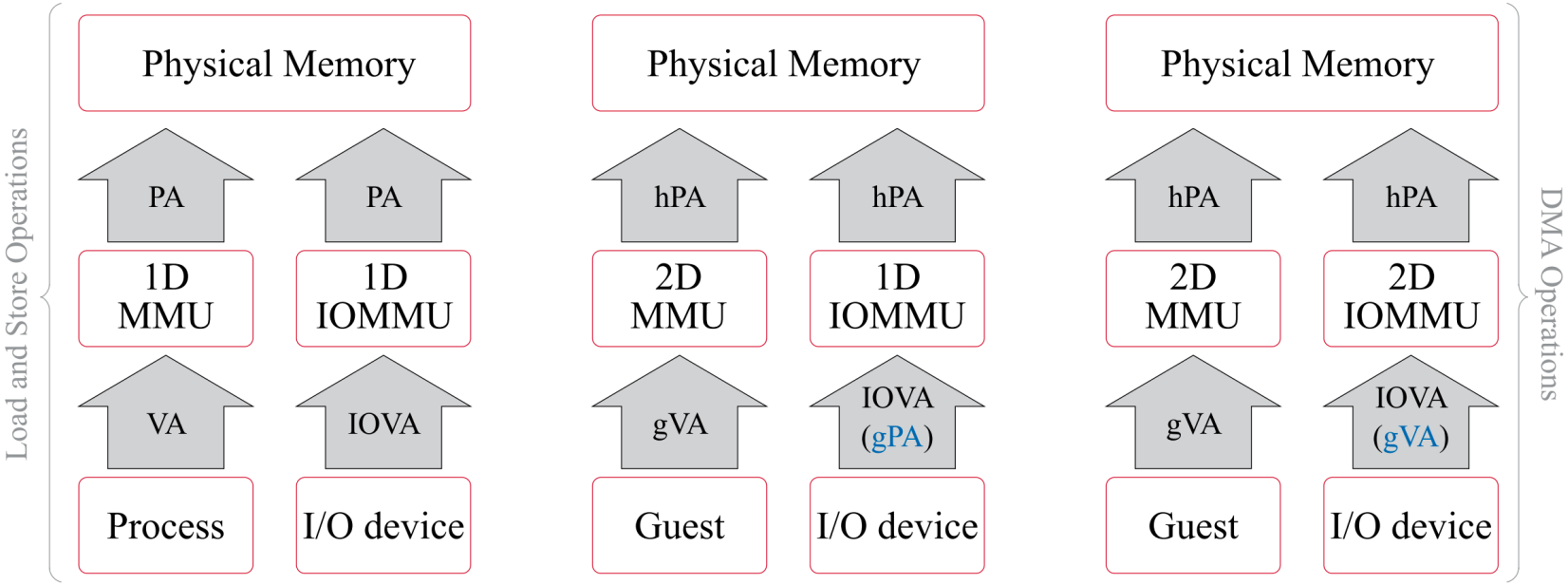
- Move data from/to ring buffers without CPU involvement? Issues:
  - VM should use "physical"-host addresses to perform DMA, which **breaks isolation** (MMU guarantees are no longer valid)
  - VM can also **trigger any interrupt vector indirectly** (VM can configure the device to raise any interrupt and vector)
  - Somehow the **principle of equivalence falls**: the VM is aware that its "physical" memory is not physical (since hypervisor will expose a fraction of the hardware)

- Vendors introduced **specific hardware** to handle such situations (**VT-d** and AMD-Vi).
  - "Chip-set" level (i.e., PCIe Root Complex)
  - Todays PCIe RC **inside the processor package** (supported by the "processor")

- Two components
  - **DMA remapping Engine** (DMAR)
    - Allow to perform DMA on virtual addresses (IOVA)
    - Hardware translate then in physical addresses using tables managed by the hypervisor/OS
  - **Interrupt remapping engine** (IR)
    - Redirect interrupt vectors fired by devices to the proper guest-OS handler

DMA Remapping

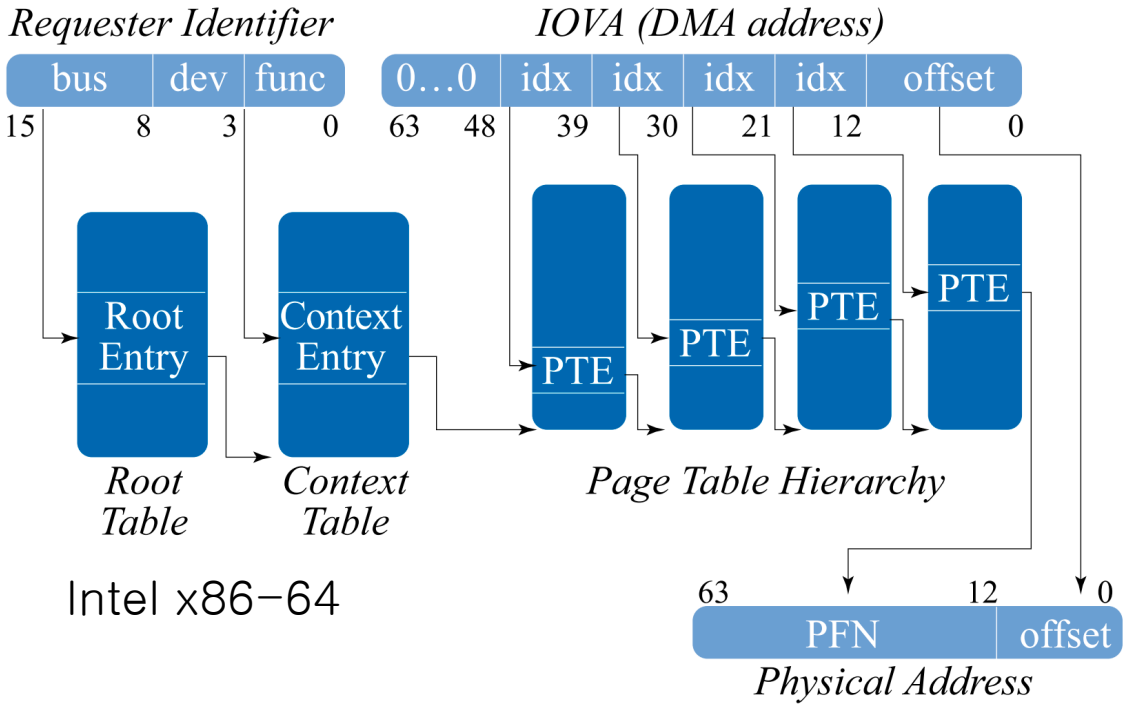
- (a) IOMMU plays the role of MMU in CPU
  - No virtualization: A 1D page table handled by the OS (**per device**)
- (b) EPT and 1D table in I/O (i.e., hypervisor does not expose the device to the guest-OS)
  - Should intervene if translation changes (i.e., **Shadow Page Table for I/O???**)
- (c) EPT and 2D IOMMU. Hypervisor expose the IOMMU to the guest
  - Hardware should support it. **"EPT" for I/O (device and VM)**



