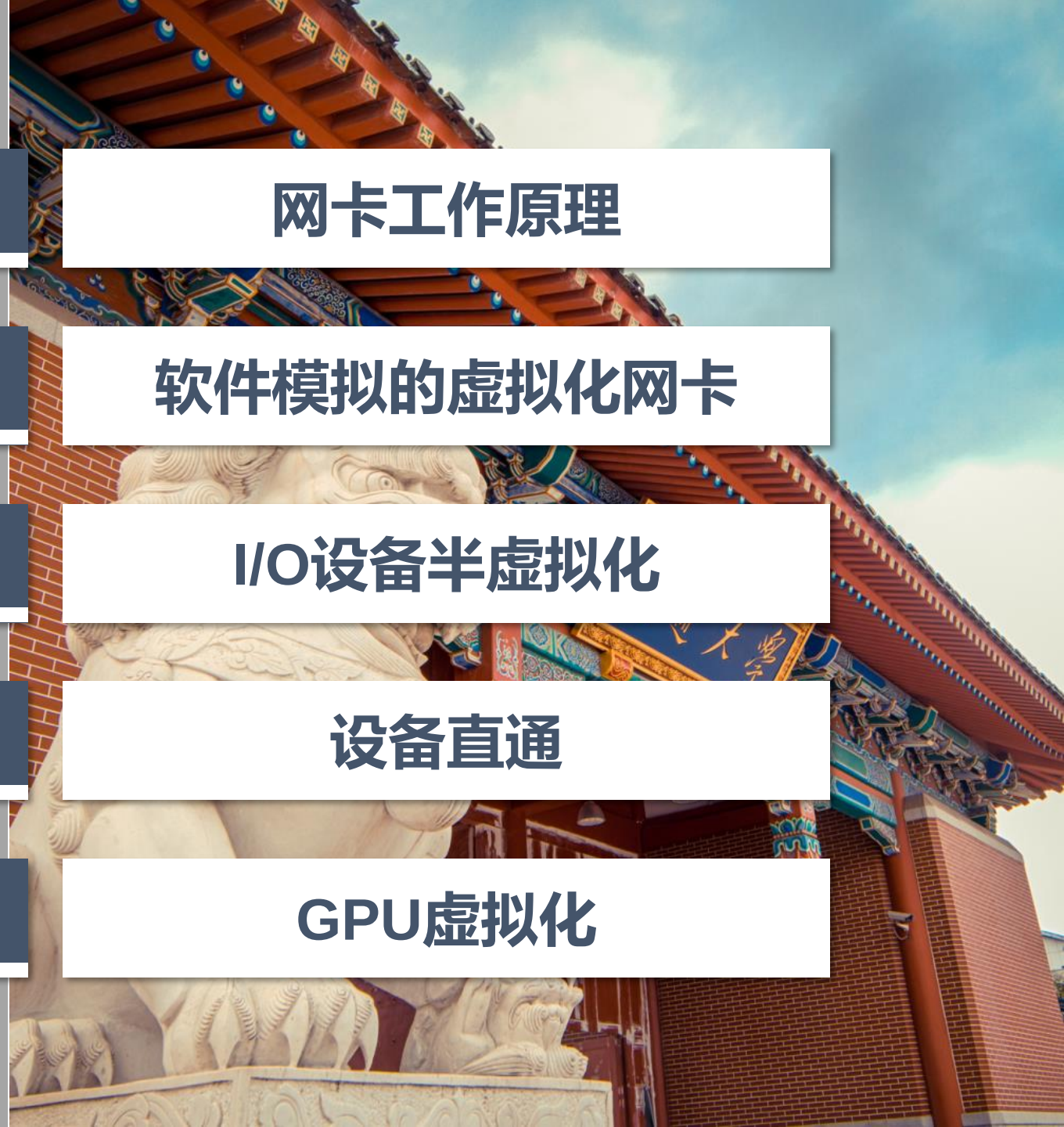
虚拟化：云计算的核心支撑技术（2）

马汝辉 副教授 博导
计算机科学与工程系
上海交通大学

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录



01

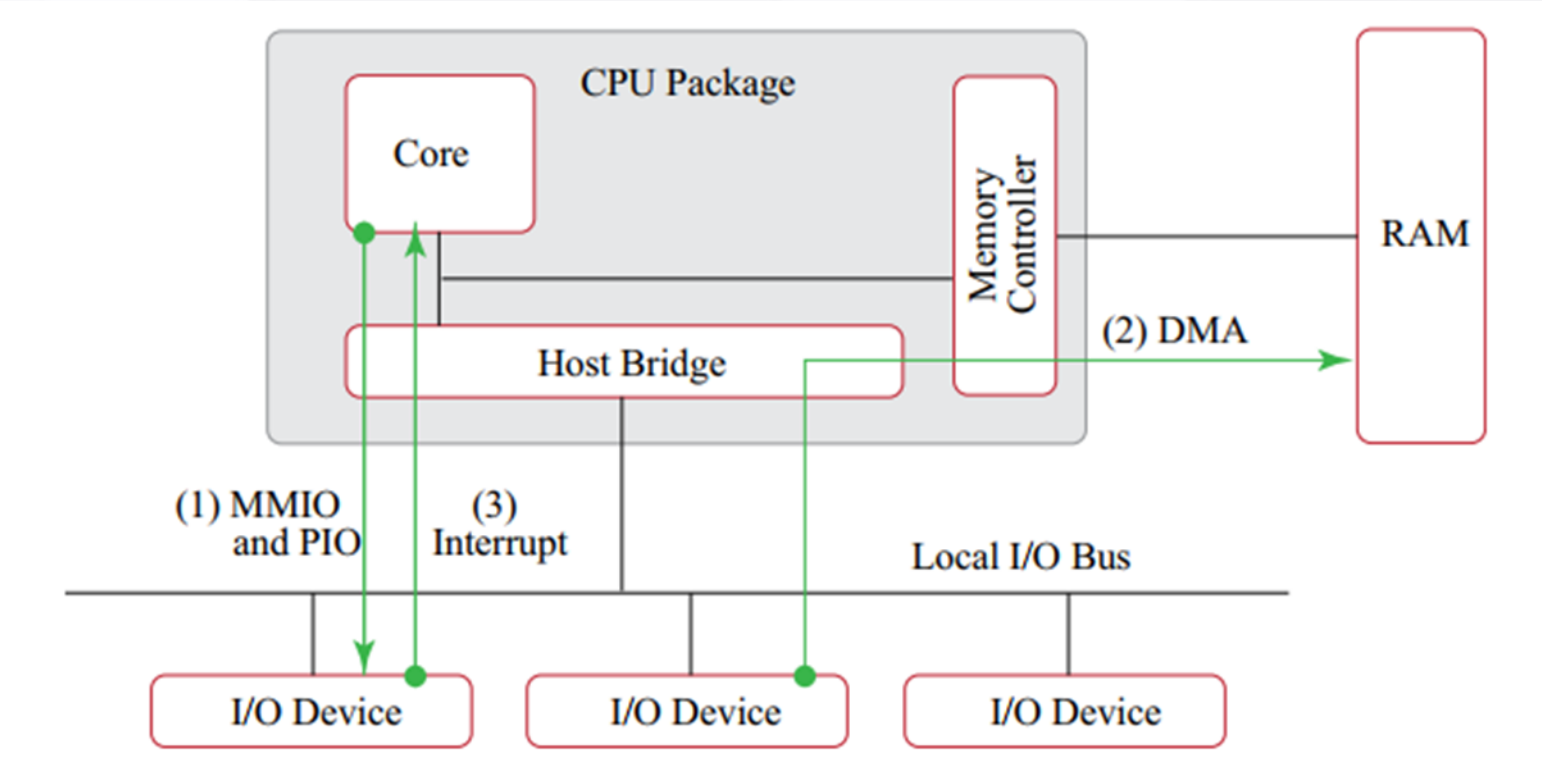
网卡工作原理

How NICs work

- Ethernet Network Interface Cards (NICs) are used to
  attach hosts to Ethernet Local Area Networks (LANs).

- NICs are deployed everywhere - laptops, PCs, machines in data centers - and many vendors and models are available - e.g. Intel, Mellanox, Broadcom, Realtek, Qualcomm.

How NICs work



Three ways that I/O devices can interact with the CPU and the memory

How NICs work

- PIO and MMIO: port-mapped I/O(PIO) and memory-mapped I/O(MMIO) are two most basic method for CPUs to interact with I/O devices. The BIOS/UEFI associate the registers of the I/O devices with unique, dedicated addresses.

- PIO: the addresses of PIO are called “ports”, they are separated from the memory address space and have their own dedicated physical bus. Such addresses are typically limited to 16 bits. They are used via the special OUT and IN x86 instructions, which write/read 1–4 bytes to/from the I/O devices.

- MMIO: the device registers are associated with physical memory addresses,
  and they are referred to using regular load and store x86 operations through the memory bus. The association between device registers and memory addresses is predetermined on startup.

How NICs work

- DMA: The core only initiates the DMA operation, asking the I/O device to asynchronously notify it when the operation completes (via an interrupt, the core is then free to engage in other work.

- Devices do DMA operation with bus addresses. In some systems, bus addresses are identical to CPU physical addresses, but in general they are not. IOMMUs and host bridges can produce arbitrary mappings between physical and bus addresses.

- Interrupts: I/O devices trigger asynchronous event notifications directed at the CPU cores by issuing interrupts.

02

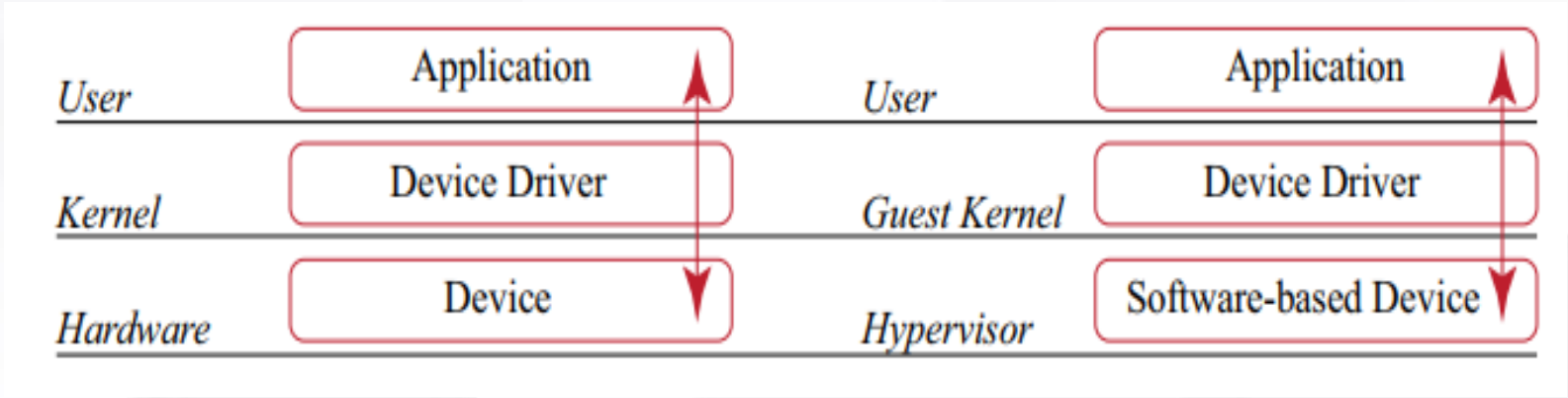
软件模拟的虚拟化网卡

SJTU

I/O Virtualization

- I/O Virtualization is an essential component in the virtualization Framework. (CPU, Memory Virtualization)

- I/O virtualization environments are created by abstracting the upper layer protocols from the physical connections.

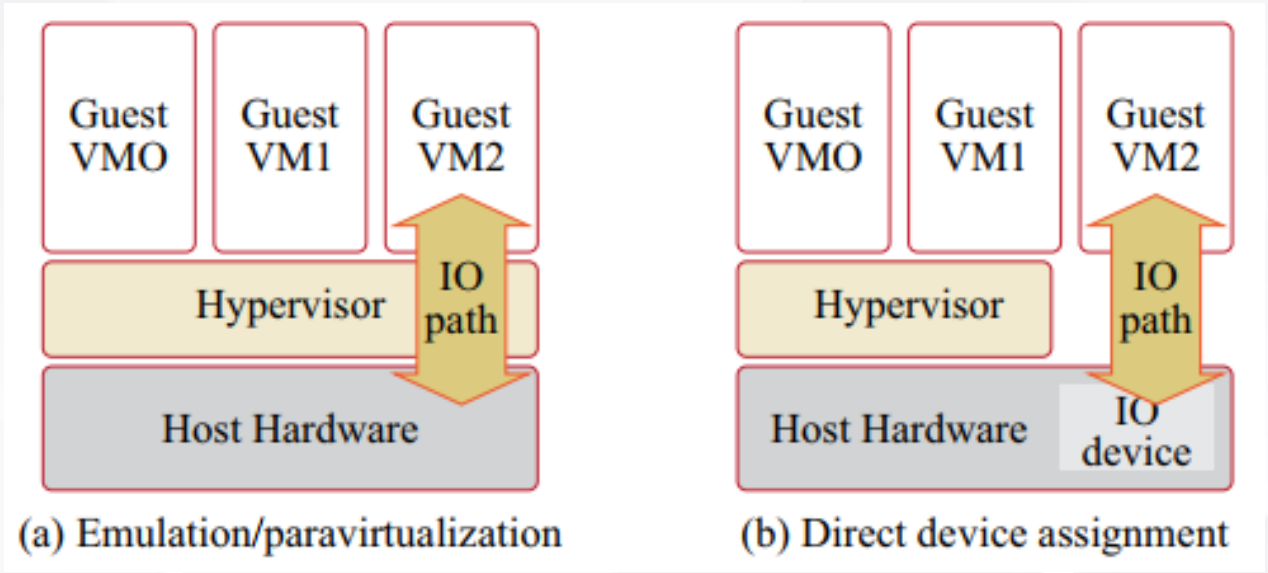


I/O Virtualization

- Software based I/O Virtualization:
  - Rich set of virtualization features
    - I/O Sharing, Security, Isolation, Mobility.
  - Simplified management
    - Encapsulate VM state->VM Suspend/Resume, Live Upgrade and Live migration.
- Hardware assisted I/O Virtualization:
  - High performance I/O: throughput, latency.
  - Forgo certain virtualization benefits.

I/O Virtualization

- Virtual I/O implementation:
  - I/O interposition(I/O emulation and Paravirtualization)
  - Direct Device Assignment(Passthrough)



How NICs are emulated

- Why devices need to be emulated in a virtualized environment ?

- A guest operating system does not cease to behave as an operating system merely because it runs as a guest. It still believes that it exclusively controls all the physical I/O devices, and it discovers, initializes, and drives these devices exactly as bare metal machine.
- Consider the system's hard drive controller. This device must be safely shared between several guest OS and the hypervisor.
- Compare RAM access with Register access.

How NICs are emulated

- How devices can be emulated in a virtualized environment ?

- I/O interposition
  - The hypervisor must prevent guests from accessing real devices while sustaining the illusion that devices can be accessed; the hypervisor must therefore “fake” I/O devices for its guests, denoted as **virtual I/O devices**.
  - The hypervisor achieves this goal by **trapping** all the guest’s I/O-related operations and by **emulating** them to achieve the desired effect.

How NICs are emulated

- **MMIO:** Guest's MMIOs are regular loads/stores from/to guest memory pages, so the hypervisor can arrange for these memory accesses to trap by mapping the pages as reserved/non-present (both loads and stores trigger exits) or as read-only (only stores trigger exits).
- **PIO:** Guest's PIOs are privileged instructions, and the hypervisor can configure the guest's VMCS to trap upon them.
- **DMA:** Emulating DMAs to/from guest memory is trivial for the hypervisor, because it can read from and write to this memory as it pleases.
- **Interrupt:** the hypervisor can use the VMCS to inject interrupts to the guest. Each interrupt causes at least two VM-Exit(direct interrupt delivery and VT-d posted interrupt).

