ADRIAN PERRIG & TORSTEN HOEFLER

A SIGINT in time saves a kill -9

Networks and Operating Systems (252-0062-00)
Chapter 11: Virtual Machine Monitors

NetKAT: A Formal System for the Verification of Networks
Dexter Kozen, Cornell University

Monday, March 30, 2015
16:15 - 17:15, CAB G61

ABSTRACT:
NetKAT is a relatively new programming language and logic for reasoning about packet switching networks that fits well with the popular software defined networking (SDN) paradigm. NetKAT was introduced quite recently by Anderson et al. (POPL 2014) and further developed by Foster et al. (POPL 2015). The system provides general-purpose programming constructs such as parallel and sequential composition, conditional tests and iteration as well as special-purpose primitives for querying and modifying packet headers and encoding network topologies. **The language allows the desired behavior of a network to be specified equationally. It has a formal mathematical semantics** and a deductive system that is sound and complete over that semantics, as well as an efficient decision procedure for the automatic verification of equationally-defined properties of networks.

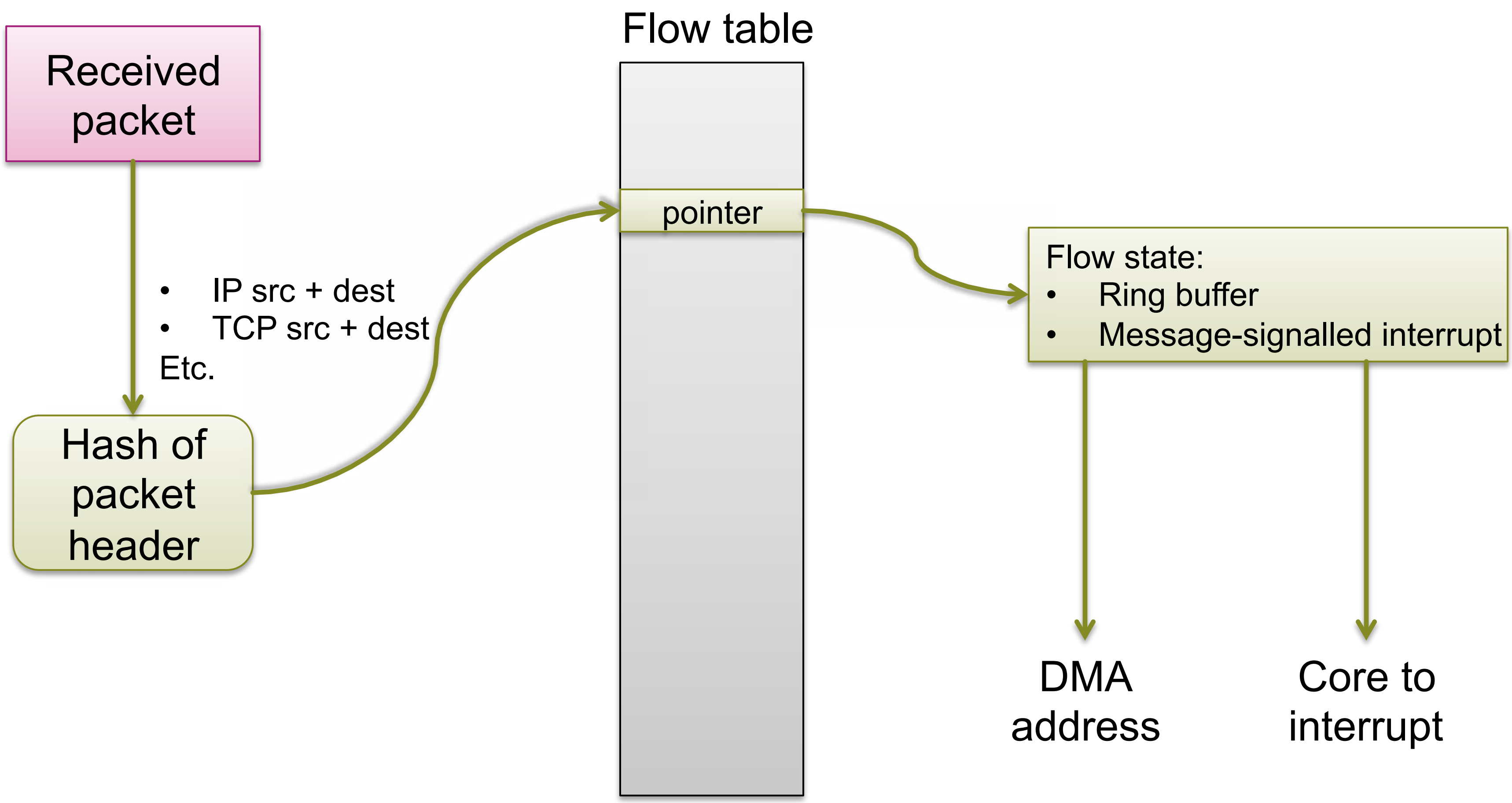
Our Small Quiz

- **True or false (raise hand)**
  - Spooling can be used to improve access times
  - Buffering can cope with device speed mismatches
  - The Linux kernel identifies devices using a number
  - From userspace, devices in Linux are identified through files
  - Standard BSD sockets require two or more copies at the host
  - Protocols are processed in the first level interrupt handler
  - The second level interrupt handler copies the packet data to userspace
  - Deferred procedure calls can be executed in any process context
  - Unix mbufs (and skbufs) enable protocol-independent processing
  - Network I/O is not performance-critical
  - NAPI's design aims to reduce the CPU load
  - NAPI uses polling to accelerate packet processing
  - TCP offload reduces the server CPU load
  - TCP offload can accelerate applications