(a) Bare metal, 1D-IOMMU (b) Virtual, 1D-IOMMU (c) Virtual, 2D-IOMMU

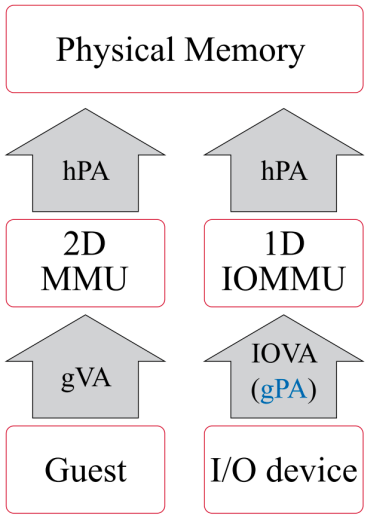
DMA remapping in PCIe (I)

- IOMMU resides in the PCIe Root Complex (**hardware**)
  - DMA propagates upstream (towards memory) and translation is done in PCI-QPI jump (unCore is connected to PCIe via (*Quick Path Interconnect* **Intel**)(*Infinity-Fabric* **AMD**)
  - Each DMA buffer will be associated to a **BDF** (16bit unique device in the hierarchy)
  - Context table is used to access the page table root
- Similarly, to MMU, IOMMU has an **IOTLB**
  - Table is only accessed **in misses by hardware**.
  - Tables are indexed with BDF (equiv to %cr3)
- Page misses are **not** handled gracefully. **DMA buffers are pinned!**

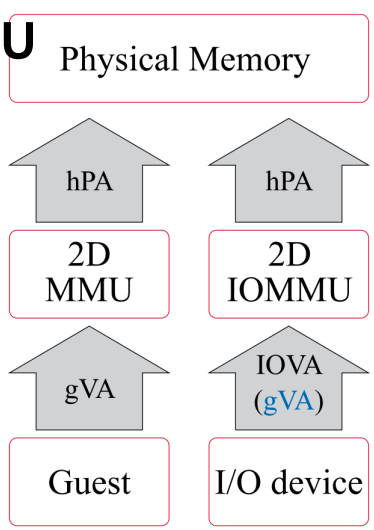


DMA remapping in PCIe (II)

- 1D IOMMU is used by old-servers
  - Hardware **can't transform** g(uest)PA into h(ost)PA
  - **No Shadow Page Table.** Maps gPA into real IOVA
    - I/O maps perceived by guest-OS is identical to the server
    - Works because the device only can be assigned to 1 VM (if no user-level I/O)
- From Skylake (8th gen) and onwards supports **2D-IOMMU**
  - Allows to protect the guest-OS from **malicious I/O devices**
  - Backward compatibility (e.g., Use 32b devices in 64b systems [place **hPA DMA reg. above 2^{32}**])
  - Allows the guest-OS to map devices to processes (**user-level device drivers**)



(b) Virtual, 1D-IOMMU



(c) Virtual, 2D-IOMMU

(2.b) Message Signaled Interrupts (MSI)

- Third type of I/O-CPU interaction (beyond PIO, MMIO)

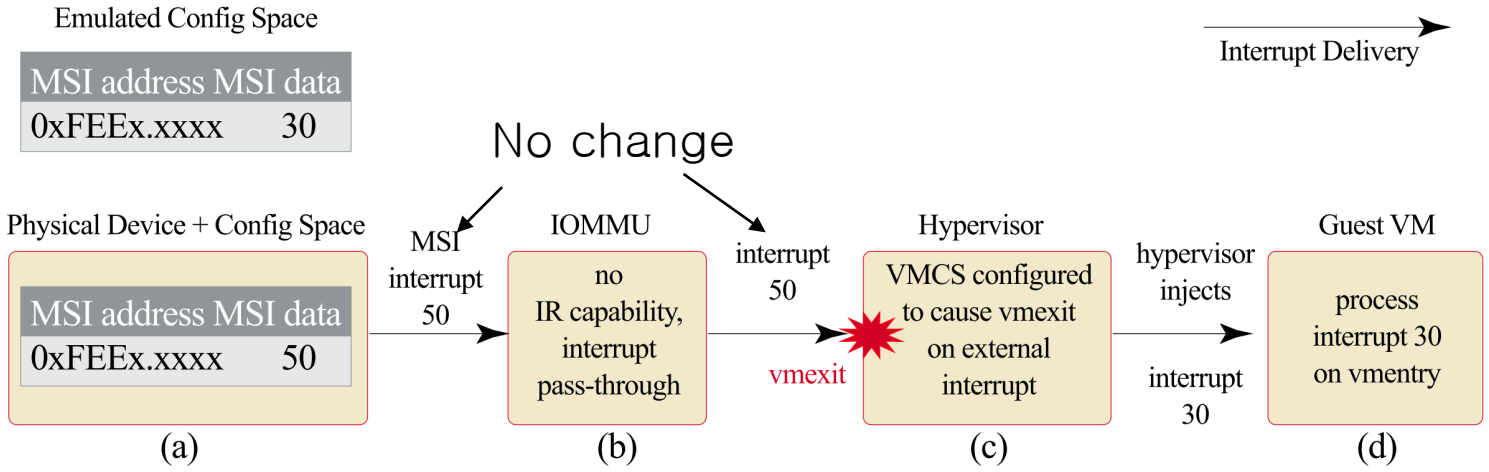
- Allows to a device to send a PCIe **interrupt packet** whose destination is a LAPIC of a specific core
  - An alternative way to send an asynchronous event **from the device** (as opposed to classical s*ingle-pin* interruptions)
  - Multiple "events" per device (and desired actions from the CPU)
  - It's a PCIe packet: propagates the tree until reaches the PCIe RC (like any DMA transfer. Target addr. is the LAPIC)

- The OS configures (via PIO or MMIO) the device to use MSI by writing the address of the **target LAPIC** and desired **vector interrupt**
  - Writing into the message-address and message-data registers in middle part of PCIe config space

- MSI supports 32 interrupts per device and 2048 in MSI-X (PCI 3.0)
  - **IOAPIC/MSI-X** replaces the core-level controller by a **package level** controller

Interrupt Remapping PCIe (I)