How NICs are emulated

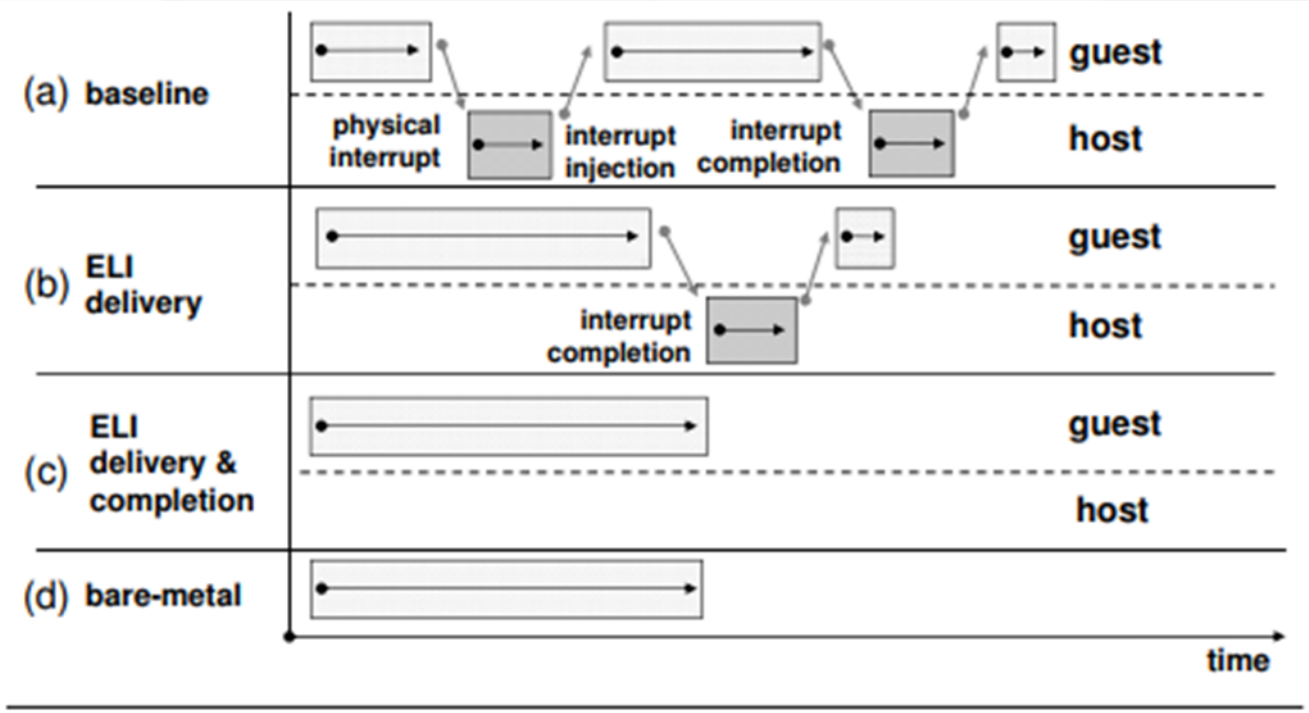
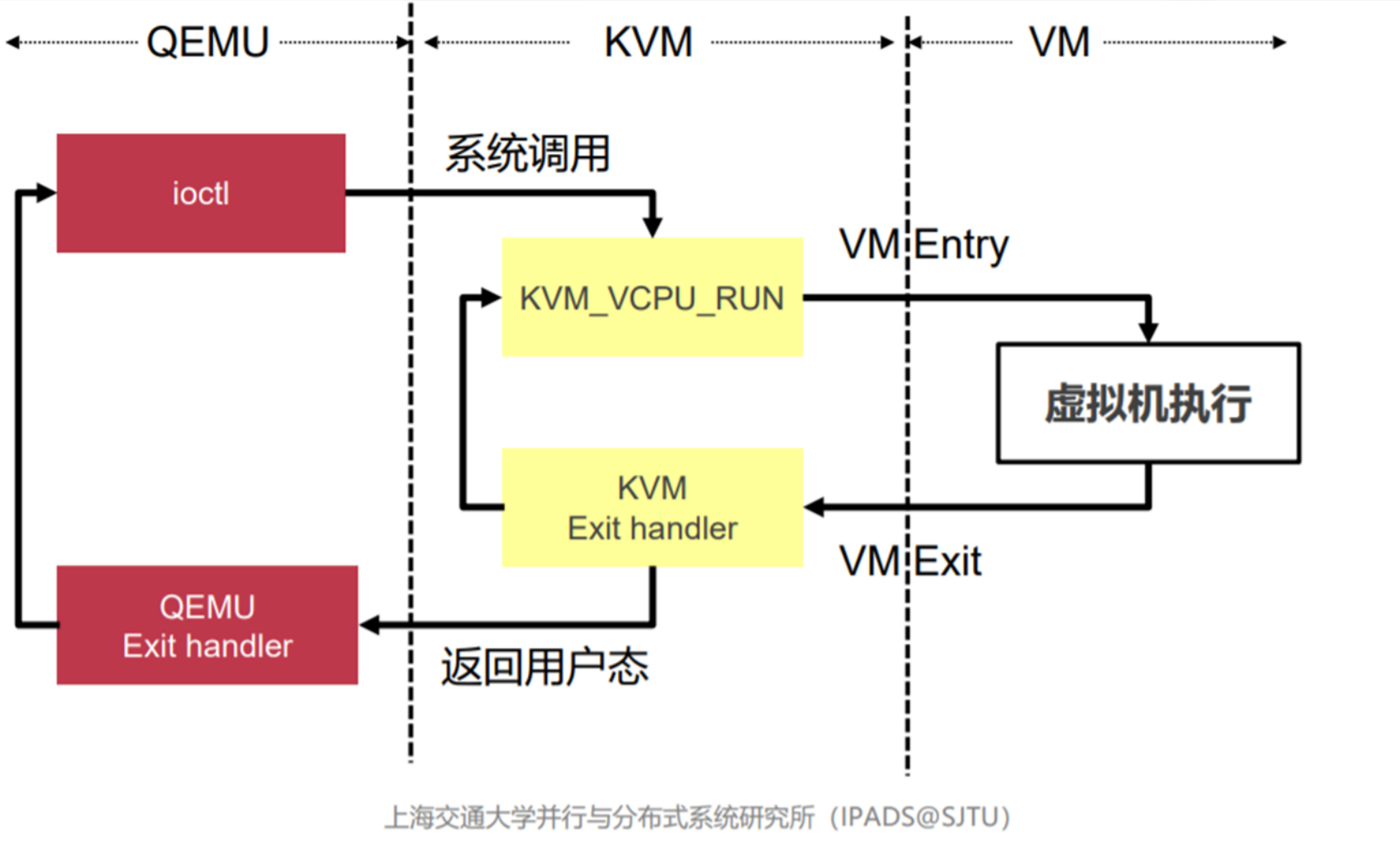


Figure 1. Exits during interrupt handling

ASPLOS'12 ELI: Bare-Metal Performance for I/O Virtualization

How NICs are emulated



```
fd = open("/dev/kvm", O_RDWR);
ioctl(fd, KVM_CREATE_VM, …);
ioctl(fd, KVM_CREATE_VCPU, …);
for(;;) {
  ioctl(fd, KVM_RUN, …);
  switch(exit_reason) {
    case EXIT_REASON_IO_INSTRUCTION: … break;
    case EXIT_REASON_TASK_SWITCH: … break;
    case EXIT_REASON_PENDING_INTERRUPT: … break;
     …
  }
}
```

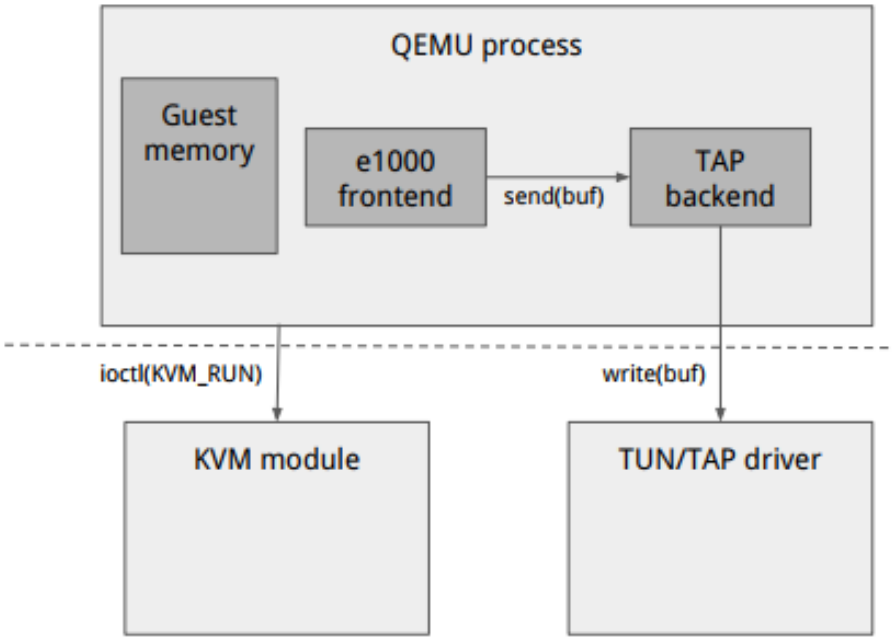
Event handling:

- timers
- I/O
- monitor commands

Event handling via select/poll system calls to wait on multiple file descriptors

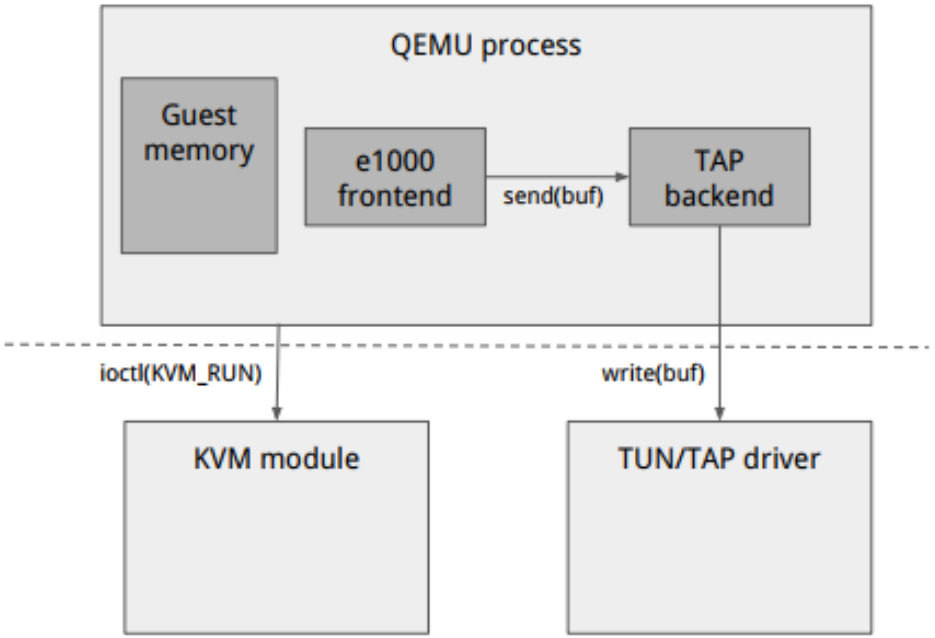
How NICs are emulated

- **Frontend:** QEMU emulation layer that understands the device’ s semantics and interacts with the driver at the guest.
- **Backend:** the software used by the frontend to implement the functionality of the virtual device using the physical resources of the host system.
- **TAP:** a virtual Ethernet network device, implemented in the kernel, that can forward Ethernet frames to and from the process that connects to it.



How NICs are emulated

- A write of guest OS to the TDT register - or to any other register - causes the CPU to exit.
- return in the KVM module (host kernel-space) that was executing an ioctl (KVM_RUN) syscall for QEMU.
- When the ioctl returns, QEMU figures out that the VM exit was due to a register access for an e1000 device and invokes the e1000 frontend.
- The e1000 handler for TDT writes collects all the produced TX descriptors. For each one, translates the guest physical address of the Ethernet frame into host virtual address, sends the frame to a network backend, writes back the descriptor (i.e. DD bit) and increments TDH.
- Once all descriptors are consumed, a Tx interrupt is injected by KVM module.

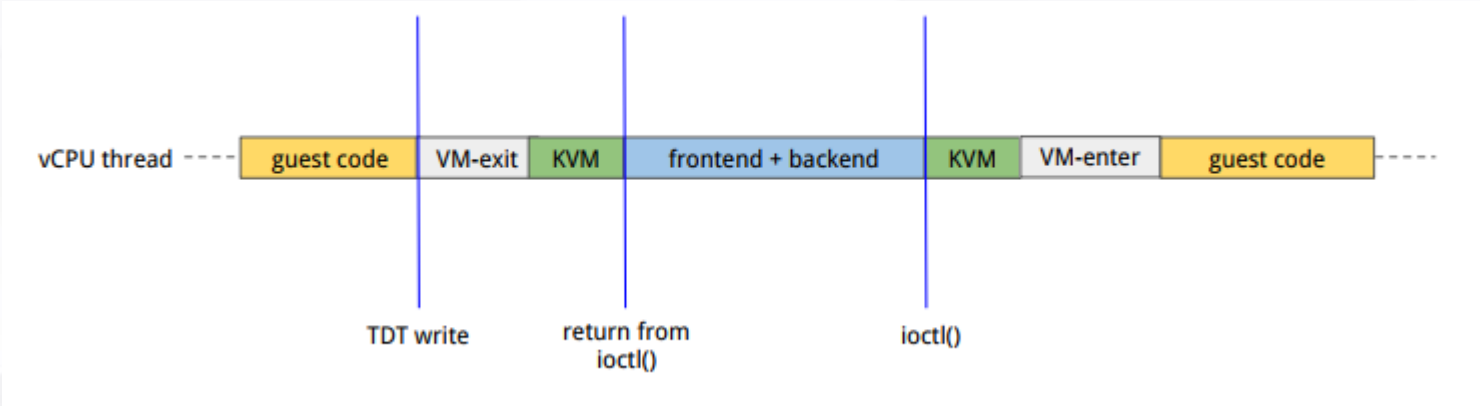


```
static void start_xmit(E1000State *s)
{
    dma_addr_t base;
    struct e1000_tx_desc desc;
    …
    while (s->mac_reg[TDH] != s->mac_reg[TDT]) {

        base = tx_desc_base(s) +
               sizeof(struct e1000_tx_desc) * s->mac_reg[TDH];

        pci_dma_read(d, base, &desc, sizeof(desc));

        process_tx_desc(s, &desc); //set DD bit, send packet to TAP

        if (++s->mac_reg[TDH] * sizeof(desc) >= s->mac_reg[TDLEN])
            s->mac_reg[TDH] = 0; // advance TDH
        …
    }
    …
}
```

QEMU: /hw/net/e1000.c

How NICs are emulated

- The e1000 emulated TX processing is completely synchronous, differently from what happens with a real e1000 NIC, where the NIC hardware runs in parallel to the CPU running the sending application and the driver code.
- Each notification causes one VM-exit.
- Each interrupt causes at least two VM-exits. Interrupt Injection and completion EOI write.