Receive-side scaling

- **Insight:**
  - Too much traffic for one core to handle
  - Cores aren't getting any faster
    ⇒ Must parallelize across cores
- **Key idea: handle different flows on different cores**
  - But: how to determine flow for each packet?
  - Can't do this on a core: same problem!
- **Solution: demultiplex on the NIC**
  - DMA packets to per-flow buffers / queues
  - Send interrupt only to core handling flow

Receive-side scaling



Receive-side scaling

- **Can balance flows across cores**
  - Note: doesn't help with one big flow!
- **Assumes:**
  - *n* cores processing *m* flows is faster than one core
- **Hence:**
  - Network stack and protocol graph must *scale* on a multiprocessor.
- **Multiprocessor scaling: topic for later (see DPHPC class)**

Virtual Machine Monitors

Literature: **Barham et al.: Xen and the art of virtualization** and **Anderson, Dahlin: Operating Systems: Principles and Practice, Chapter 14**

Virtual Machine Monitors

- **Basic definitions**
- **Why would you want one?**
- **Structure**
- **How does it work?**
  - CPU
  - MMU
  - Memory
  - Devices
  - Network

- Acknowledgement: Thanks to Steve Hand for some of the slides!

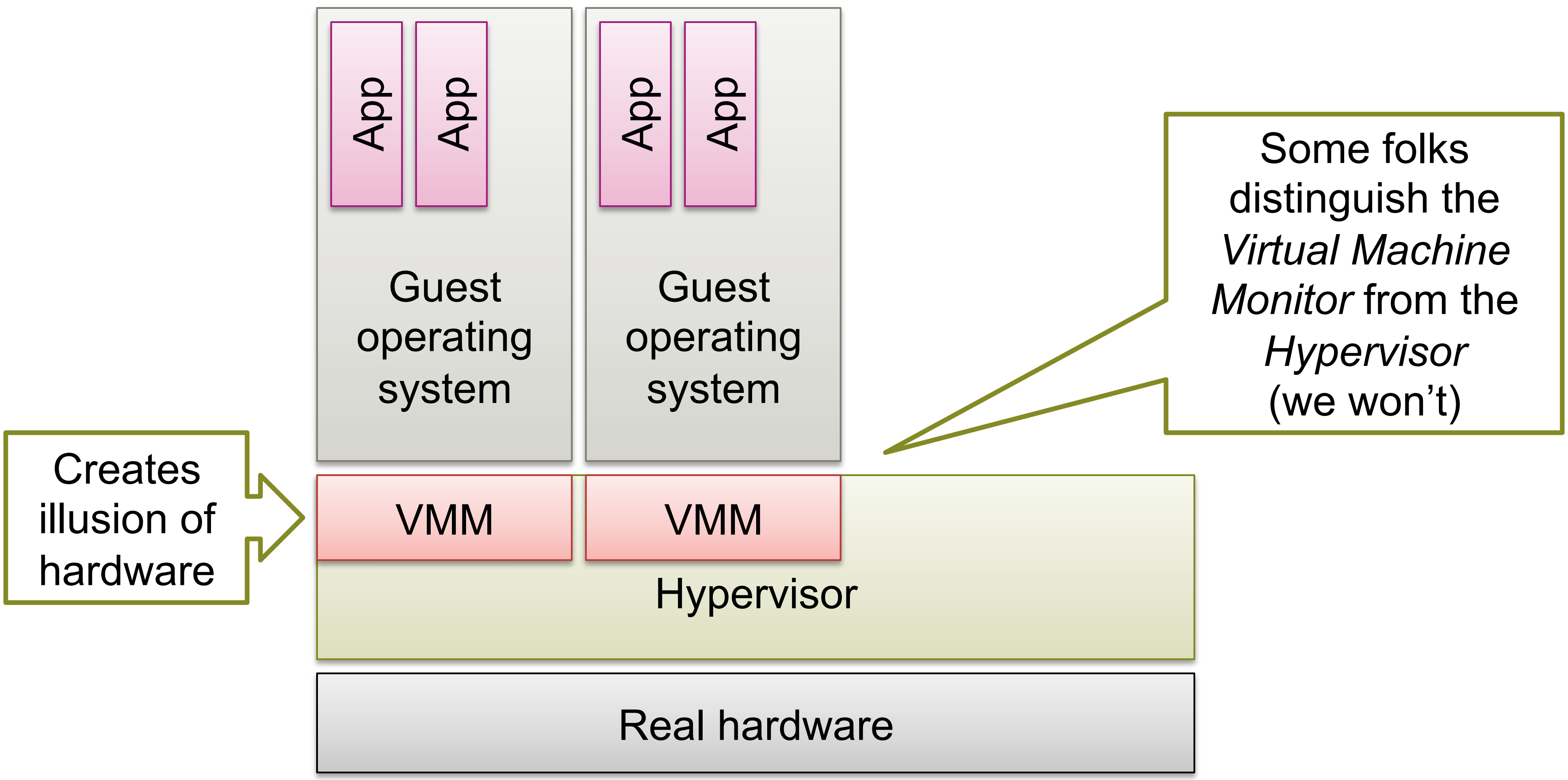
Xen

What is a Virtual Machine Monitor?

- **Virtualizes an entire (hardware) machine**
  - Contrast with OS processes
  - Interface provided is “illusion of real hardware”
  - Applications are therefore complete Operating Systems themselves
  - Terminology: ***Guest Operating Systems***

- **Old idea: IBM VM/CMS (1960s)**
  - Recently revived: VMware, Xen, Hyper-V, kvm, etc.