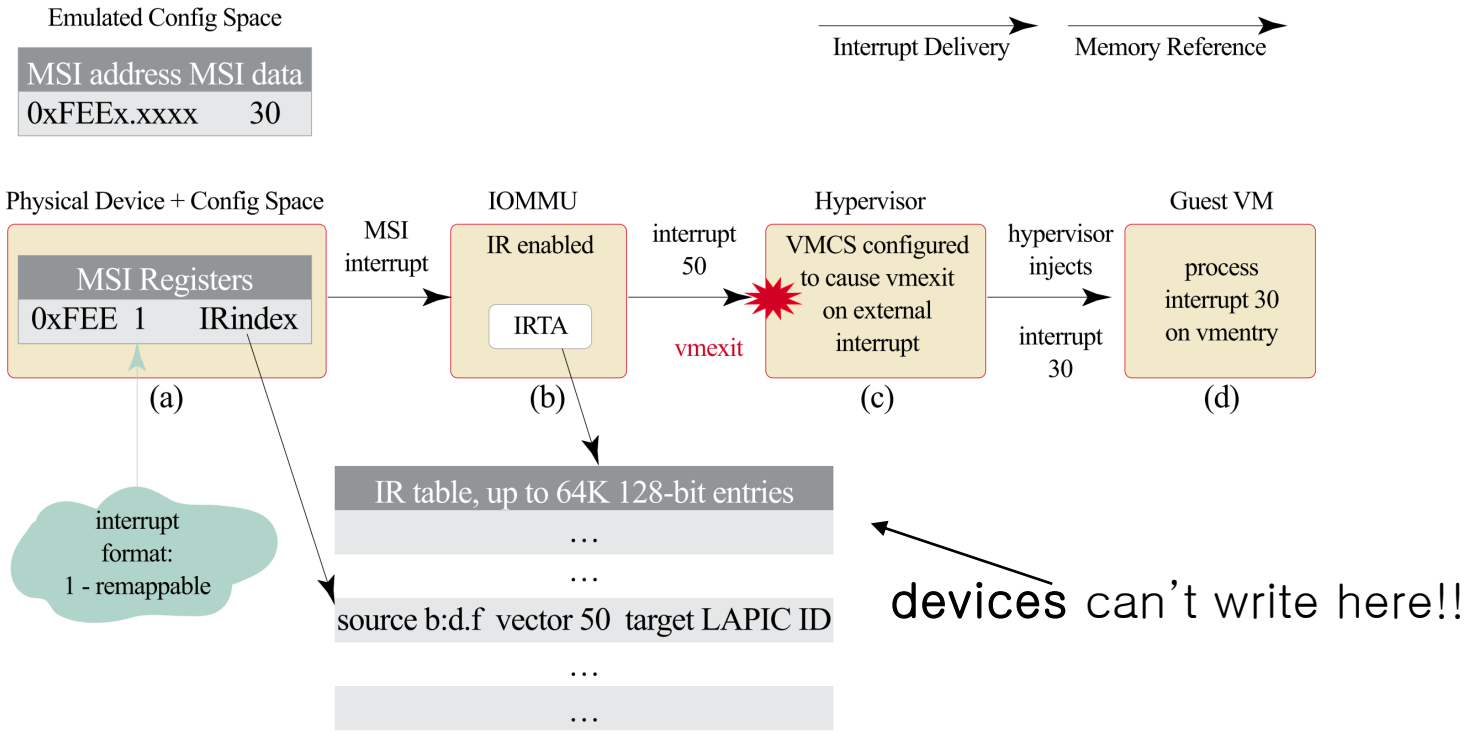
- PCIe defines (MSI/MSI-X) interrupts similar to DMA memory writes directed to "**interrupt space**" (PCIe RC defines this. In x86 `0xFEEx_xxxx`)
- **Non-Interrupt Remapping case** (**v** virtual machine, **d** real device) [**broken]**
  1. Setting in **v** interrupt vector (30) and target LAPIC for **d** requires to write in emulated config space (vmexit)
  2. Hypervisor decides to use 50 as vector: need to intercept 50 and remap it to the actual 30 **but the MSI packet passes unchanged trough the IOMMU**



- **Unsafe**: **v** can remap DMA-writes into interrupt space making the hypervisor to be vulnerable to guest attacks (by writing any value in the vector 50 and redirecting the hypervisor elsewhere). **d is controlled by v (can be programed to write anything by DMA in MSI space!!** For example, changing 50 to something else
  - IOMMU can't distinguish between a legitimate MSI interrupt and a **rogue DMA** that pretend to be an interrupt!

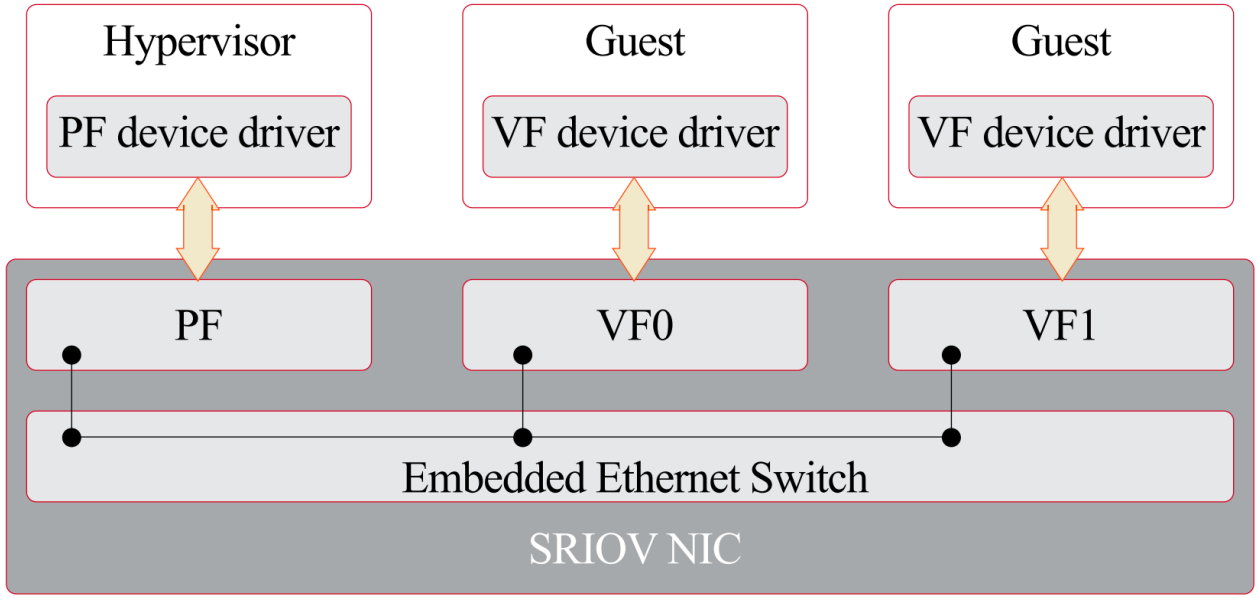
Interrupt Remapping in x86-64 (II)