03

I/O设备半虚拟化

SJTU

I/O Emulation

- Guest OS believes exclusive control on I/O devices.
- Hypervisor traps the I/O related operations and emulates them
- Strengths:
  - Full virtualization, no modification to the guest OS.
  - Software device states are easy to encapsulate->flexibility:
    - dynamically decouple/recouple a physical device from/to a VM
    - Live upgrading, reconfiguring, live migration.
  - I/O consolidation, improving efficiency, reducing cost.
  - I/O aggregation: better performance and robustness.
- Weaknesses:
  - Efficient virtualization was not considered for hardware design.
  - Substantial performance overheads.

I/O paravirtualization ideas

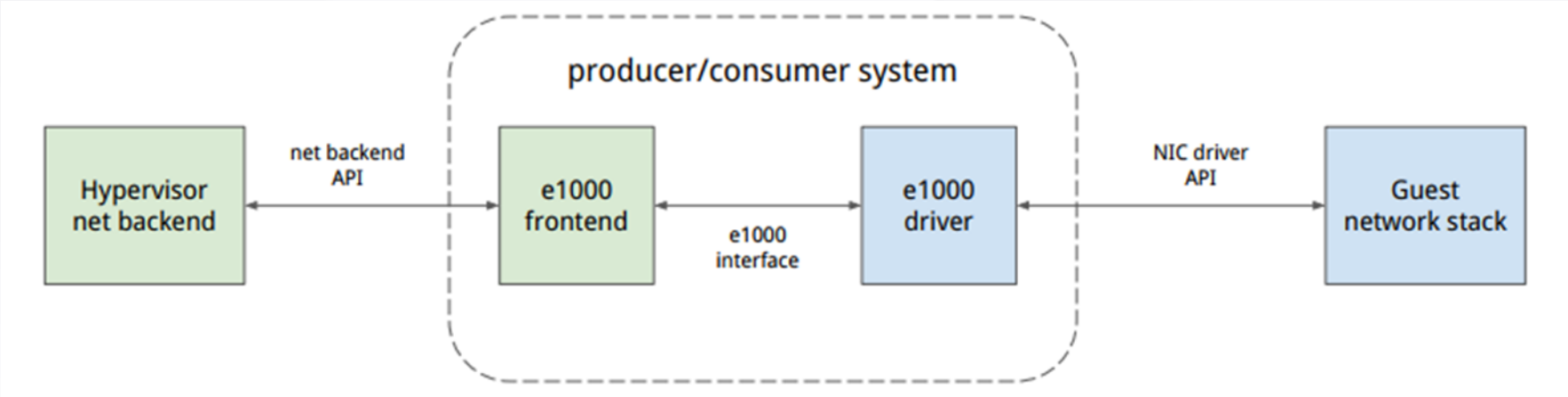
- While I/O emulation implements a correct behavior, it might induce **substantial performance overheads**, because efficient emulation was not recognized as a desirable feature when the physical device was designed.

- Virtualization overheads caused by inefficient interfaces of physical devices could, in principle, be eliminated, if we **redesign the devices** to have **virtualization-friendlier interfaces**.

- I/O paravirtualization: guests and hosts agree upon a (virtual) device specification to be used for I/O emulation, with the explicit goal of **minimizing overheads**.

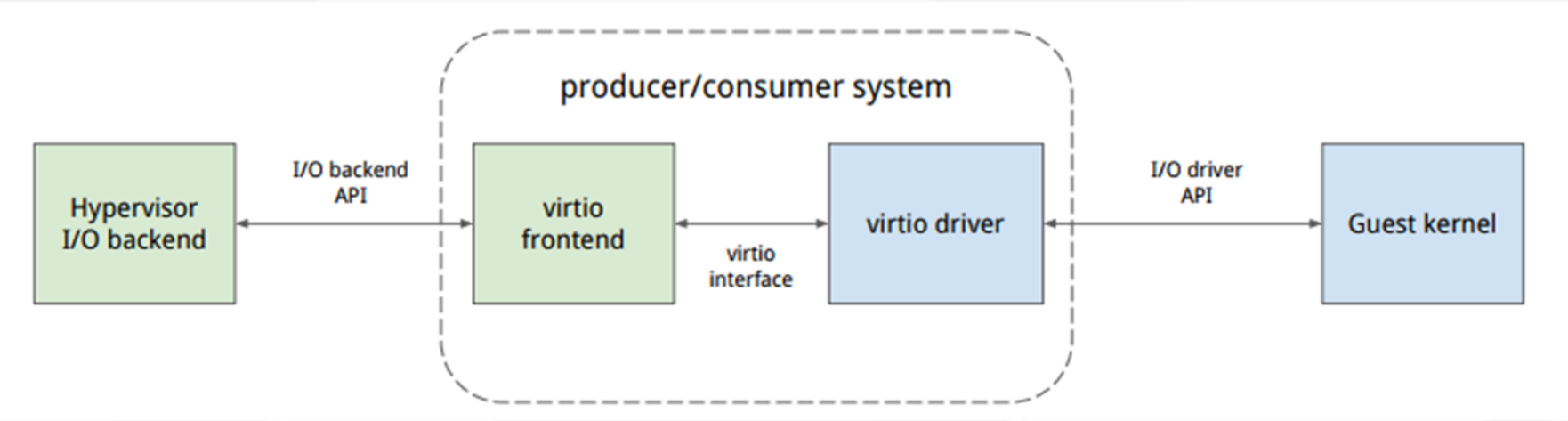
I/O paravirtualization ideas

- Is it possible to build a better - simpler and more efficient - producer/consumer system that matches the same interfaces as the composite e1000 driver + frontend?



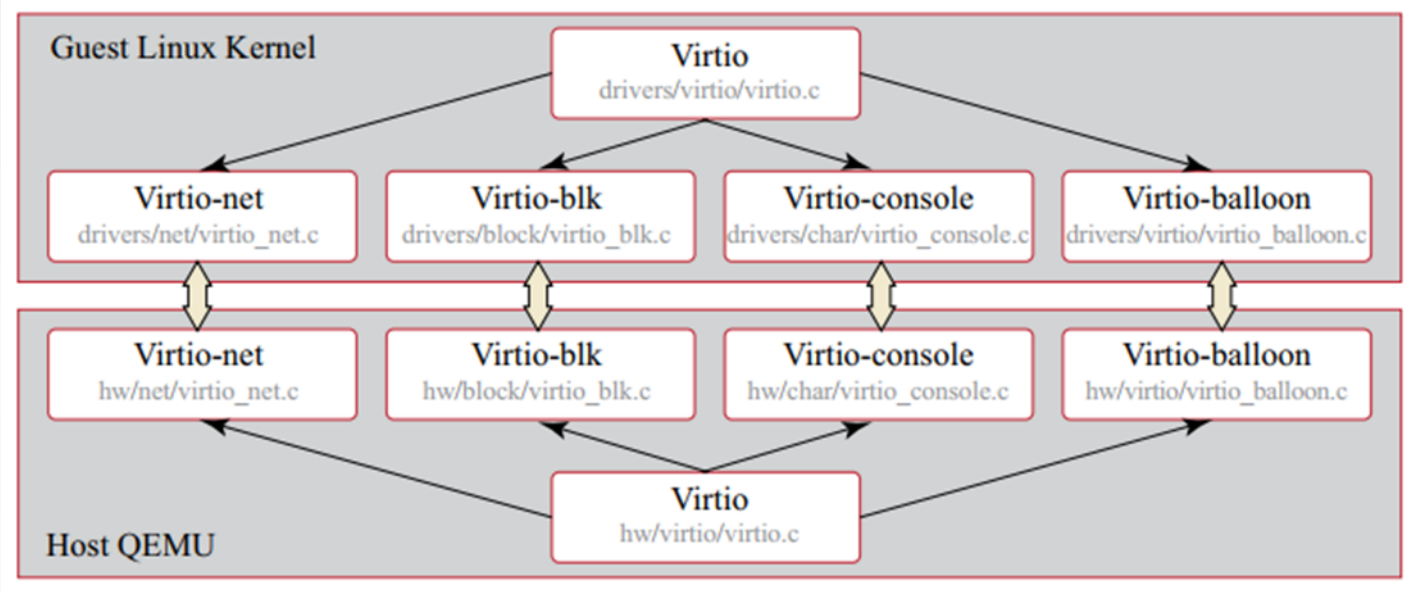
The VirtIO Standard

- The framework of paravirtual I/O devices of KVM/QEMU is called VirtIO, offering a common guest-host interface and communication mechanism.



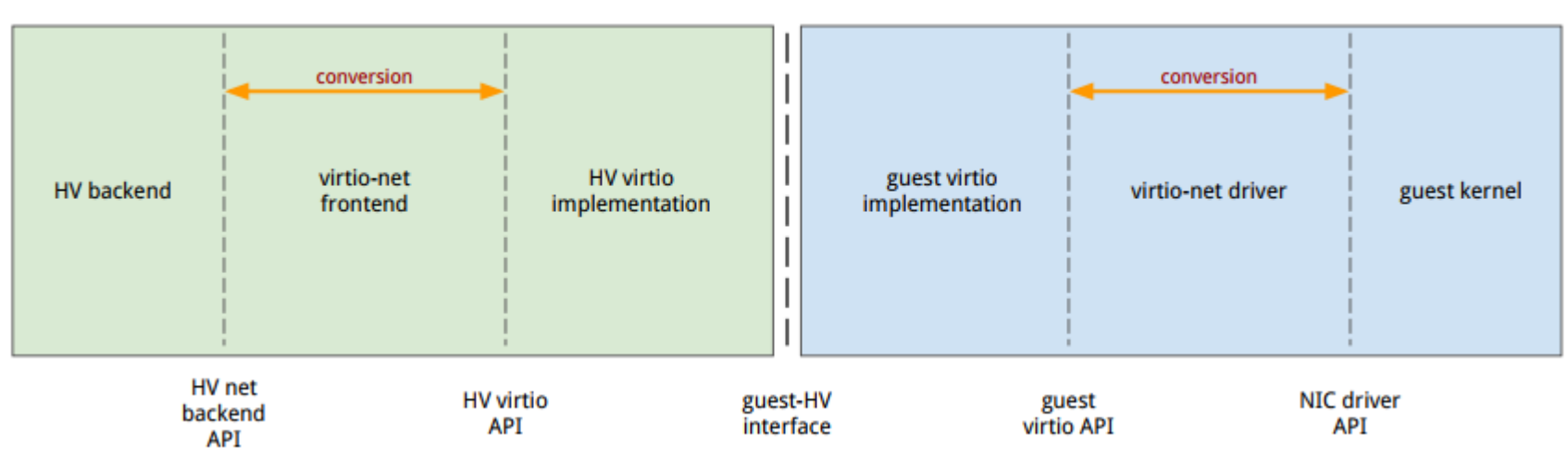
The VirtIO Standard

- The same principles can be applied also to other forms of virtualized I/O (block storage, serial ports, …), since all forms of I/O can be seen as producer/consumer systems that exchange messages.



The VirtIO Standard

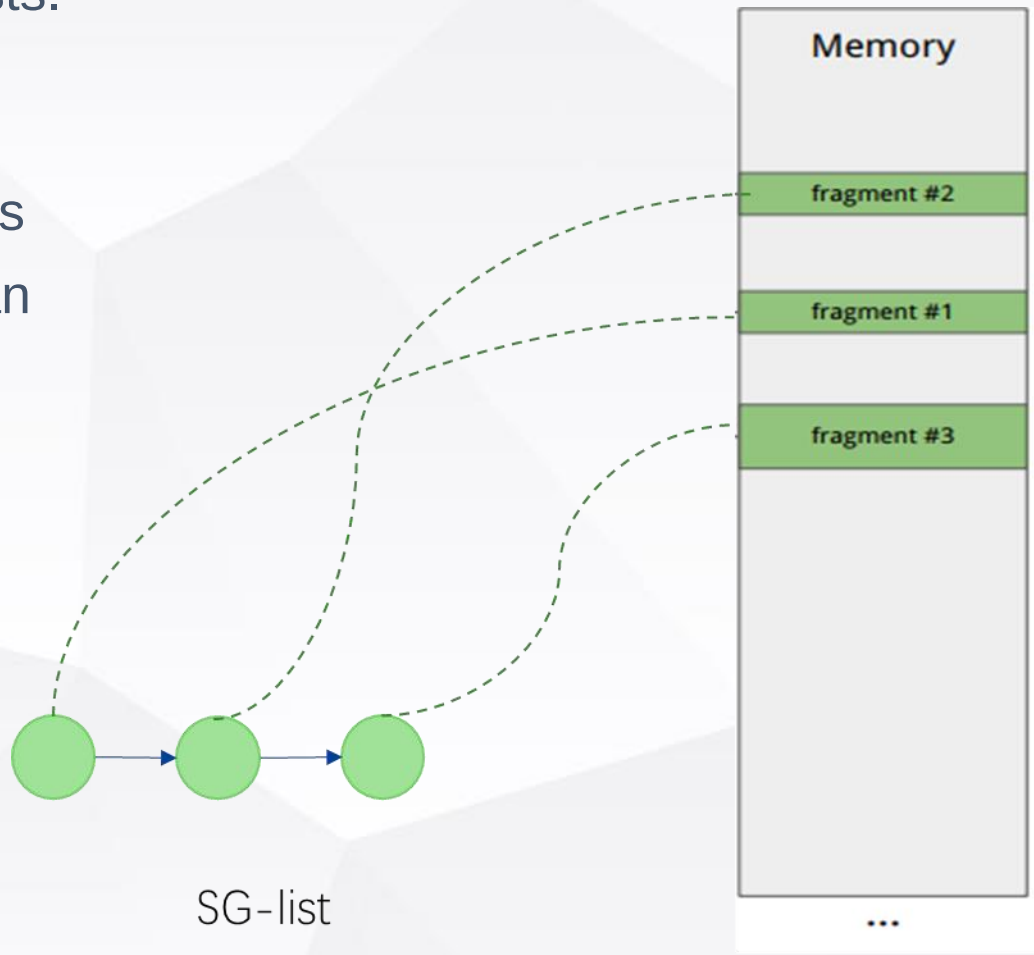
- The task of a VirtIO driver is to convert the OS-specific representation of the message (e.g. a skb object for a Linux network driver) to the VirtIO message format, and the other way around.
- The virtio-net frontend performs the same task on the hypervisor side, converting VirtIO messages from/to formats understandable to the backend.



Scatter-Gather list

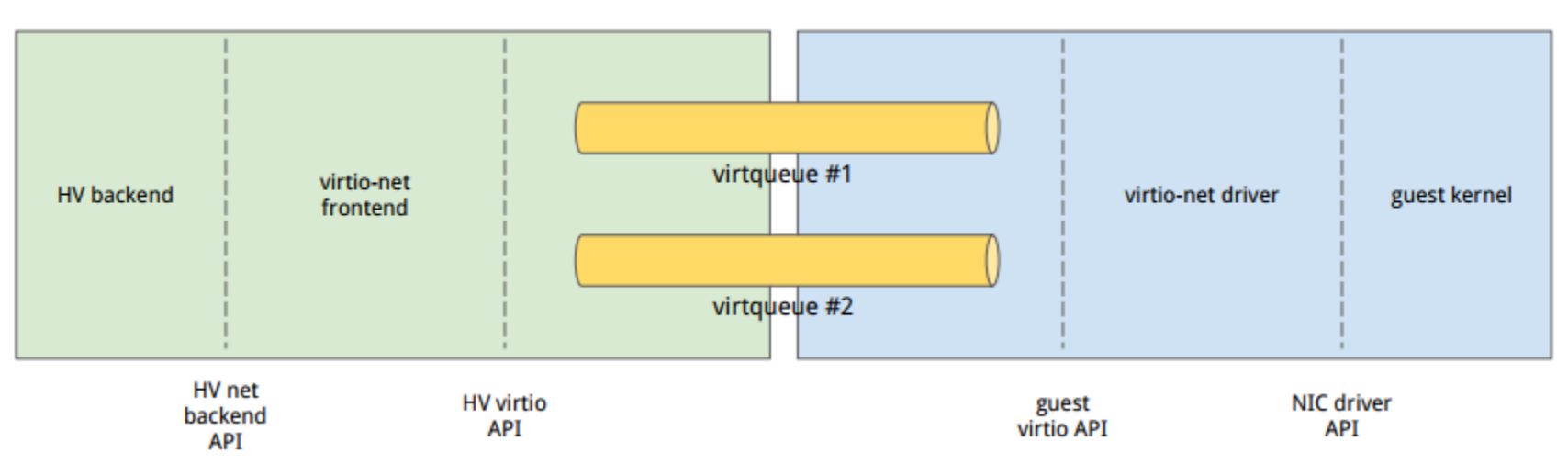
- VirtIO exchanges data using Scatter-Gather (SG) lists.