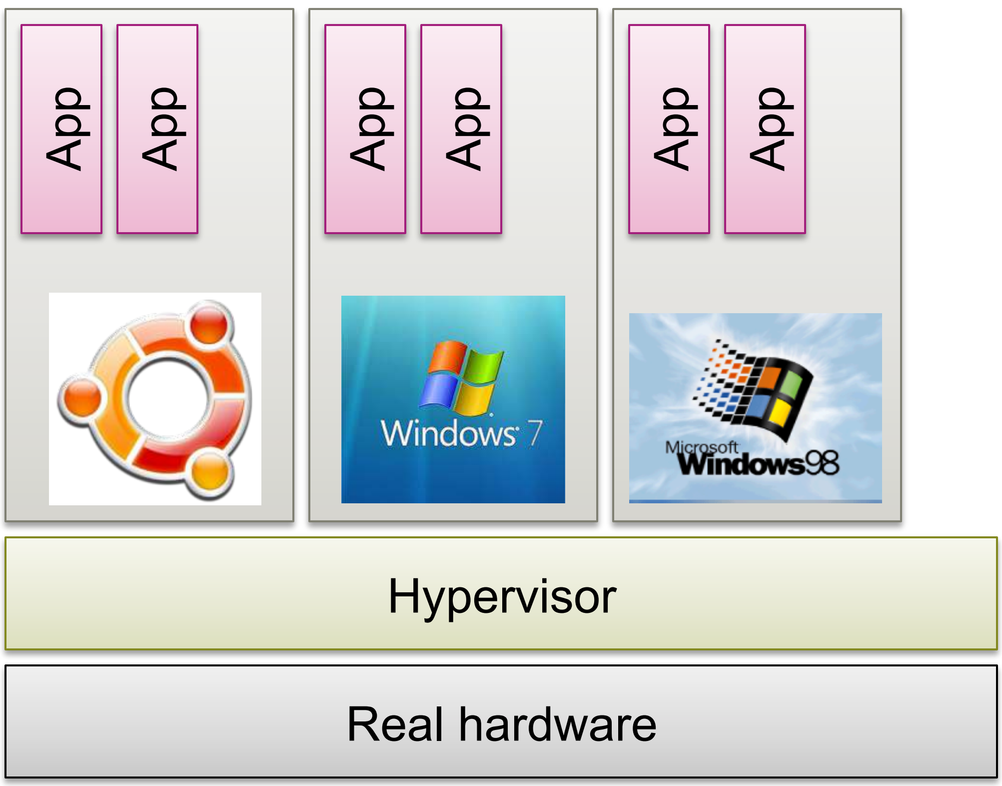
VMMs and Hypervisors



Why would you want one?

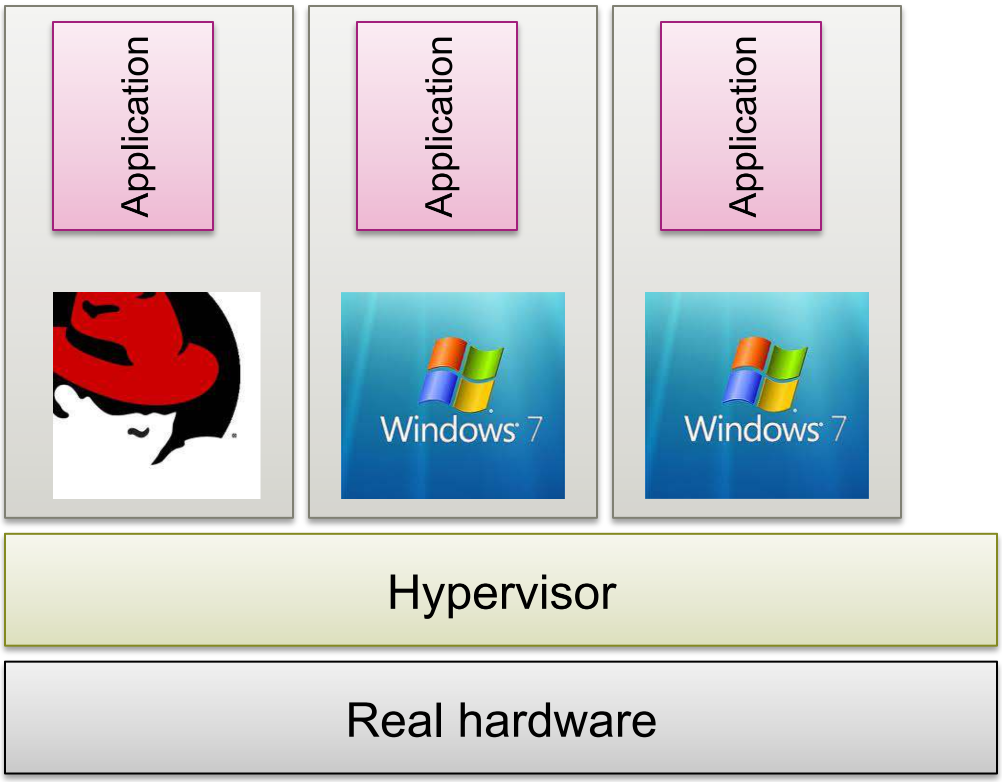
- Server consolidation (program assumes own machine)
- Performance isolation
- Backward compatibility
- Cloud computing (unit of selling cycles)
- OS development/testing
- Something under the OS: replay, auditing, trusted computing, rootkits