- IR capability eliminates the vulnerability
  - The format of the MSI now only can contain an index in the IR table (not any value) when **the remappable bit** is set
  - To prevent spoofing IR contains BDF (indicates that **d** is allowed to raise interrupts associated to the IRIndex)



(1) SRIOV (Single-root I/O Virtualization)

- Addresses the issue of making the physical device allocation scalable
  - Servers can house a limited number of devices
  - From cost perspective, its impractical to have one physical device (e.g., NIC) per VM
- SRIOV I/O devices presents multiple instances of the self to the software (i.e., devices are "self-virtualized"). Its is a standard supported by PCI-SIG
- SRIOV devices are defined by a Physical Function (PF) and multiple Virtual Functions (VFs)
  - PF is a standard PCIe functions
  - VF is a lightweight PCIe function (partial, e.g., don't have power management capabilities)
- VFs are mapped to VM and can operate using IOMMU
- Up to 64K VFs in standard (current NIC 512VFs)



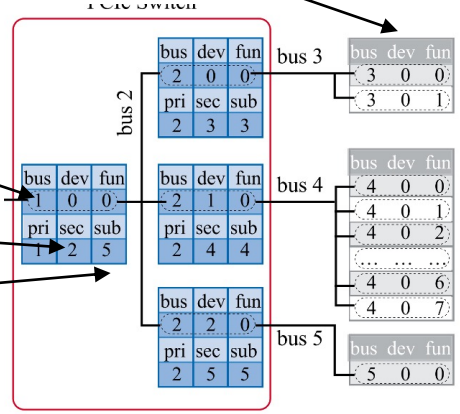
SRIOV: lscpi output

- `lspci -s 06:00.0 -vv`

```
06:00.0 Ethernet controller: Intel Ethernet Controller 10-Gigabit X540-AT2
        Subsystem: Intel Corporation Ethernet 10G 2P X540-t Adapter
        Flags: bus master, fast devsel, latency 0
        Memory at 91c00000 (64-bit, prefetchable) [size=2M]
        Memory at 91e04000 (64-bit, prefetchable) [size=16K]
        Expansion ROM at 91e80000 [disabled] [size=512K]
        Capabilities: [70] MSI-X: Enable+ Count=64 Masked-
        Capabilities: [a0] Express Endpoint, MSI 00
        Capabilities: [150] Alternative Routing-ID Interpretation (ARI)
        Capabilities: [160] Single Root I/O Virtualization (SR-IOV)
                Total VFs: 64, Number of VFs: 0
                VF offset: 128, stride: 2, Device ID: 1515
                Supported Page Size: 00000553, System Page Size: 00000001
                Region 0: Memory at 92300000 (64-bit, non-prefetchable)
                Region 3: Memory at 92400000 (64-bit, non-prefetchable)
        Capabilities: [1d0] Access Control Services
```

Review: PCIe

- **Node Enumeration**
  - Each node (function) and edge (bus) is uniquely identified in the PCI graph as 16-bit N `bus:device:function(BDF)` (8, 5 and 3 bits)
- **Edge Enumeration**
  - Buses are numbered from 0 upward (up to 256) in order
  - Each G bridge is denoted by
    - **Primary bus**: upstream that G connects
    - **Secondary** bus: first downstream bus
    - **Subordinate**: last downstream bus