- An SG list is conceptually a list of (physical) address and length couples and is usually implemented as an array.



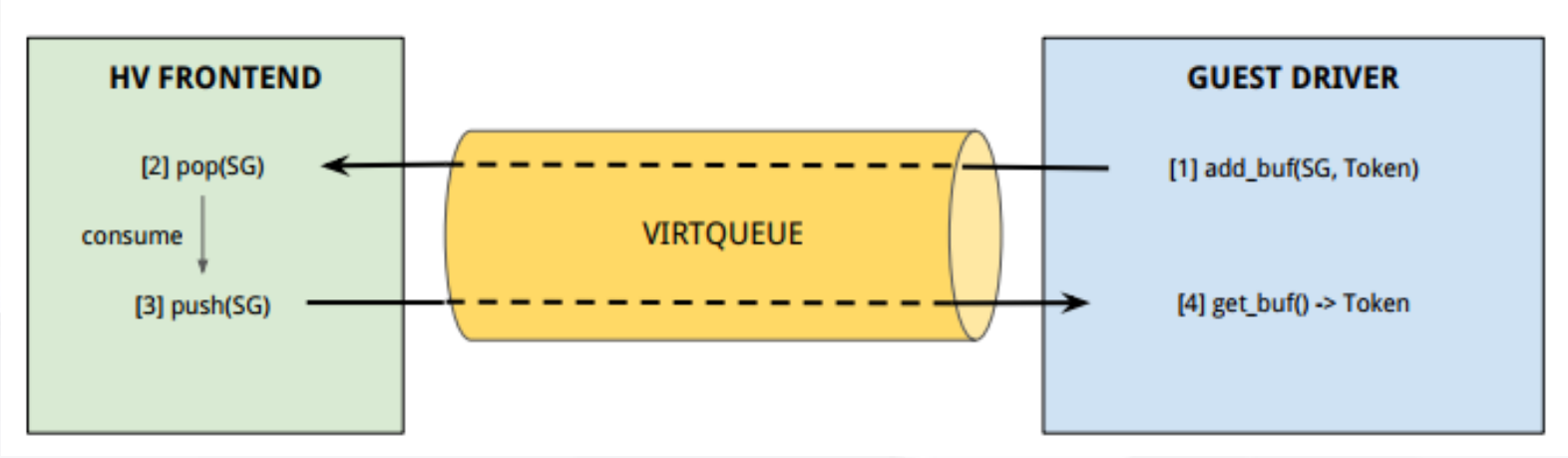
Virtqueues

- Central to VirtIO is the Virtqueue (VQ) abstraction. A VQ is a queue where SGs are posted by the guest driver to be consumed by the hypervisor. Output SGs are used to send data to the hypervisor, while input SGs are used to receive data from the hypervisor.
- A device can use one or more queues, and the number of queues may be negotiated.

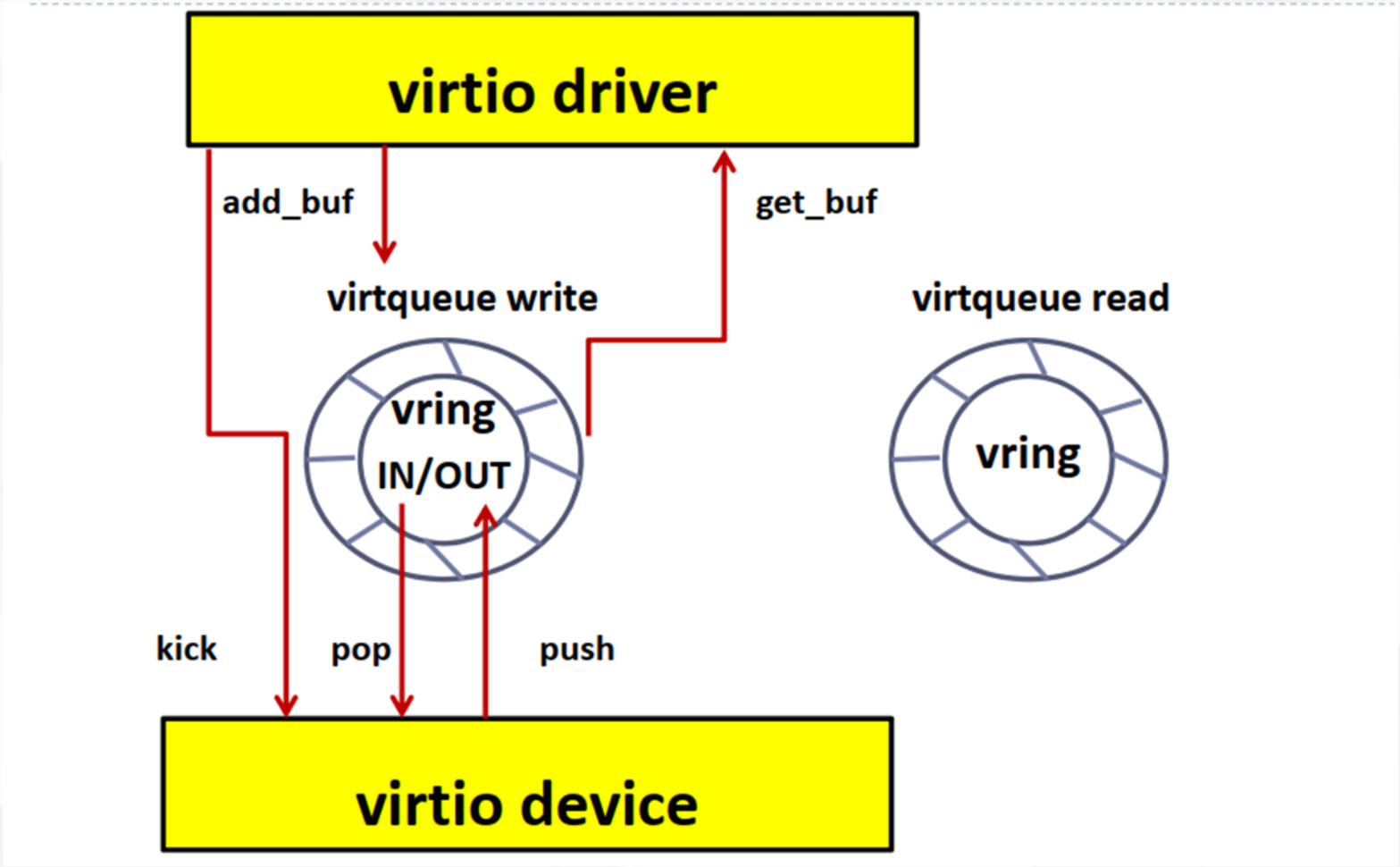


Virtqueues

- When a guest driver wants to produce a SG, it calls the add_buf VQ method, also passing a token. On the other side, the HV pops the SG, consumes it (interacting with a backend), and pushes it back in the VQ.
- The guest polls for used SGs by calling the get_buf VQ method, so that can perform cleanup operations. The token - which is opaque for the VQ and is not passed to the HV - can be used by the driver to match produced SGs (requests) against consumed ones (responses). Tokens allow for out-of-order SG consumption (e.g. useful with block I/O).

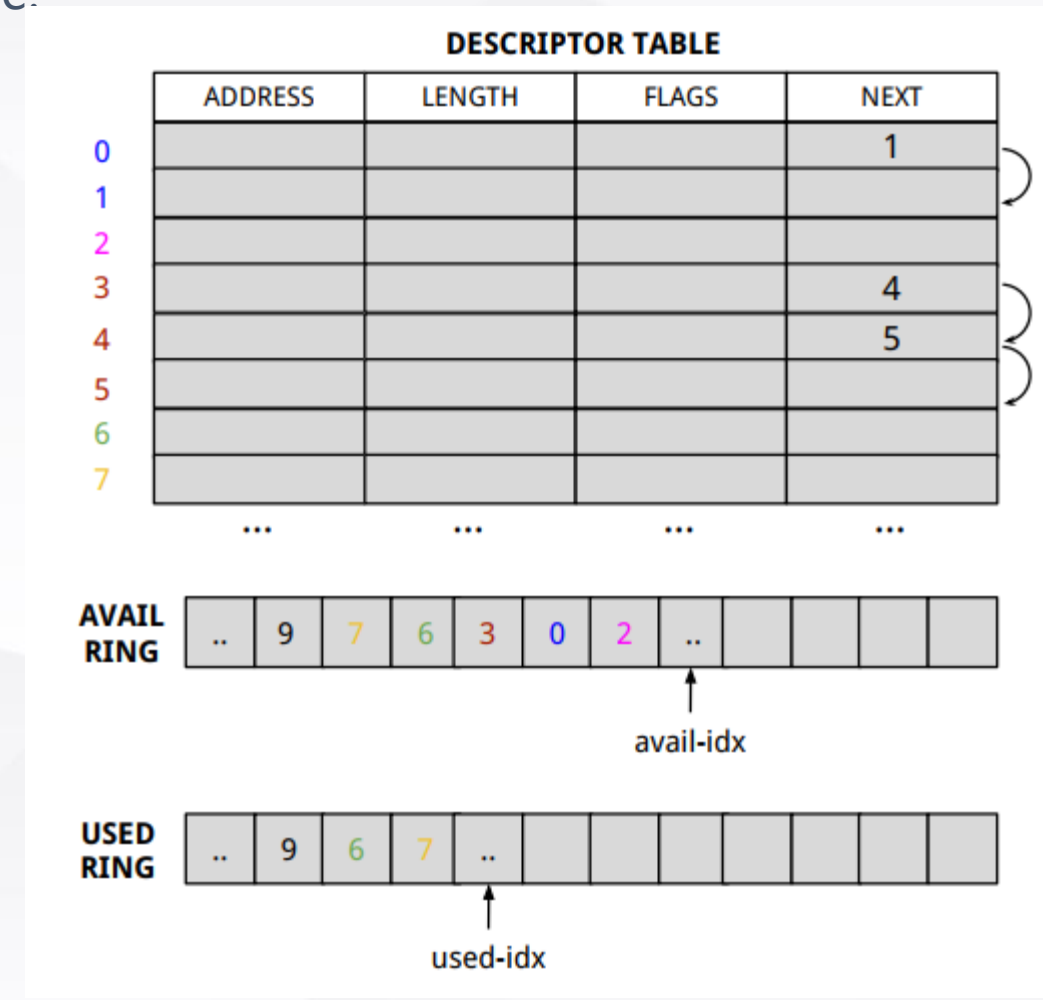


Virtqueues


virtio driver
add_buf
get_buf
virtqueue write
virtqueue read
vring
IN/OUT
vring
kick
pop
push
virtio device

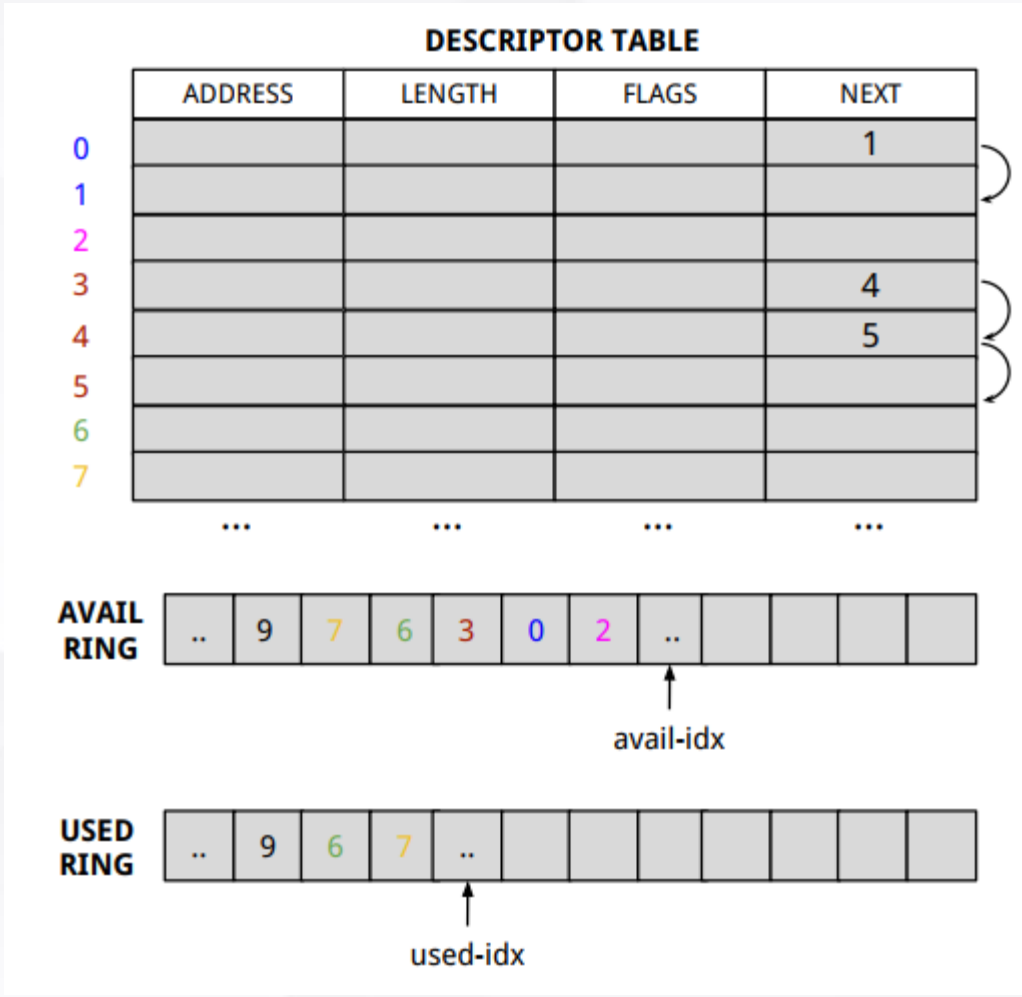
The VRing implementation

- Three data structures used to describe one VirtQueue:
  - Descriptor Table
  - Avail Ring
  - Used Ring
- Multi-fragments SG
- Out-of-order SG consumption
- Optimized cache usage.