Running multiple OSes on one machine



- **Application compatibility**
  - I use Debian for almost everything, but I edit slides in PowerPoint
  - Some people compile Barrelfish in a Debian VM over Windows 7 with Hyper-V
- **Backward compatibility**
  - Nothing beats a Windows 98 virtual machine for playing old computer games

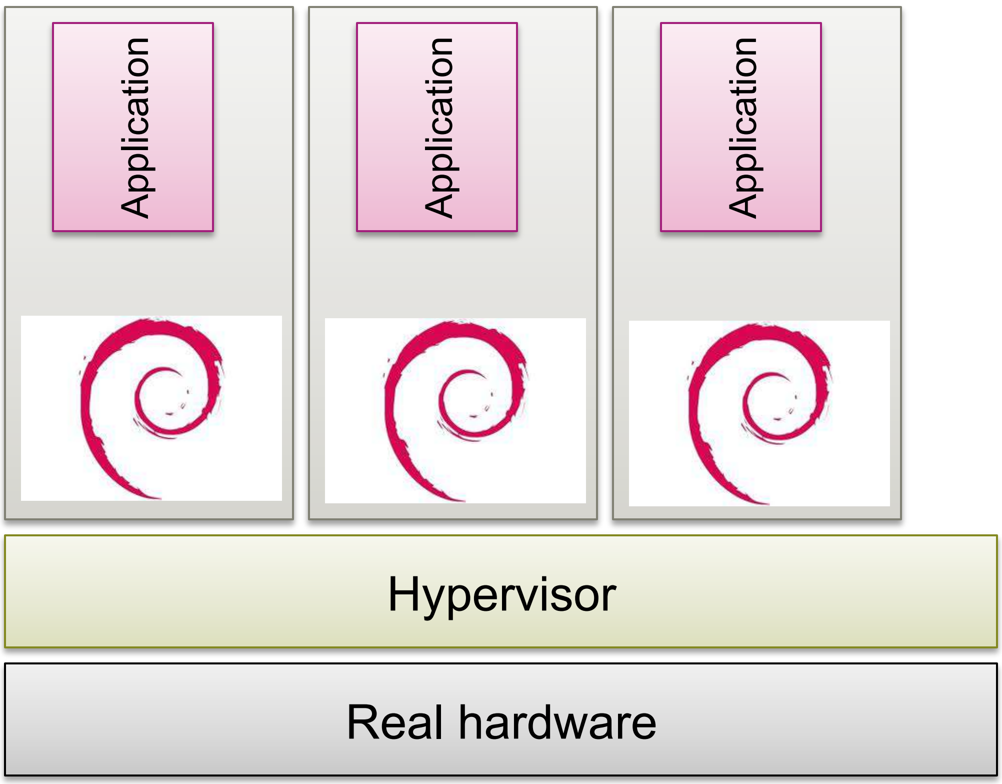
Server consolidation



- Many applications assume they have the machine to themselves
- Each machine is mostly idle

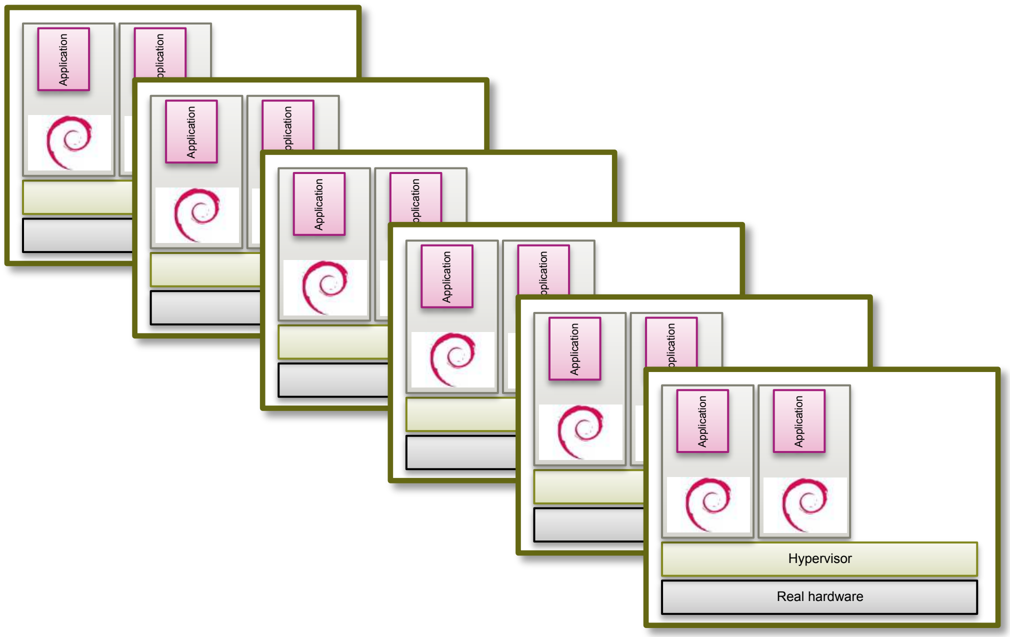
⇒ Consolidate servers onto a single physical machine