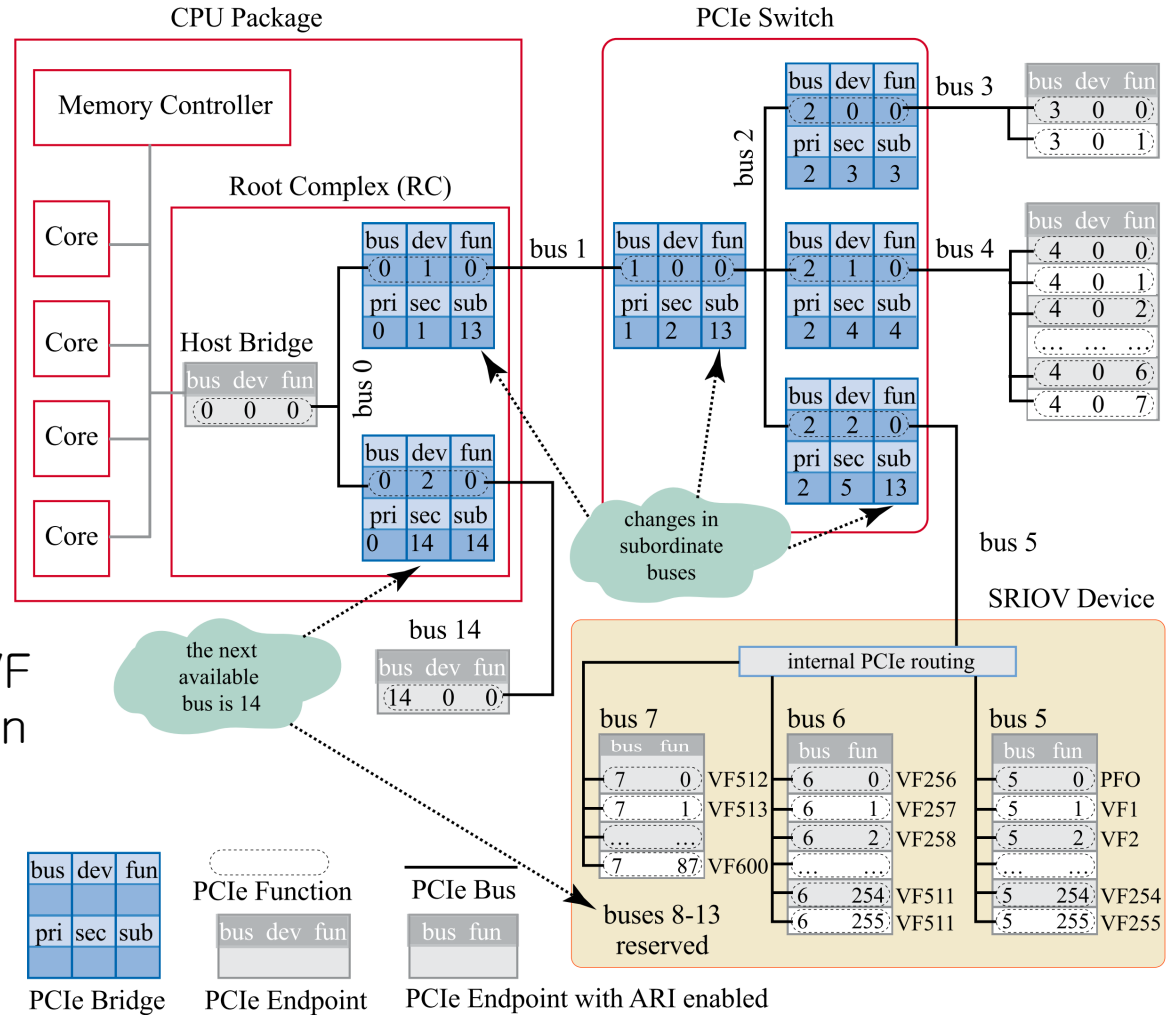


SRIOV: Alternate Routing ID (ARI) Enabled

- >8 functions at end-point (5bit device used for function: 256 function)
- Multiple buses inside the device (must be considered in the subordinate buses in edges)

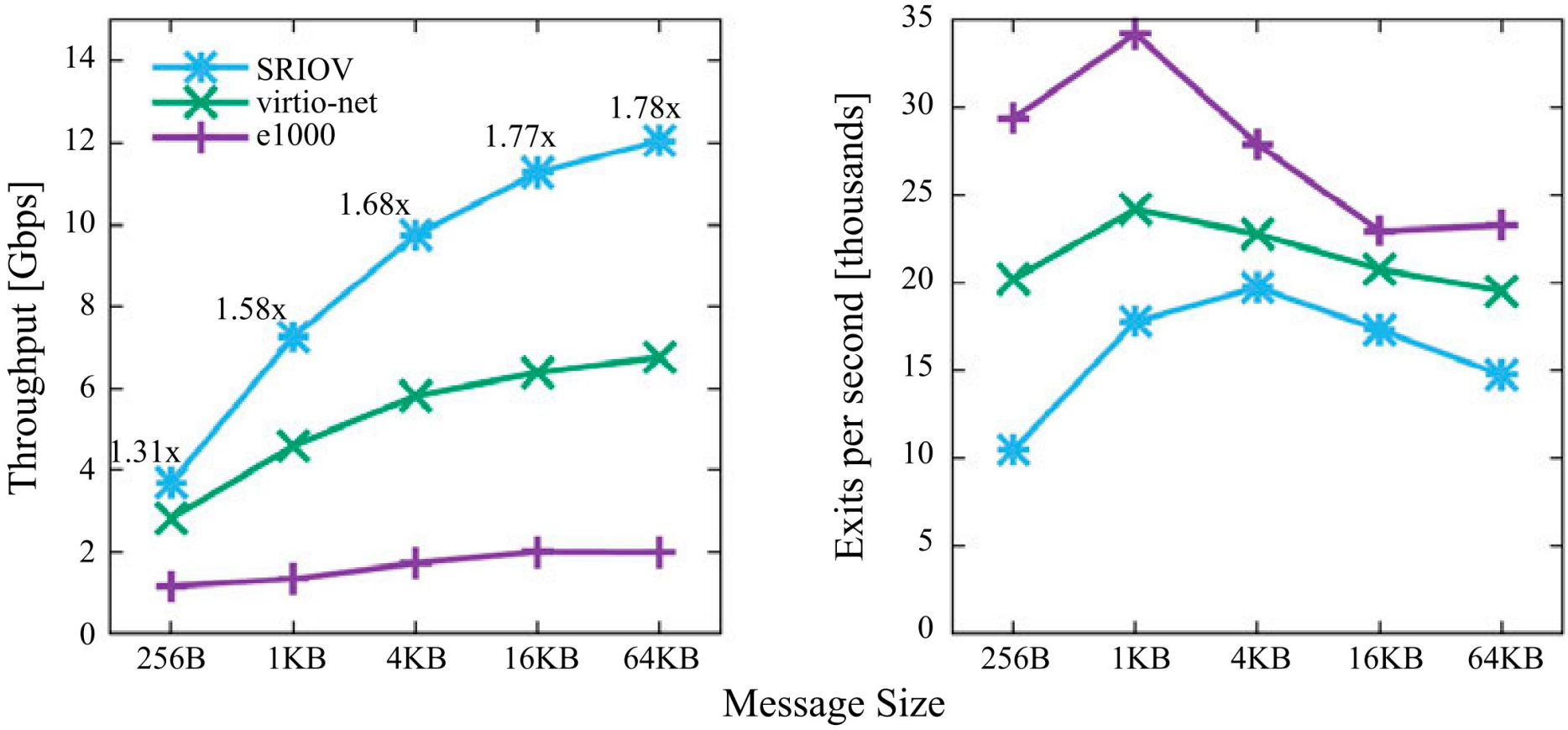
Up to 2048VF
(600 allocated)

Total supported VF creates a "hole" in the hierarchy



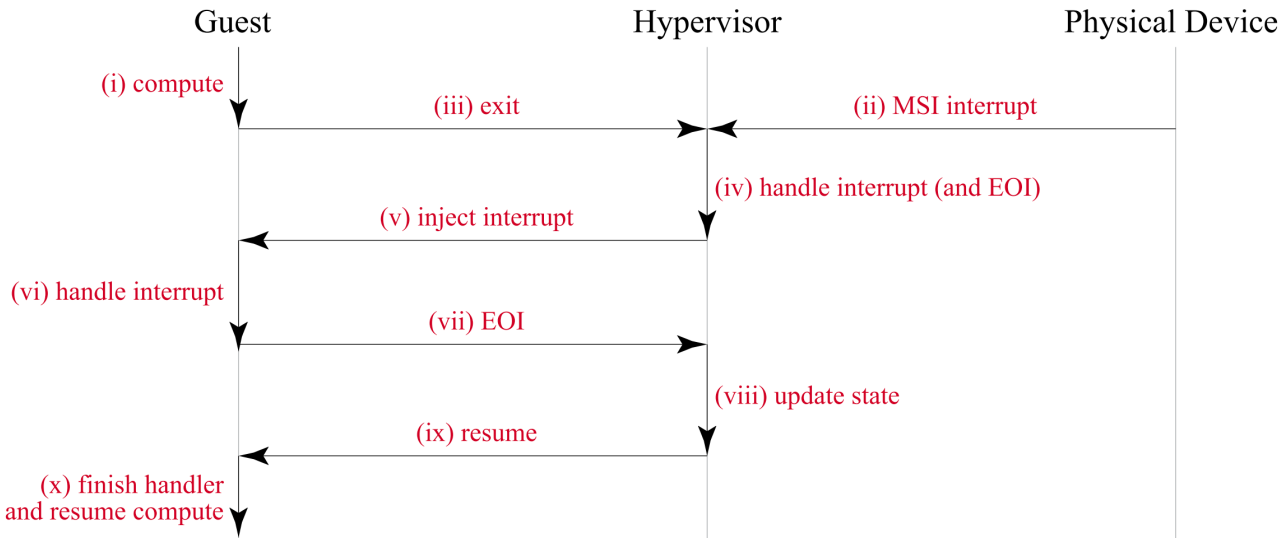
SRIOV: Performance and `vmexits` Netperf