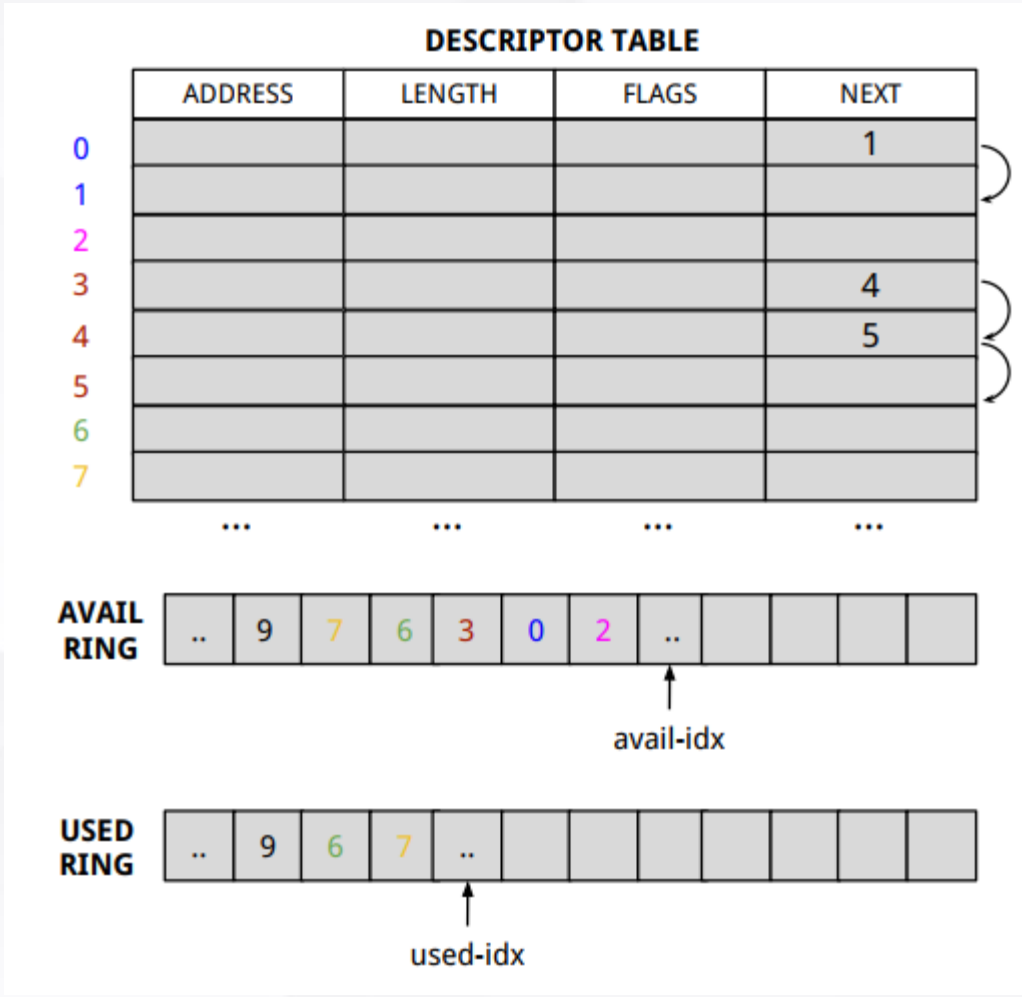
The VRing implementation

- Descriptor Table
  - Physical address and length of a buffer.
  - Next pointer for chained descriptors.
  - Flags(Input or Output).
- Write only by Guest.
- An SG with N fragments is mapped to N descs.



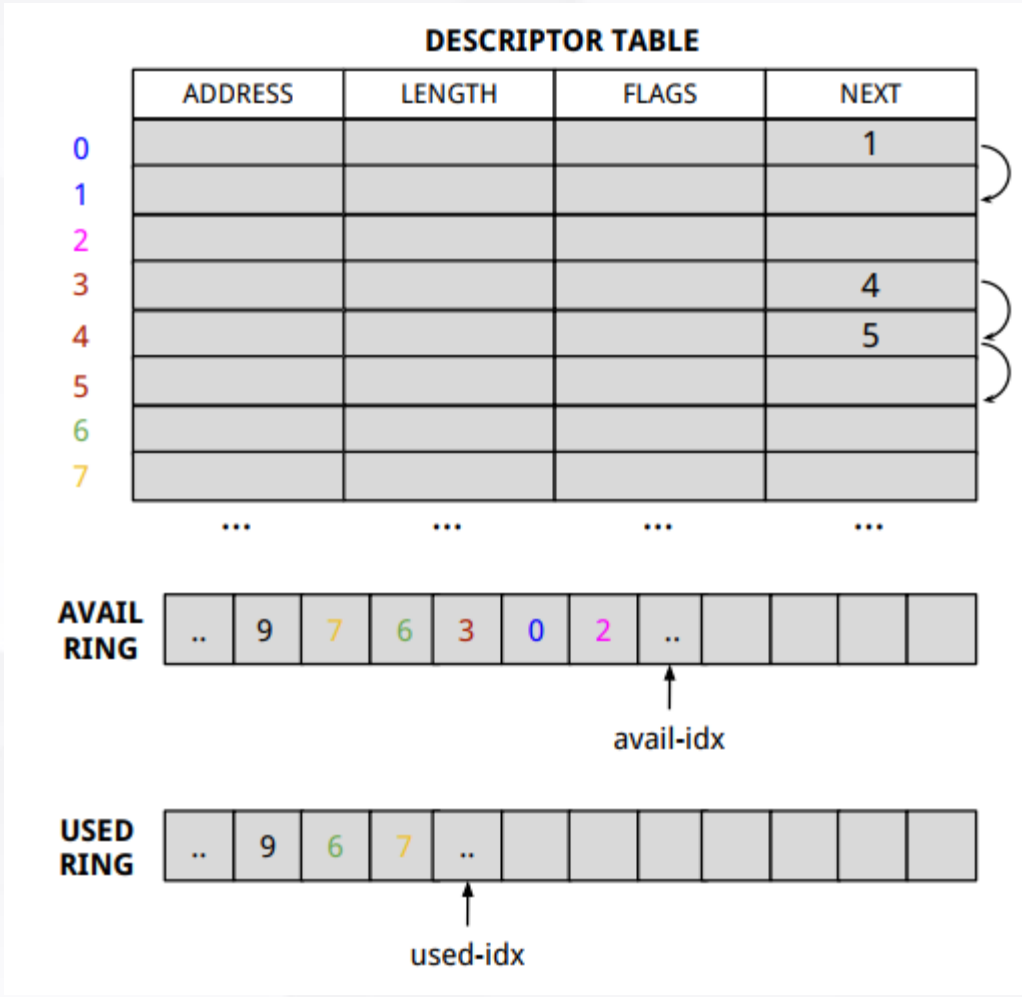
The VRing implementation

- Used Ring
  - Used by the hypervisor to return consumed SGs to the guest.
  - Each slot contains the head of the consumed desc.
  - Used-idx like the head register of e1000 ring but different. It is stored in memory, not a MMIO register.



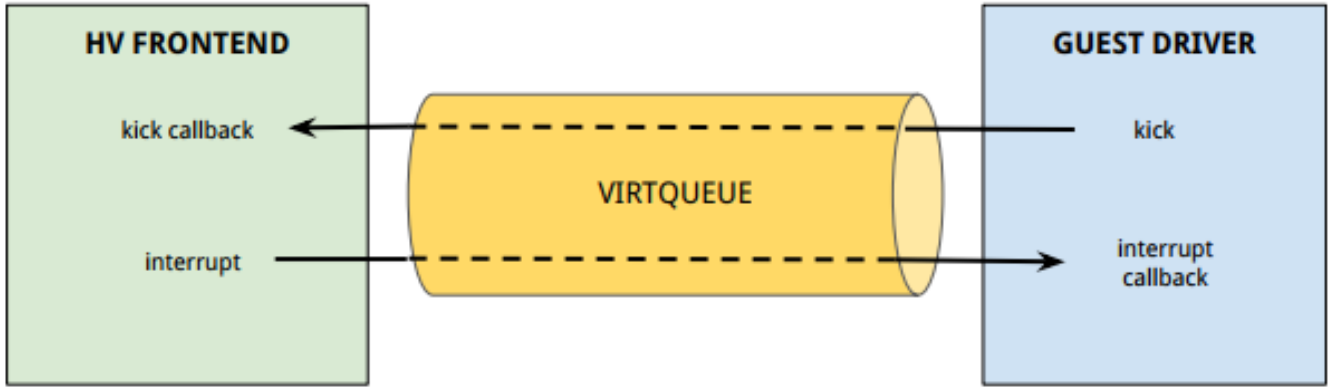
The VRing implementation

- Avail Ring
  - Used by the guest to expose produced SGs.
  - Each slot contains a head which is an index in the desc table-to the first desc of a SG.
  - Avail-idx like the tail register of e1000 ring but different. It is stored in memory, not a MMIO register.
- Decouple counting of available SGs from notification.



Minimizing notifications

- When the driver wants the HV to start consuming SGs, it notifies by using a **VQ kick** (implemented with a register write). Similarly, when the HV wants to notify the driver about consumed SGs, it uses a **VQ interrupt**.
- The kick and interrupt operations are part of the VirtIO interface. The driver should produce as many SGs as possible before kicking (principle E). Similarly, the HV should consume as many SGs as possible before sending an interrupt. In this way, notification costs are amortized over many SGs.

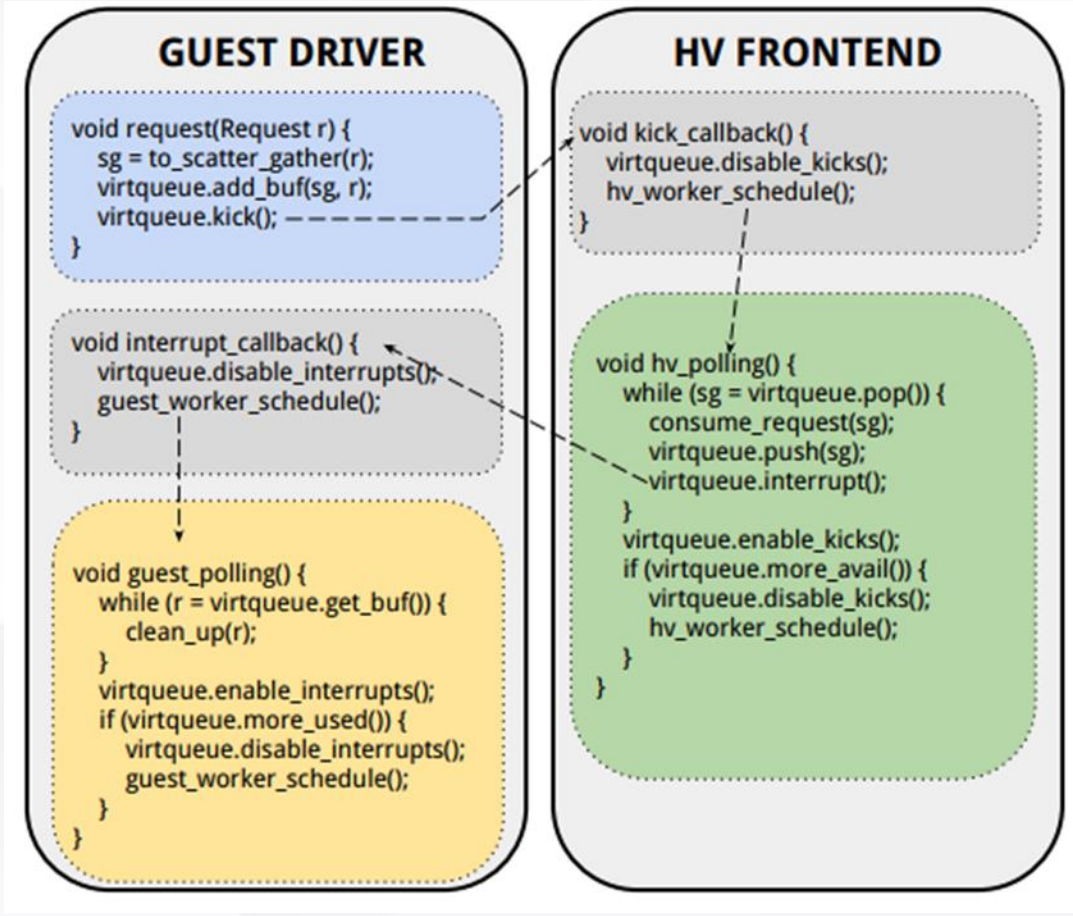


Minimizing notifications

- Key observations:
  - Once the HV consumer thread has been woken up, VQ kicks can be disabled to temporarily switch to polled mode. When all the pending SGs have been consumed, VQ kicks can be enabled again.
  - Once the guest cleanup thread has been woken up, VQ interrupts can be disabled to temporarily switch to polled mode. When all the consumed SGs have been cleaned up, VQ interrupts can be enabled again.
- This strategy (similar to Linux NAPI) allows both producer and consumer to temporarily switch from interrupt mode to polled mode. In this way, notifications (kicks and interrupts) may be amortized over many packets.

Minimizing notifications

- The blue thread produces requests in parallel to the green thread consuming them.
- The green thread returns consumed requests in parallel to the yellow thread cleaning them up.
- While the threads run, notifications are disabled.
- This scheme is general and can be applied to any paravirtualized I/O device.



Performance Test

	Metric	e1000	Virtio-net	Ratio
Guest	throughput (Mbps)	239	5,230	22x
	exits per second	33,783	1,126	1/30x
	interrupts per second	3,667	257	1/14x
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
Ethernet frames	per second	23,804	–	–
	avg. size (bytes)	1,259	–	–

Netperf TCP stream running in a Linux 3.13 VM on top of Linux/KVM (same version) and QEMU 2.2, equipped with e1000 or virtio-net NICs, on a Dell PowerEdge R610 host with a 2.40GHz Xeon E5620 CPU.

I/O paravirtualization

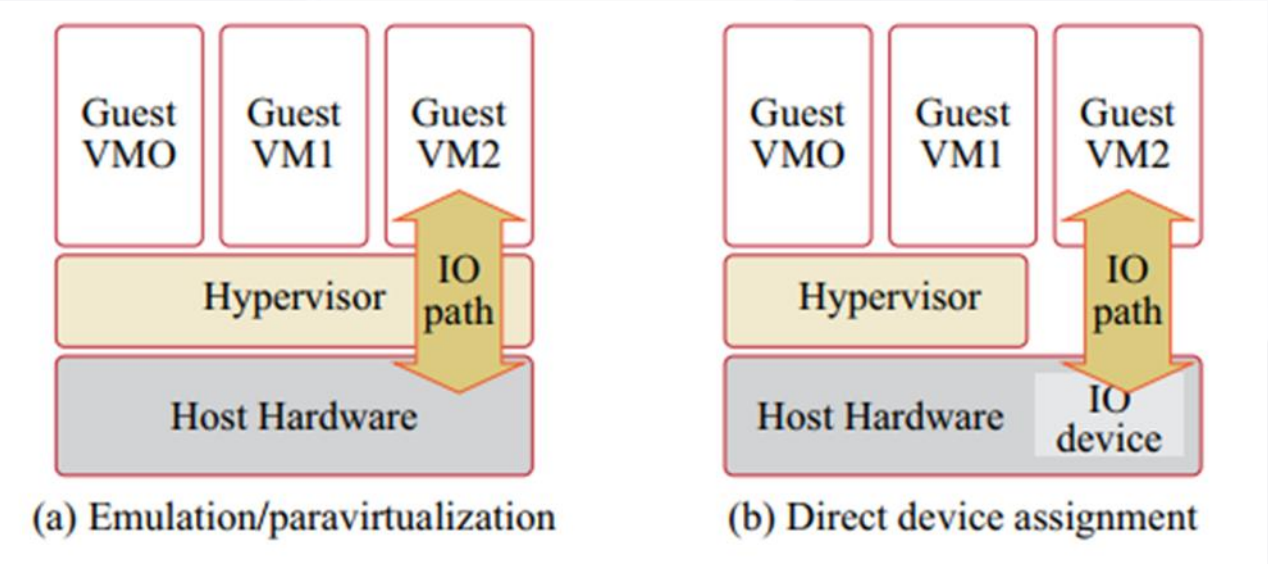
- Redesign virtual device and its interface.
- Guest uses specialized driver with minimal overhead associated with emulation.
- Strengths:
  - Most virtualization benefits.
  - Much better performance than I/O emulation.
- Weaknesses:
  - Performance gap with bare-metal machines.(Especially large packets)
  - Modified guest OS make it less portable than full virtualization.
    - Installation of paravirtual drivers.
    - Implementation for each type of OS.