Resource isolation



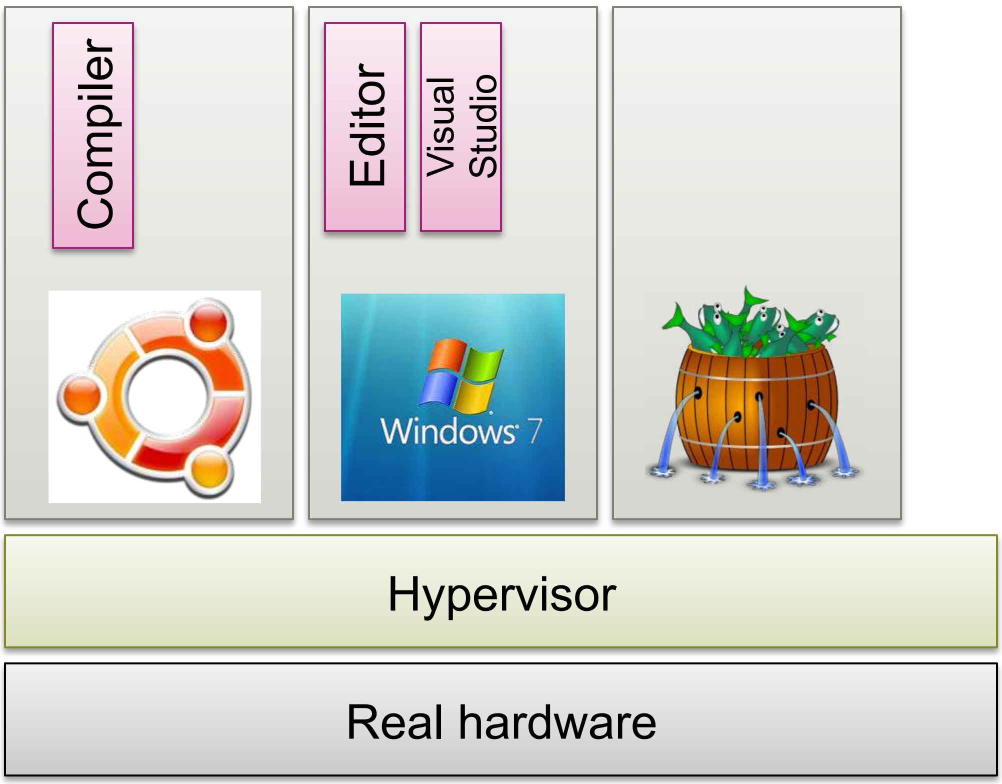
- **Surprisingly, modern OSes do not have an abstraction for a single application**
- **Performance isolation can be critical in some enterprises**
- **Use virtual machines as *resource containers***

Cloud computing



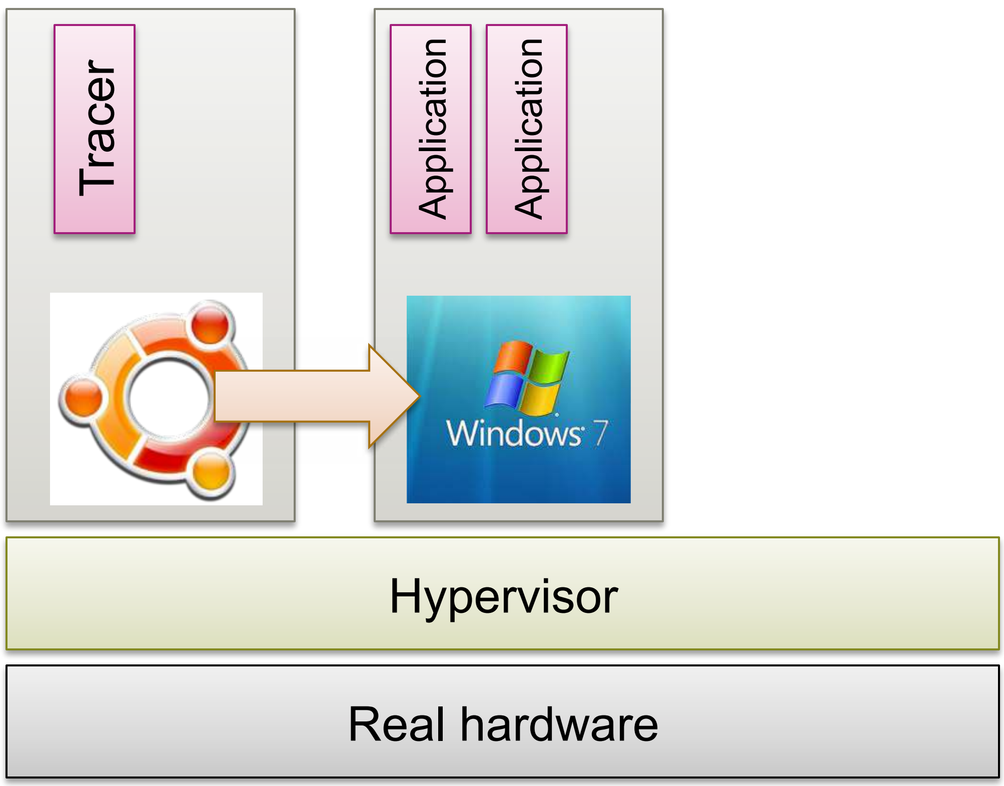
- **Selling computing capacity on demand**
  - E.g. Amazon EC2, GoGrid, etc.
- **Hypervisors decouple *allocation* of resources (VMs) from *provisioning* of infrastructure (physical machines)**

Operating System development



- Building and testing a new OS without needing to reboot real hardware
- VMM often gives you more information about faults than real hardware anyway