- Lower number of exits but still significative



Interrupt Overheads

- SRIOV and IOMMU allows to directly communicate VM with devices but **does not prevent exist** when the **device "talk-back"** (i.e., interrupts)



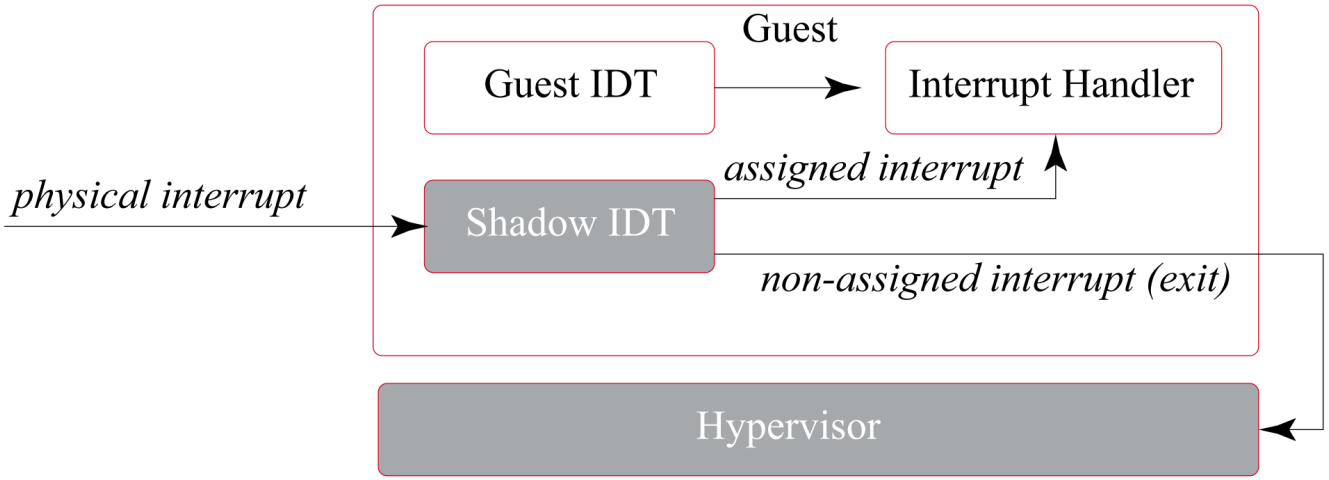
1. When guest receives an external interrupt, exits (marking the reason in VMCS)
2. Hypervisor identify the cause and handles the interruption (signaling it to the LAPIC via EOI clearing the ISR bit register)
3. The hypervisor injects the interruption into the guest via VMCS
4. Guess acknowledges the the interrupt handling by signaling the LAPIC (emulated) another exist is required
5. Hypervisor updates the emulated LAPIC state and resume the handler

Interrupt Overheads : Assigned EOI register

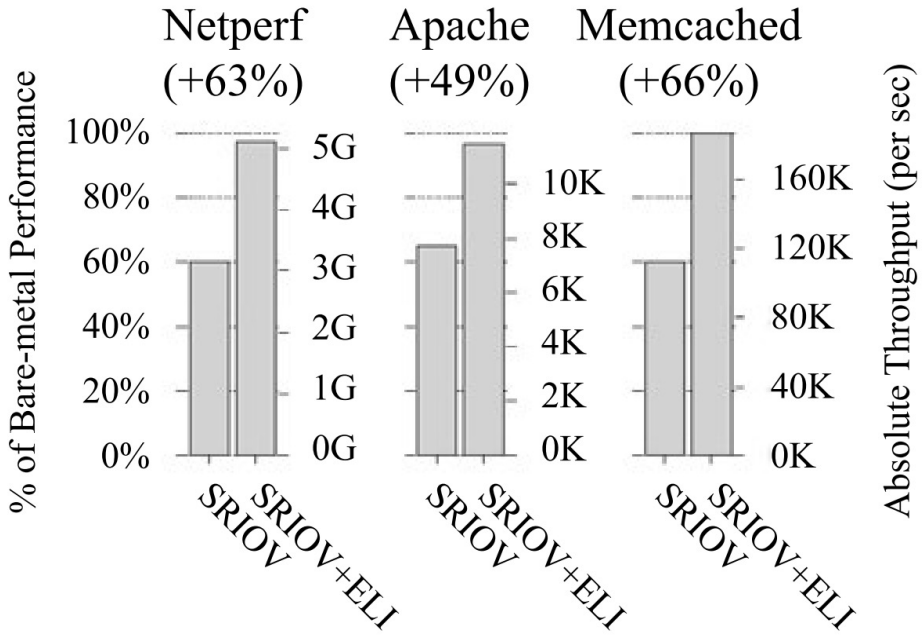
- Previous approach requires 2 `vmexits` per interrupt
  - Virtualization doubles interruption overhead (can be substantial in stress conditions)
- LAPIC signals End-of-interruption handling via the EOI register
  - In current generation (2xAPIC) is possible to allow the guest to write on it (not in the emulated one) [saves 2nd `vmexit`]
  - If not, LAPIC will be emulated