04

设备直通

Direct Device Assignment

- Device emulation and paravirtualization both incur virtualization overheads but provide safe virtual I/O and the benefits of I/O interposition.
- Given a physical device d, the hypervisor may decide to assign the right to access d exclusively to some specific virtual machine v, such that no other VM, and not even the hypervisor, are allowed access. This approach, is denoted as **direct device assignment**.

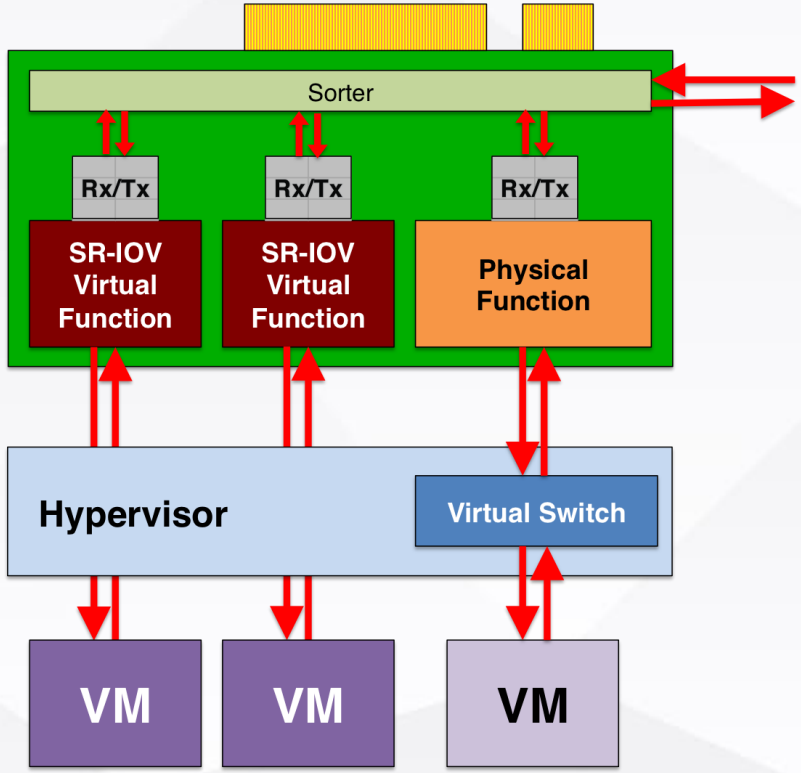


Direct Device Assignment

- Problems of Direct Device Assignment
- Lack of scalability
  - Extra physical I/O devices are inherently limited in quantity. The number of virtual machines that a modern server can support is much larger than the number of physical I/O devices that it can house.
- Isolation and security
  - If v controls a device, then v can program the device to perform DMA operations directed at *any* physical memory location. In other words, by assigning d to v, we allow v to (indirectly) access the entire physical memory, including areas belonging to other VMs or the hypervisor. Therefore, by utilizing naive direct device assignment, we essentially eliminate isolation between VMs.

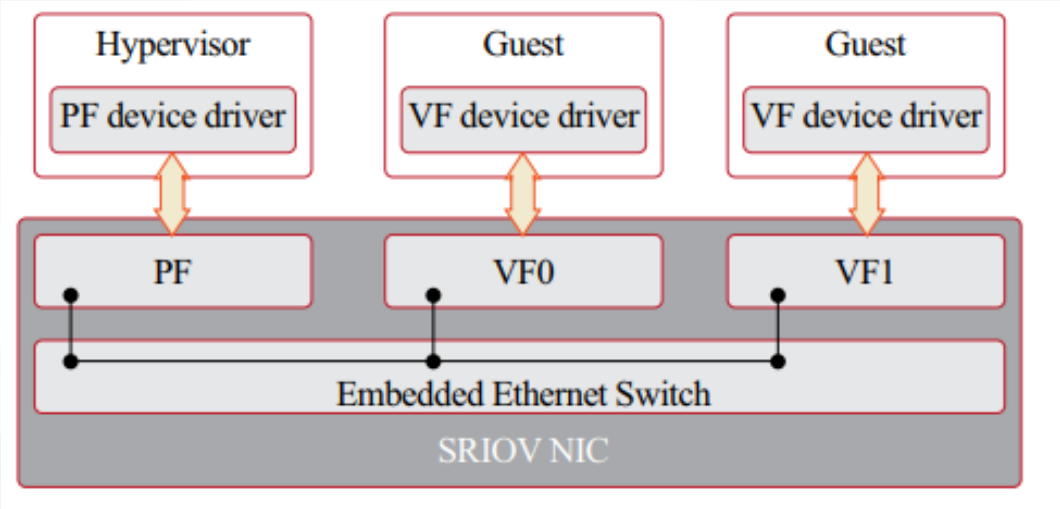
Single Root I/O Virtualization

- The SR-IOV specification from PCI-SIG defines the extensions to the PCI Express (PCIe) specification suite that enable multiple virtual machines (VMs) to share the same PCIe physical hardware resources.
- One physical function(PF) ->multiple virtual function(VF), each with own DMA streams, interrupts.

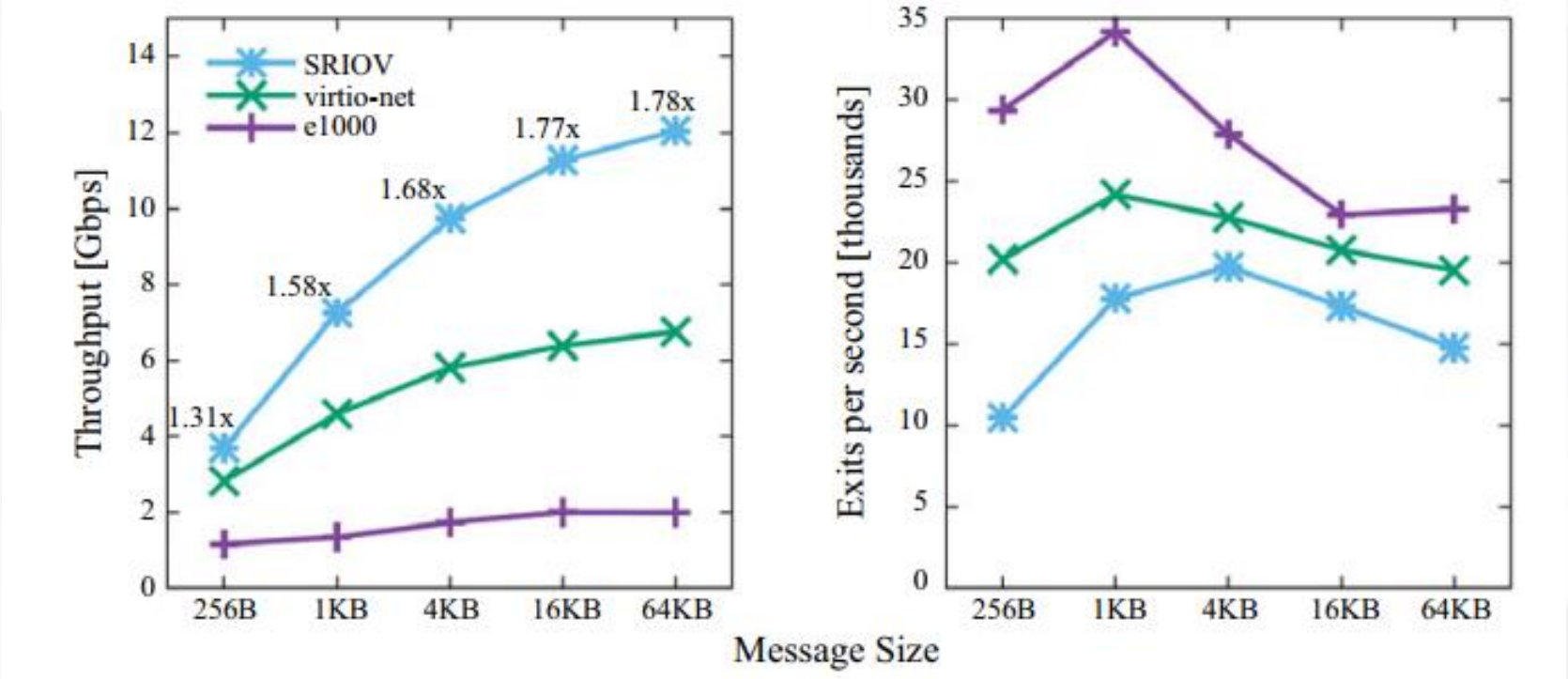


Single Root I/O Virtualization

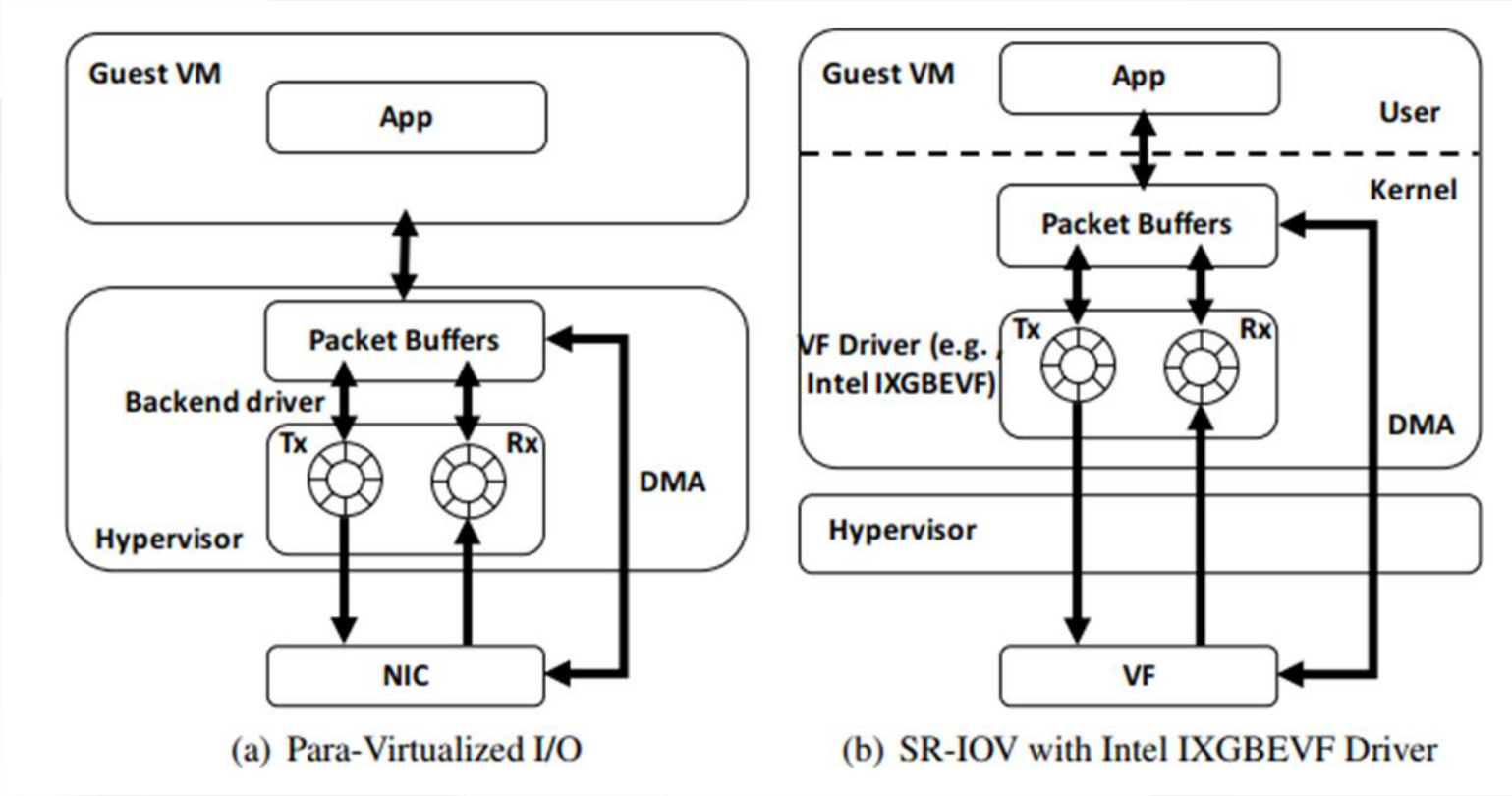
- An SRIOV-capable I/O device can present multiple instances of itself to software. Each instance can then be assignment to a different VM, to be used directly (and exclusively) by that VM, without any software intermediary. Traditionally, it is the role of the operating system to multiplex the I/O devices. Conversely, an SRIOV device knows how to multiplex itself at the hardware level.



Single Root I/O Virtualization



Single Root I/O Virtualization



Summary

- Strengths:
  - Self-virtualized hardware, no emulation overhead
  - High performance: high throughput, low latency
  - Mitigate Scalability Problem(Intel 710 series 1 PF supports 64 VF)
- Weaknesses:
  - Less flexibility:
  - Memory optimization disabled.
  - Live migration problems:
    - Dirty page loss during pre-copy iteration
    - Device state migration

Summary

- Emulation (Full Virtualization)
  - Best option for correctness and abstraction
  - Heavy performance overhead
- Paravirtualization
  - Optimize driver and virtual device interaction
  - Guest is “aware” of virtualization
- Direct Device Assignment
  - Best option for performance
  - Strong coupling with hardware