Other cool applications…

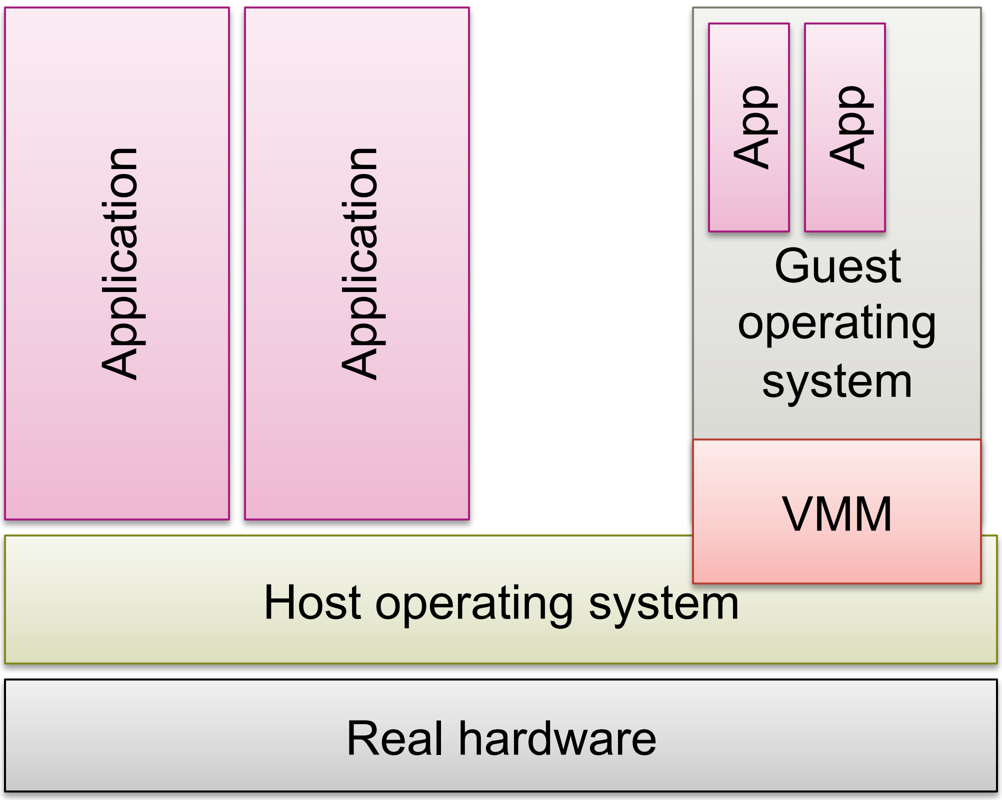


- Tracing
- Debugging
- Execution replay
- Lock-step execution
- Live migration
- Rollback
- Speculation
- Etc.…

How does it all work?

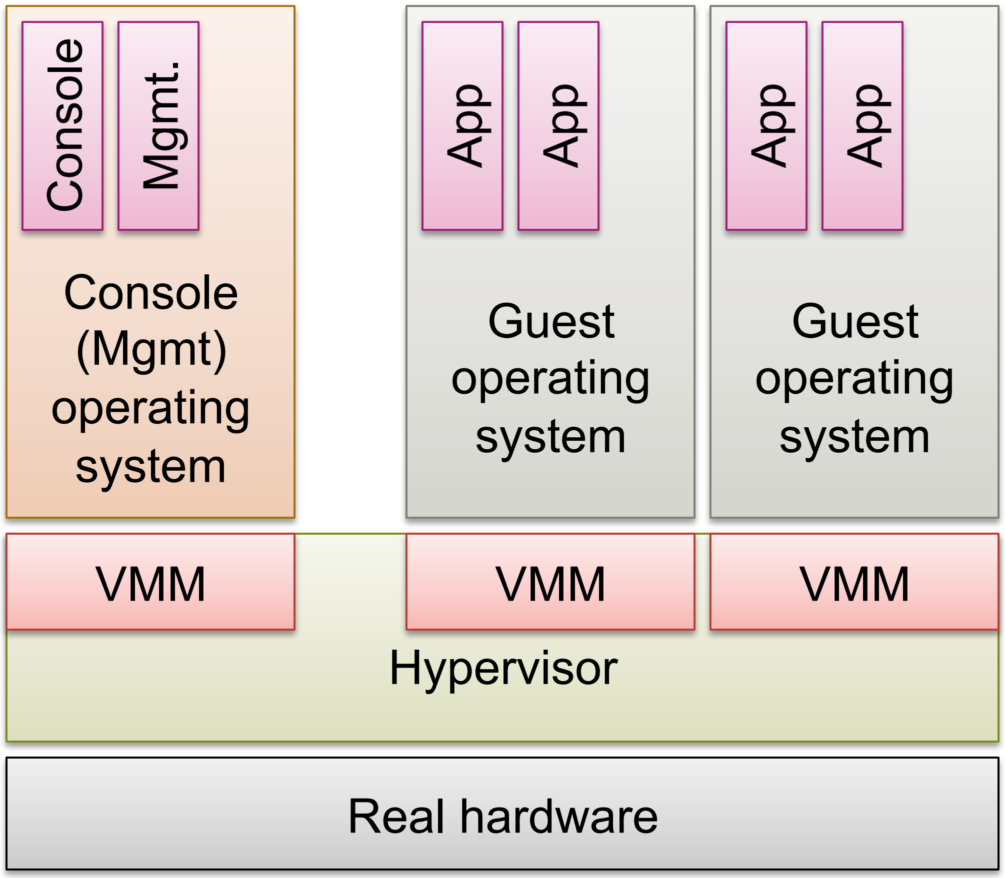
- **Note: a hypervisor is basically an OS**
  - With an “unusual API”
- **Many functions quite similar:**
  - Multiplexing resources
  - Scheduling, virtual memory, device drivers
- **Different:**
  - Creating the illusion of hardware to “applications”
  - Guest OSes are less flexible in resource requirements

Hosted VMMs



Examples:
- VMware workstation
- Linux KVM
- Microsoft Hyper-V
- VirtualBox

Hypervisor-based VMMs



Examples:
- VMware ESX
- IBM VM/CMS
- Xen

How to virtualize…

- The CPU (s)?
- The MMU?
- Physical memory?
- Devices (disks, etc.)?
- The Network

and?