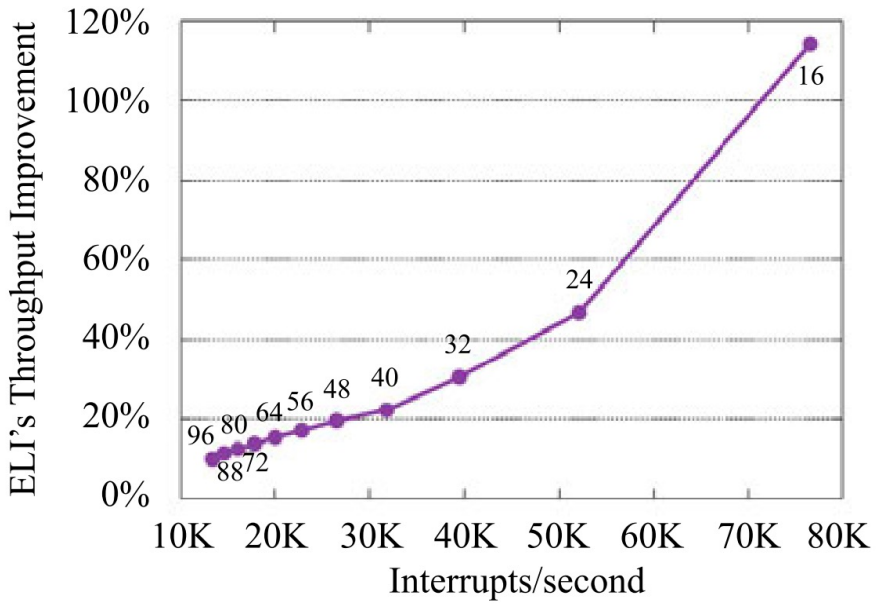
Exitless Interrupts: Assigned Interrupt

- Prevent any `vmexit` delivering directly the interrupt to the guest (Exitless Interrupt or ELI). Software solution turning off "external-interrupt" bit in VMCS
- Use a shadow IDT with pointers to handlers into the guest
  - Still in certain devices exist can be forced making entries in the SIDT non-present (e.g., devices not assigned to the guest)



Exitless Interrupts: Performance





Posted Interrupts

- ELI Increases hypervisor complexity (not widely used)
  - Must handle shadowing.
  - Extra security measures to prevent guest attacks (e.g., Guest never ack completion will affect hypervisor, since **VM and hypervisor share the physical LAPIC EOI**)

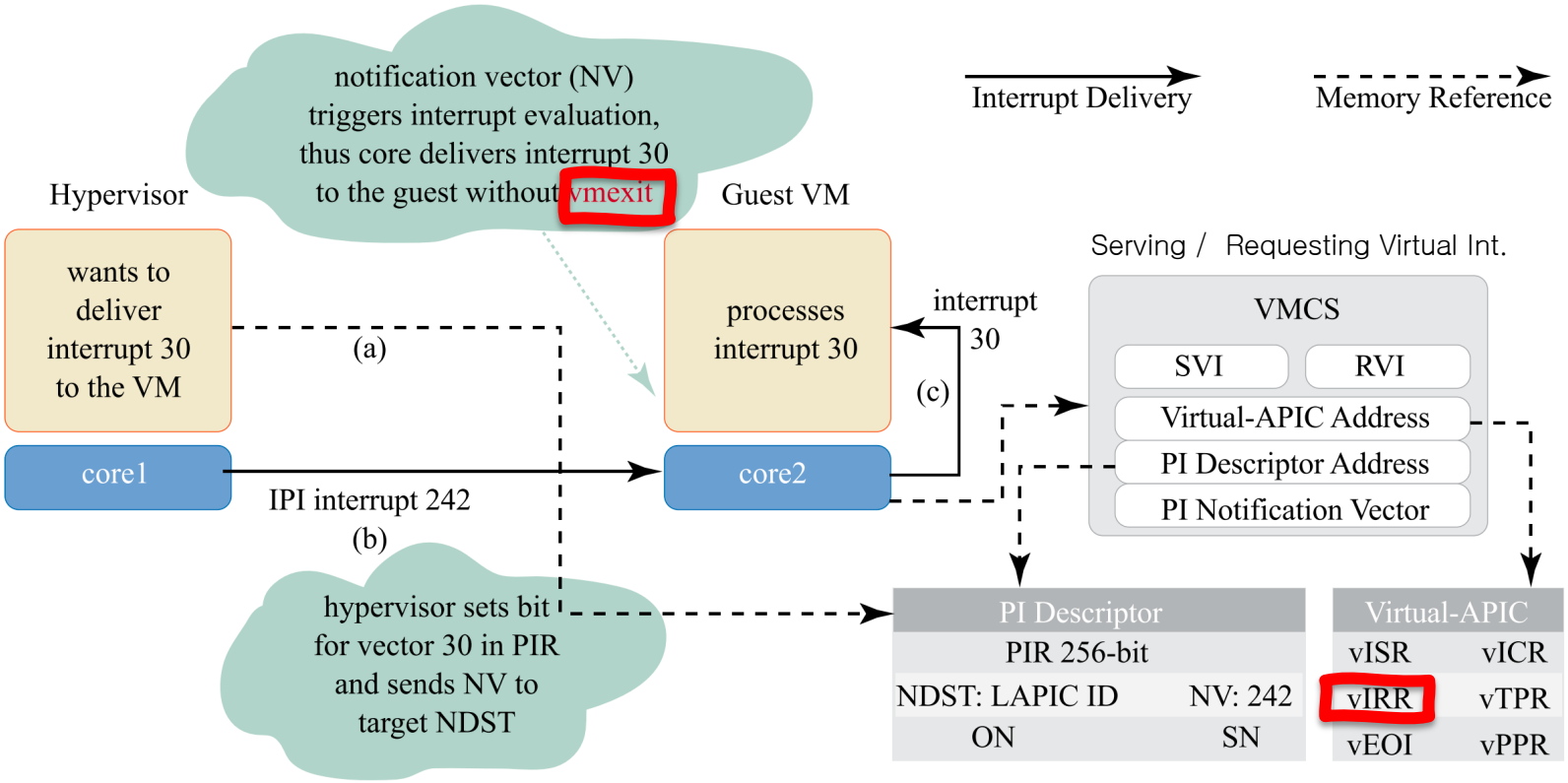
- Posted interrupts allows to easily assign specific interrupts of specific devices to specific guests-OS and the remaining interrupt directed to the host.
  - **CPU posted interrupts**: inject interruptions by hypervisor in one core, affecting a guest-OS in another core (without hypervisor intervention)
  - **IOMMU posted interrupts**: Interrupts delivered by I/O devices into guest VMs

- The hypervisor can configure guest to receive and acknowledge the completion of interrupts without the involving of the hypervisor on the guest's code

- Introduced by **APICv** (Intel) /**AVIC** (AMD) in server processors since Skylake (8^{th})