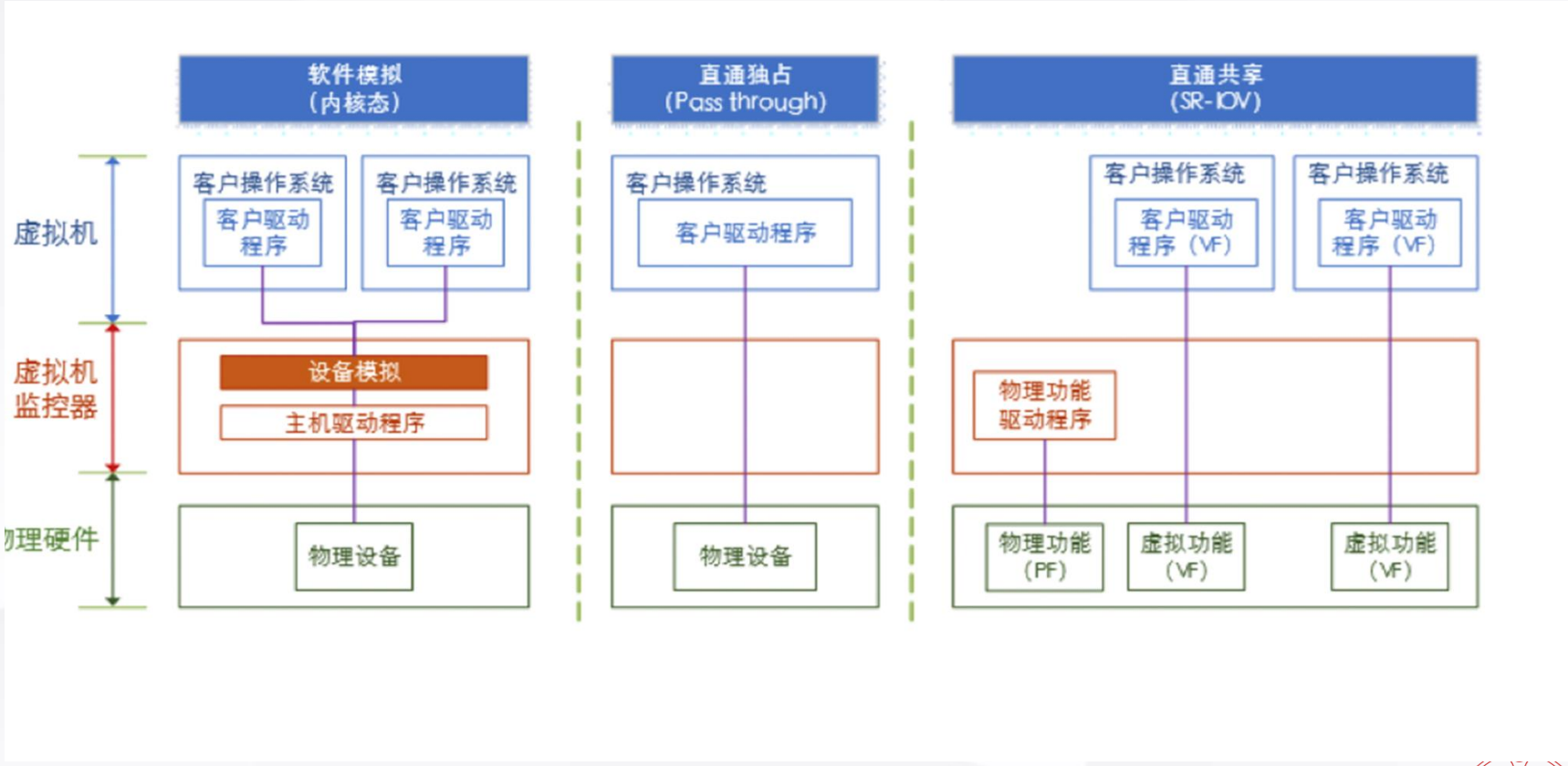
05

GPU虚拟化

SJTU

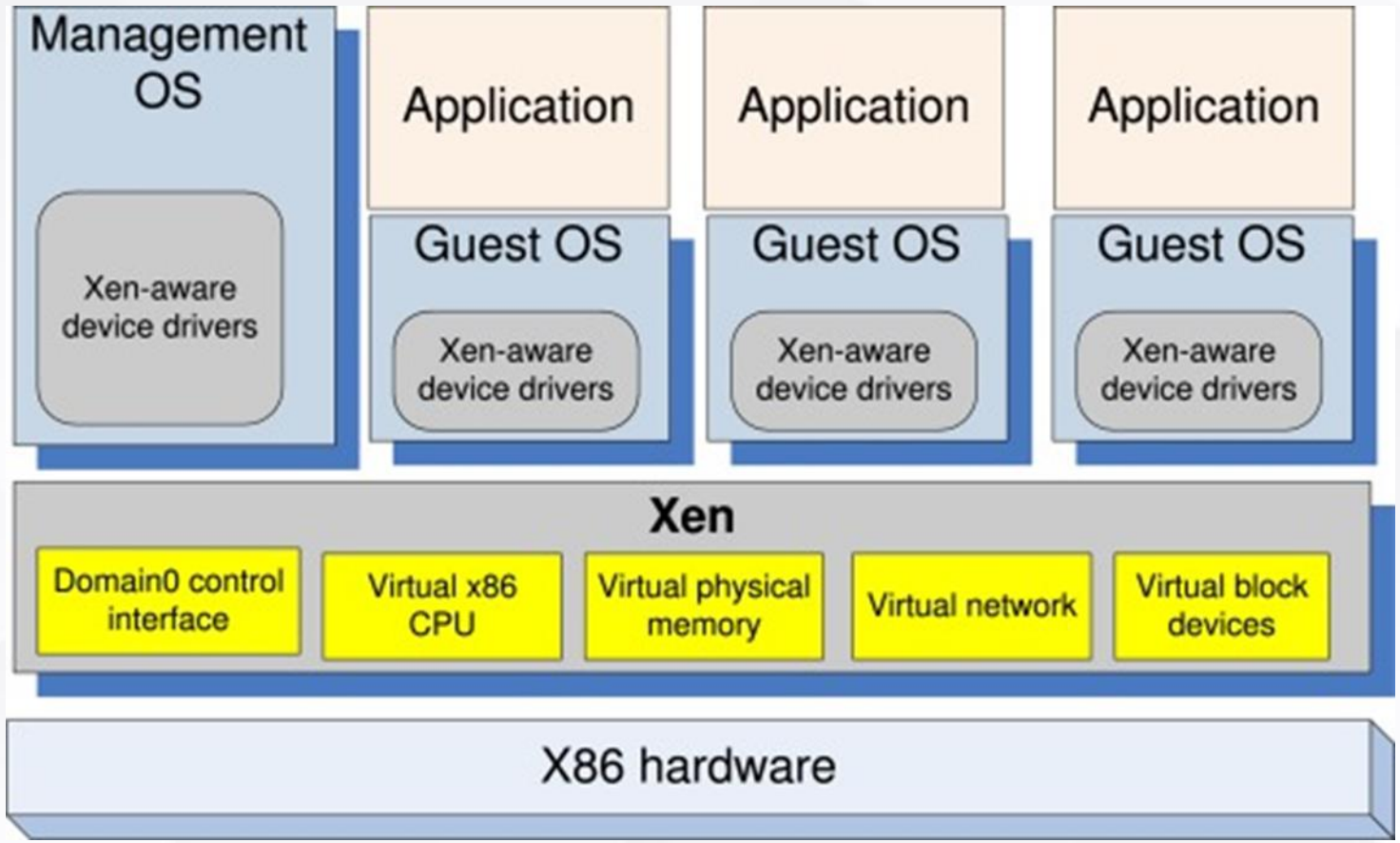
GPU 虚拟化的起源

- 软件模拟：VMware SVGA 3D software renderer，Citrix GPU accelerator
- 直通独占：VMware Virtual Dedicated Graphics Acceleration (vDGA)
- 直通共享：SR-IOV，gvirt



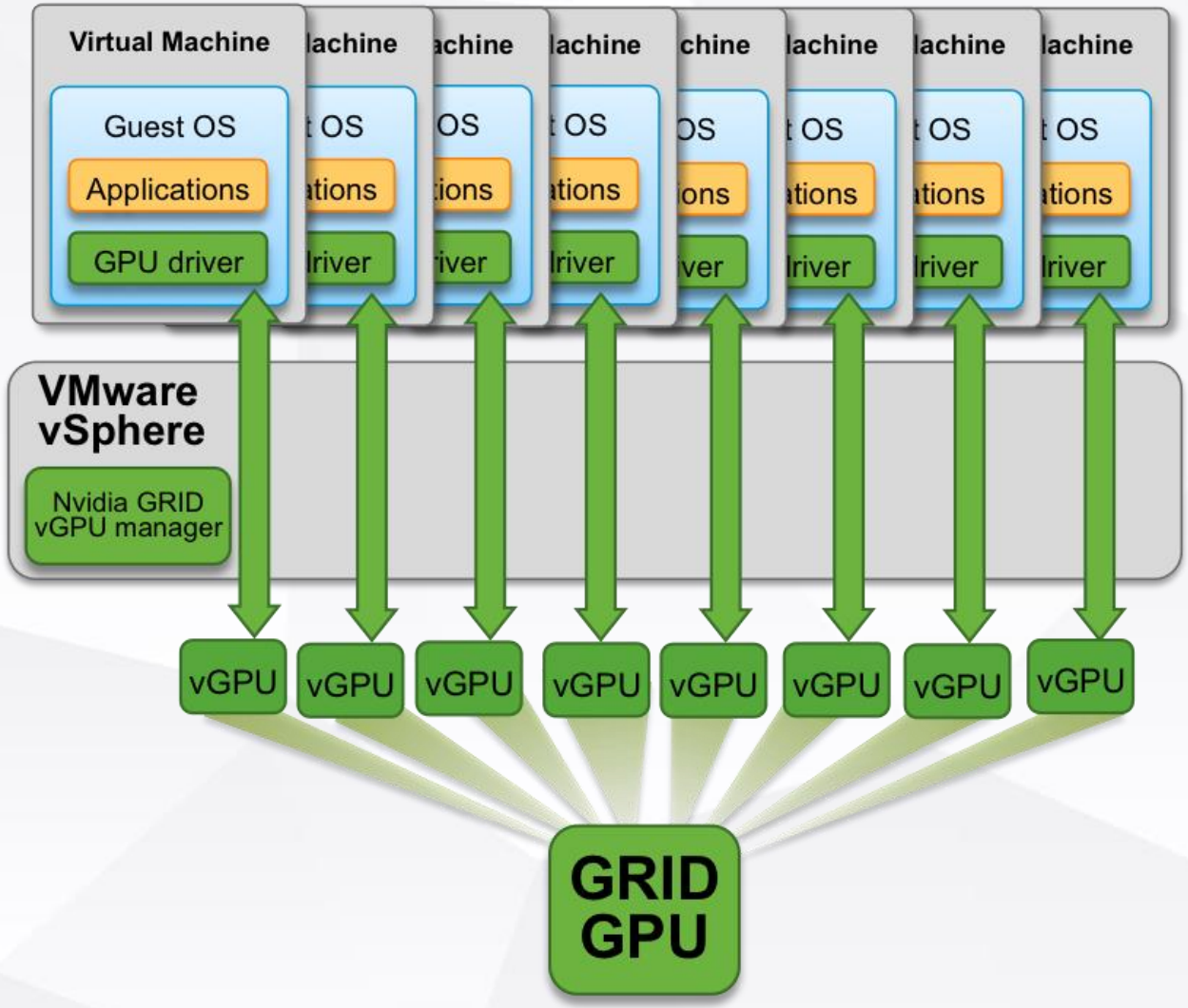
软件模拟(废弃)

- 软件模拟，无法真正继承GPU的优点
- 指令集更新， xen 驱动无法自适应
- 能耗高，内存高，CPU无法适应
- 性能低，3D模拟无法达到应用的QoS



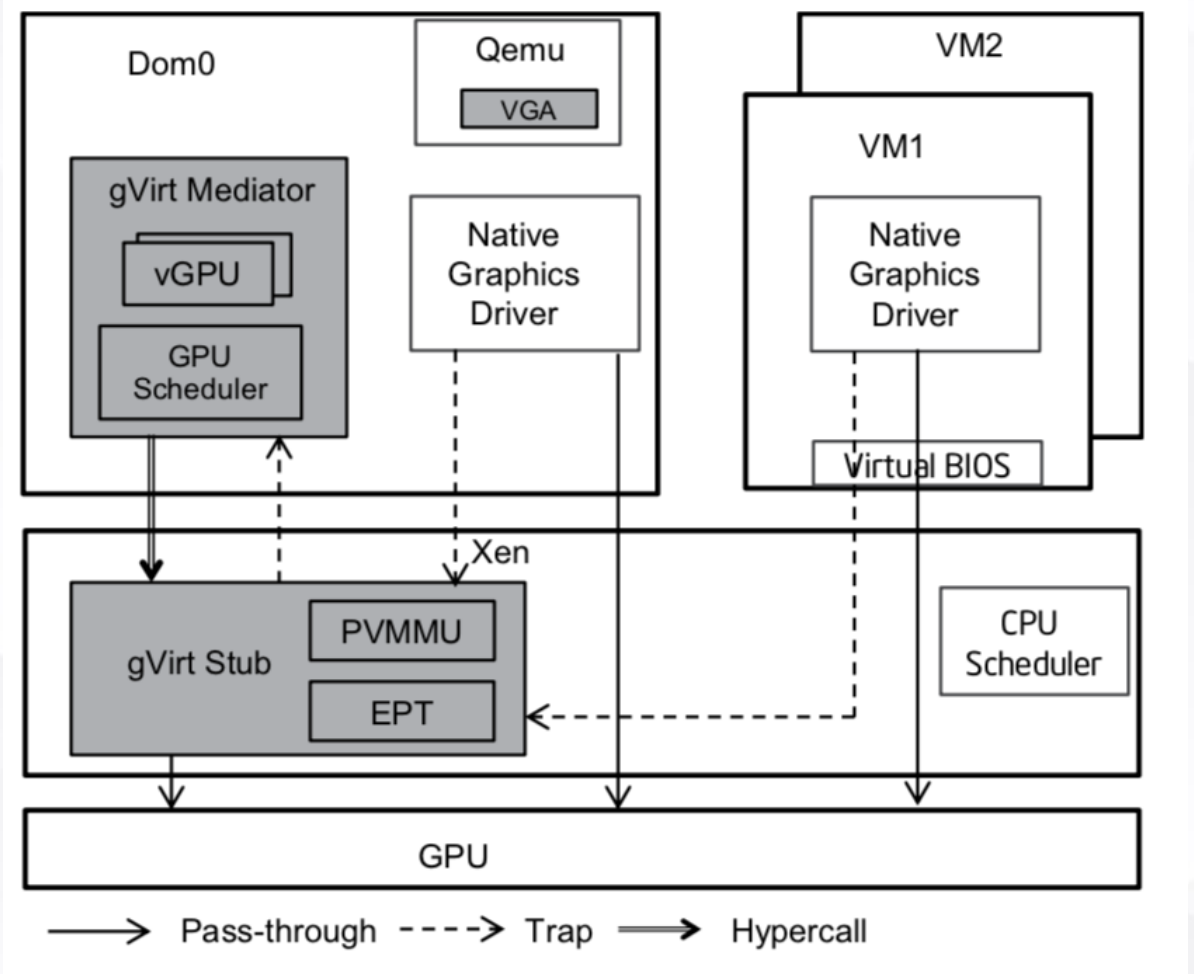
直通独占(废弃)

- 允许一个虚拟机直接独占一个GPU硬件
- 直接可以享受硬件加速
- 无法提高硬件利用率
- 安全隐患



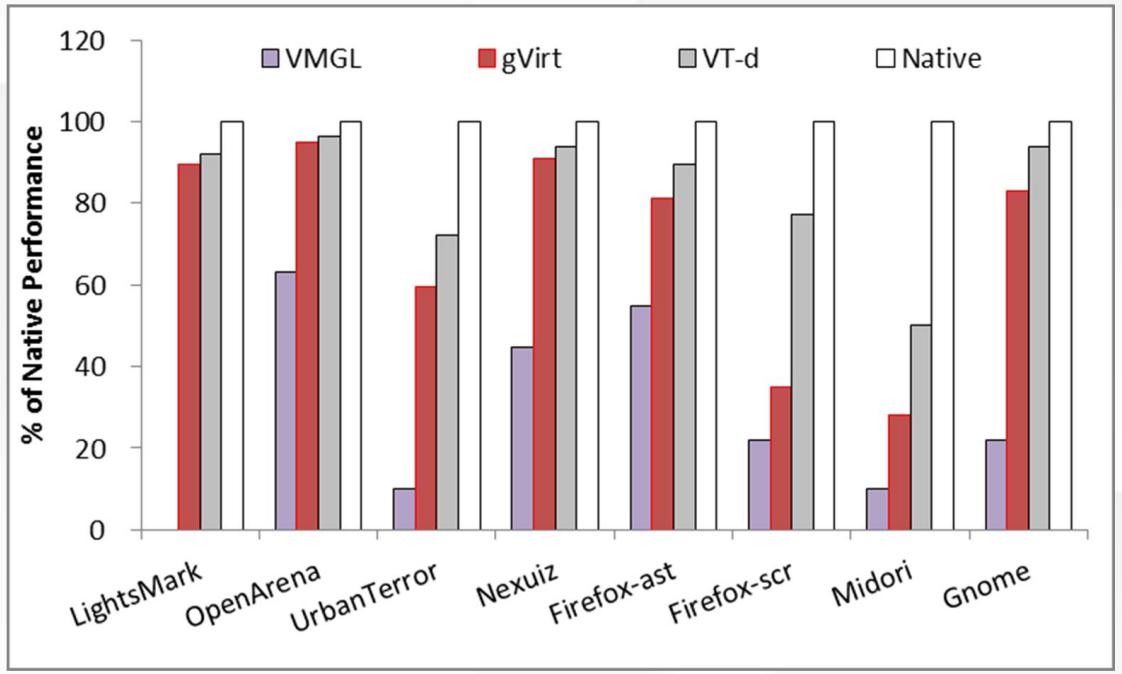
直通共享（共享GPU）

- 设置了一个gvirt stub，将所有的GPU请求送入 gvirt stub
- 通过pvmmu和ept，对一些指令和数据进行筛选，保证安全。
- gvirt mediator管理vGPU之间的切换
- GPU scheduler负责调度。