Virtualizing the CPU

- **A CPU architecture is *strictly virtualizable* if it can be perfectly emulated over itself, with all non-privileged instructions executed natively**

- **Privileged instructions ⇒ trap**
  - Kernel-mode (i.e., the VMM) emulates instruction
  - Guest's kernel mode is actually user mode

    Or another, extra privilege level (such as ring 1)

- **Examples: IBM S/390, Alpha, PowerPC**

Virtualizing the CPU

- **A strictly virtualizable processor can execute a complete native Guest OS**
  - Guest applications run in user mode as before
  - Guest kernel works exactly as before
- **Problem: x86 architecture is not virtualizable ☹**
  - About 20 instructions are sensitive but not privileged
  - Mostly segment loads and processor flag manipulation

Non-virtualizable x86: example

- **`PUSHF/POPF` instructions**
  - Push/pop condition code register
  - Includes interrupt enable flag (`IF`)

- **Unprivileged instructions: fine in user space!**
  - `IF` is ignored by `POPF` in user mode, not in kernel mode

⇒ VMM can’t determine if Guest OS wants interrupts disabled!

- Can’t cause a trap on a (privileged) `POPF`
- Prevents correct functioning of the Guest OS

Solutions

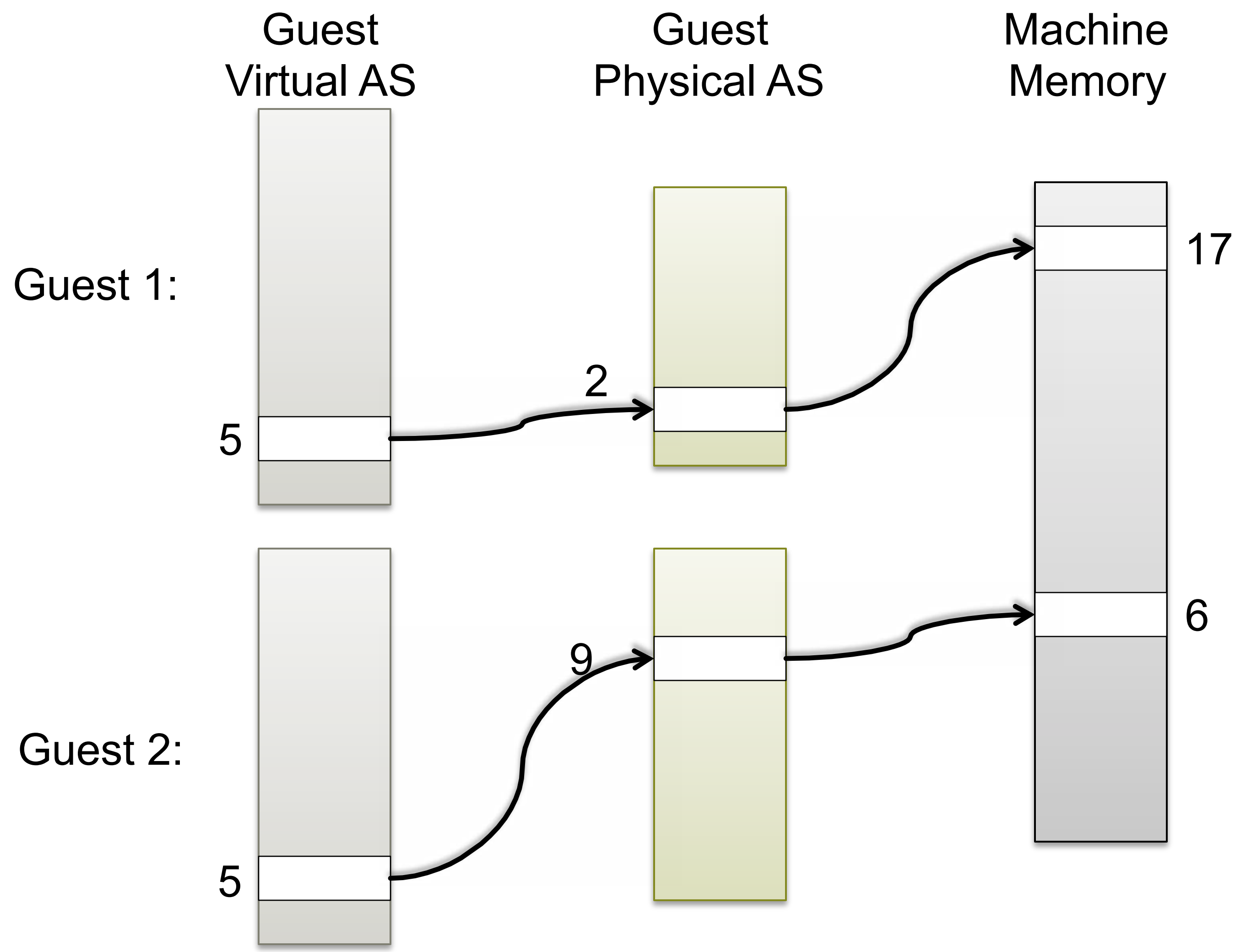
1. **Emulation: emulate all kernel-mode code in software**
   - Very slow – particularly for I/O intensive workloads
   - Used by, e.g., SoftPC
2. ***Paravirtualization*: modify Guest OS kernel**
   - Replace critical calls with explicit trap instruction to VMM
   - Also called a "HyperCall" (used for all kinds of things)
   - Used by, e.g., Xen
3. **Binary rewriting:**
   - Protect kernel instruction pages, trap to VMM on first IFetch
   - Scan page for POPF instructions and replace
   - Restart instruction in Guest OS and continue
   - Used by, e.g. VMware
4. **Hardware support: Intel VT-x, AMD-V**
   - Extra processor mode causes `POPF` to trap