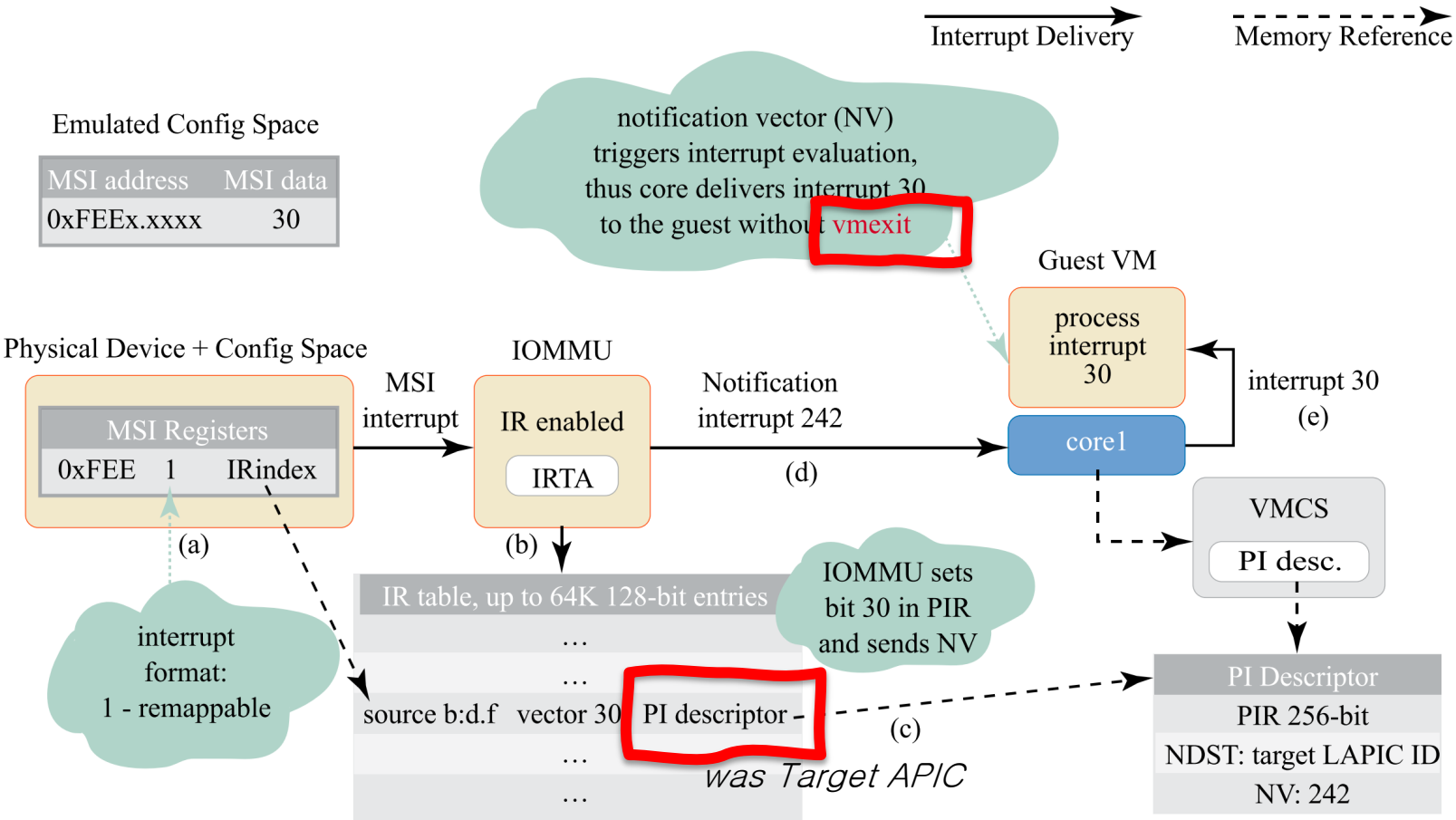
CPU Posted Interrupts

- Hypervisor (via VMCS) instruct the VM to use a **Virtual APIC**
  - `vISR, vIRR, vICR, vEOI, …` mapped there
  - Hardware intercept the guestOS accessed and real APIC is updated accordingly (with no `vmexit`, unless desired)
- Hypervisor uses **Posted Interrupt Descriptor** to relay the interrupts
  - Post-interrupt Request(PIR), Notification Vector(NV), Notification Destination (NDST), …



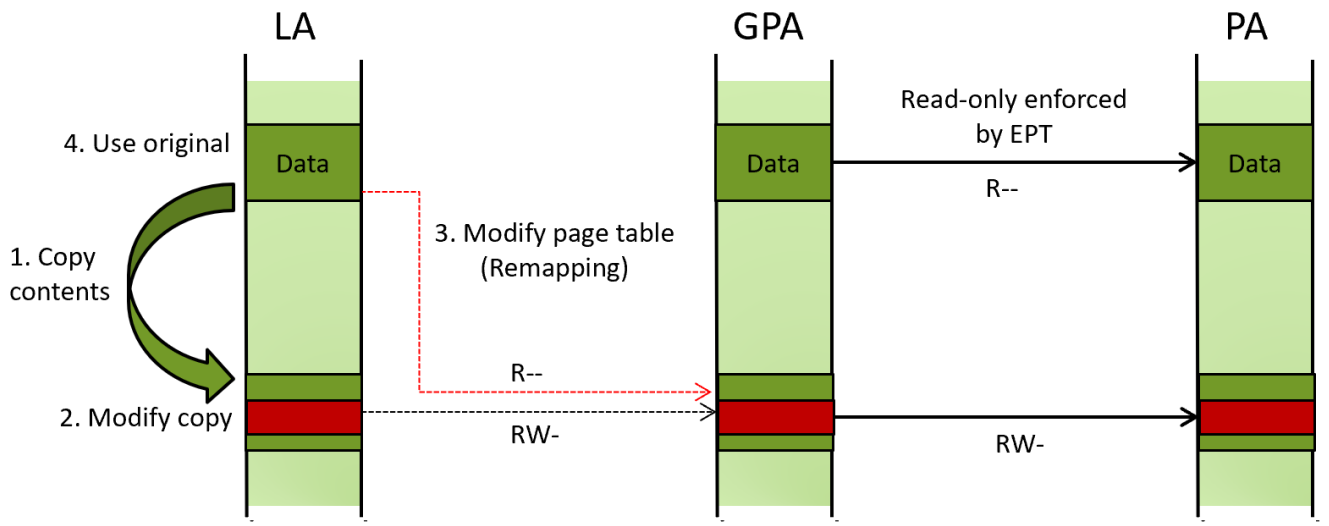
IOMMU Posted Interrupts

- Hypervisor involvement only during config



Virtualization Extension Keeps changing

- E.g., VT-rd (redirect protection)
  - Available since Alder Lake (12th gen) [curr. is Meteor Lake (14th gen)]
  - Additional protection for EPT without Memory Tracing
    - Remapping within gVA to gPA, can lead to exploit EPT hPA protections
    - E.g., we can make writable code pages in VM (allegedly protected by EPT)



https://t.ly/RJp0N