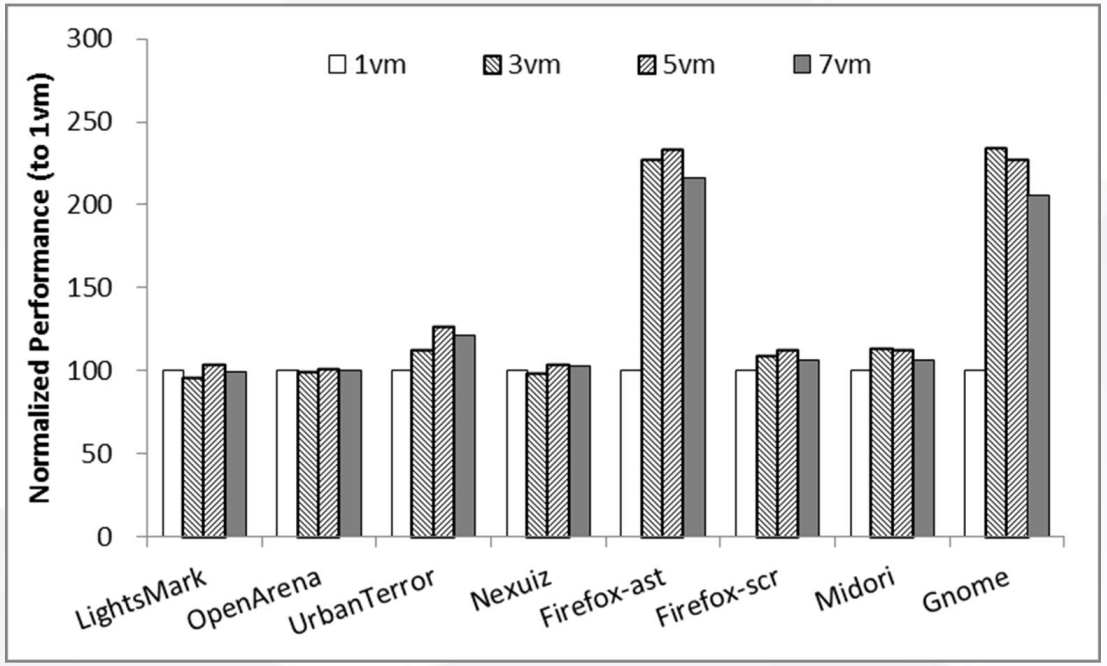


gVirt（ATC14）

直通共享（成就）



小于50%的性能损失



可以扩展出7个虚拟机

总结

- 上述的方法除了直通共享，其余的已经废弃。
- 对于硬件有依赖性。
- 现存直通共享技术需要对GPU内存有访问和修改权限
- 上述GPU虚拟化秉持一个核心原则：扩大GPU能力。
- 上述GPU虚拟化前提条件：GPU相较于CPU短缺。

GPU虚拟化发展：API 转发 (Thin-Client sosp15)

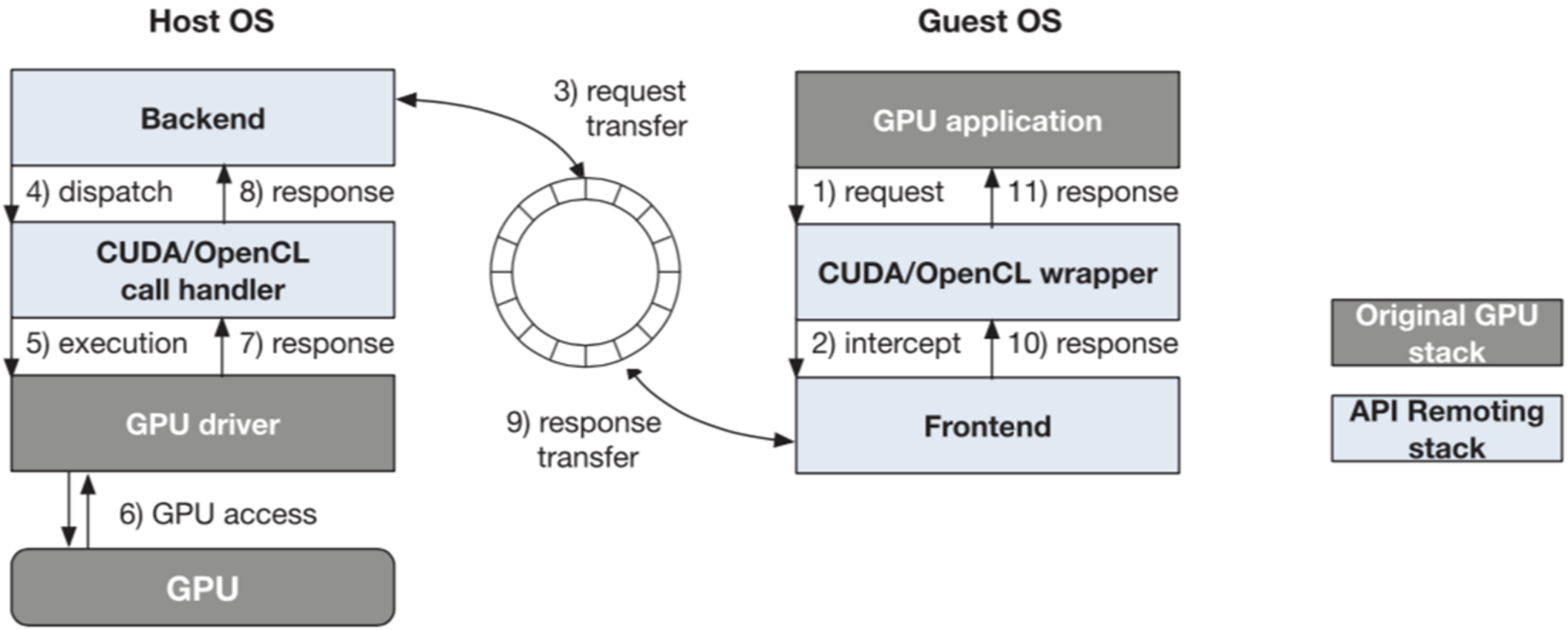
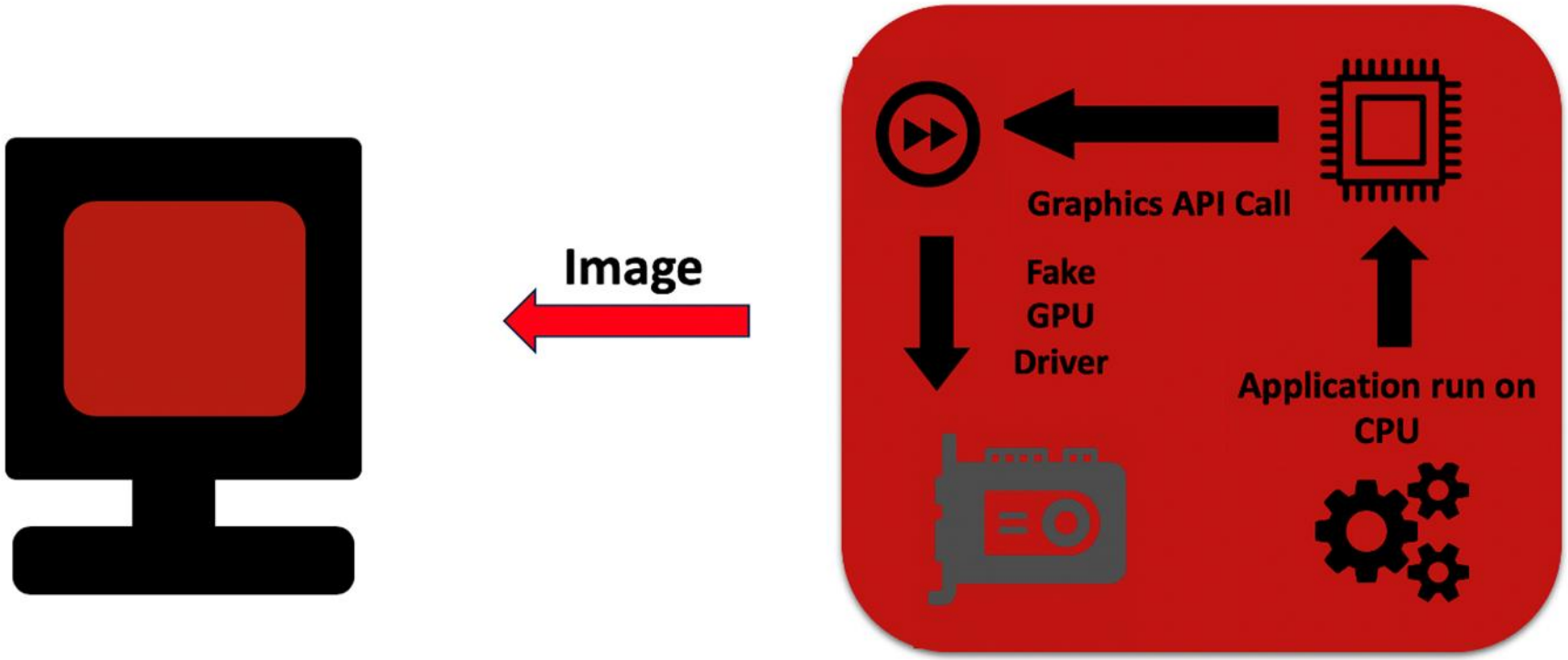


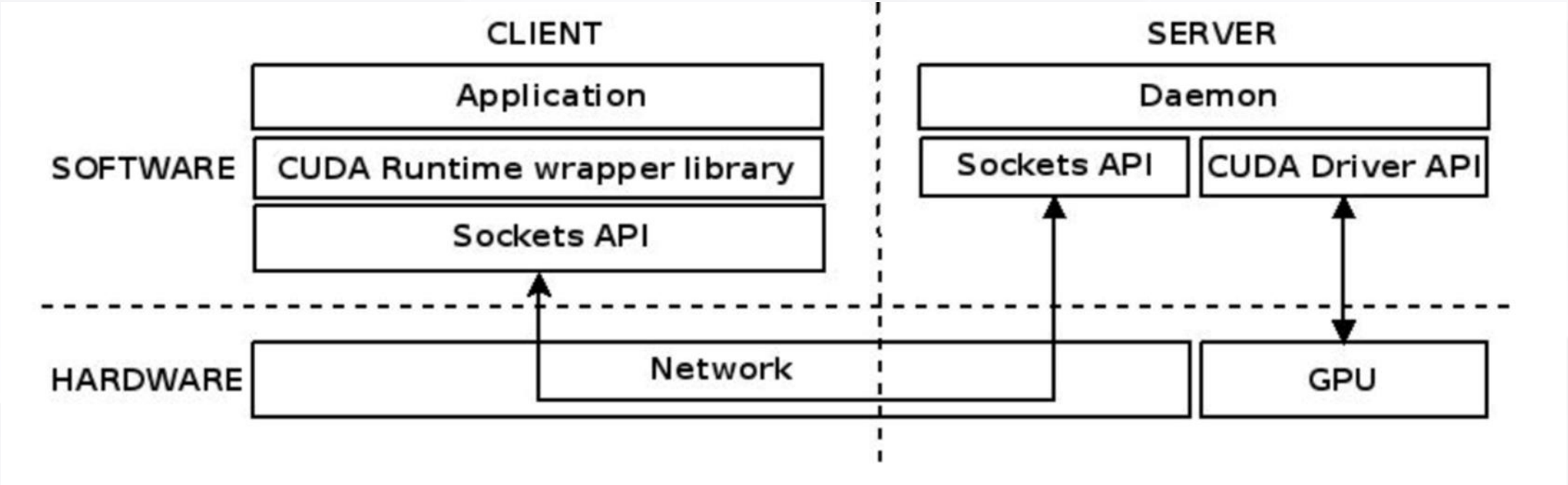
Fig. 2. Architecture of the API remoting approach.

GPU虚拟化发展：API 转发 (Thin-Client sosp15)



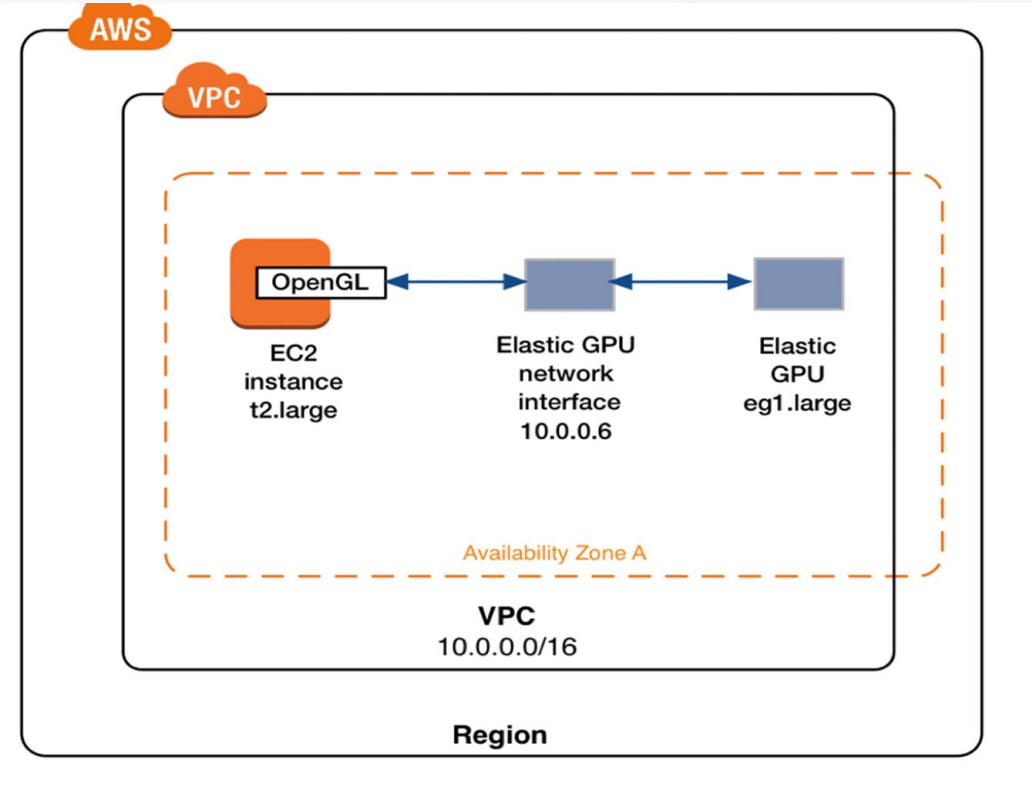
GPU虚拟化发展：远程虚拟化

- 通过wrapper library 截获相关GPU指令
- 通过sockets 将其送往远端GPU
- 享受到远端硬件加速
- 将结果通过socket传回



GPU虚拟化发展：远程虚拟化

- 扩大应用范围，不再局限于本地机器，远端机器也可以享用硬件加速（rCUDA）
- 相较于瘦客户端， 远程虚拟化可以零成本提高硬件配置 （gremote）
- 提高云内的服务器利用率。 （microsoft xcloud）



GPU虚拟化发展：总结

- GPU虚拟化需要走很长的路（从生态到架构）
- 相比于CPU, GPU架构和虚拟化处于初级状态
- 打破英伟达垄断，是GPU发展的目标
- 欢迎大家了解并真正认识GPU

谢谢！

饮 水 思 源　爱 国 荣 校