Virtualizing the MMU

- **Hypervisor allocates memory to VMs**
  - Guest assumes control over all physical memory
  - VMM can't let Guest OS to install mappings
- **Definitions needed:**
  - *Virtual* address: a virtual address in the guest
  - *Physical* address: as seen by the guest
  - *Machine* address: real physical address

    As seen by the Hypervisor

Virtual/Physical/Machine



MMU Virtualization

- **Critical for performance, challenging to make fast, especially SMP**
  - Hot-unplug unnecessary virtual CPUs
  - Use multicast TLB flush paravirtualizations etc.
- **Xen supports 3 MMU virtualization modes**
  1. Direct ("Writable") pagetables
  2. Shadow pagetables
  3. Hardware Assisted Paging
- **OS Paravirtualization compulsory for #1, optional (and very beneficial) for #2&3**

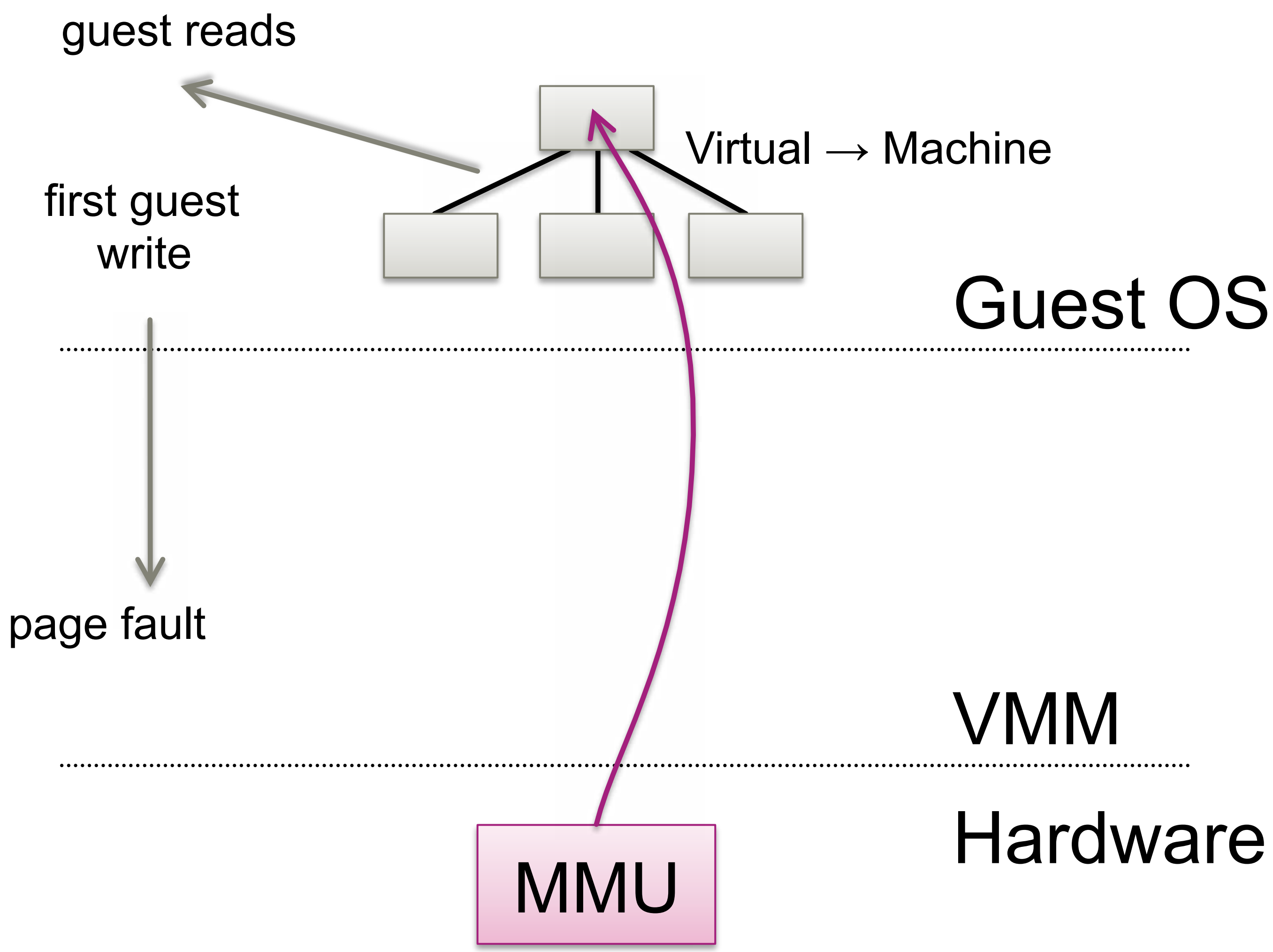
Paravirtualization approach

- **Guest OS creates page tables the hardware uses**
  - VMM must validate all updates to page tables
  - Requires modifications to Guest OS
  - Not quite enough…
- **VMM must check *all* writes to PTEs**
  - Write-protect all PTEs to the Guest kernel
  - Add a HyperCall to update PTEs
  - Batch updates to avoid trap overhead
  - OS is now aware of machine addresses
  - Significant overhead!

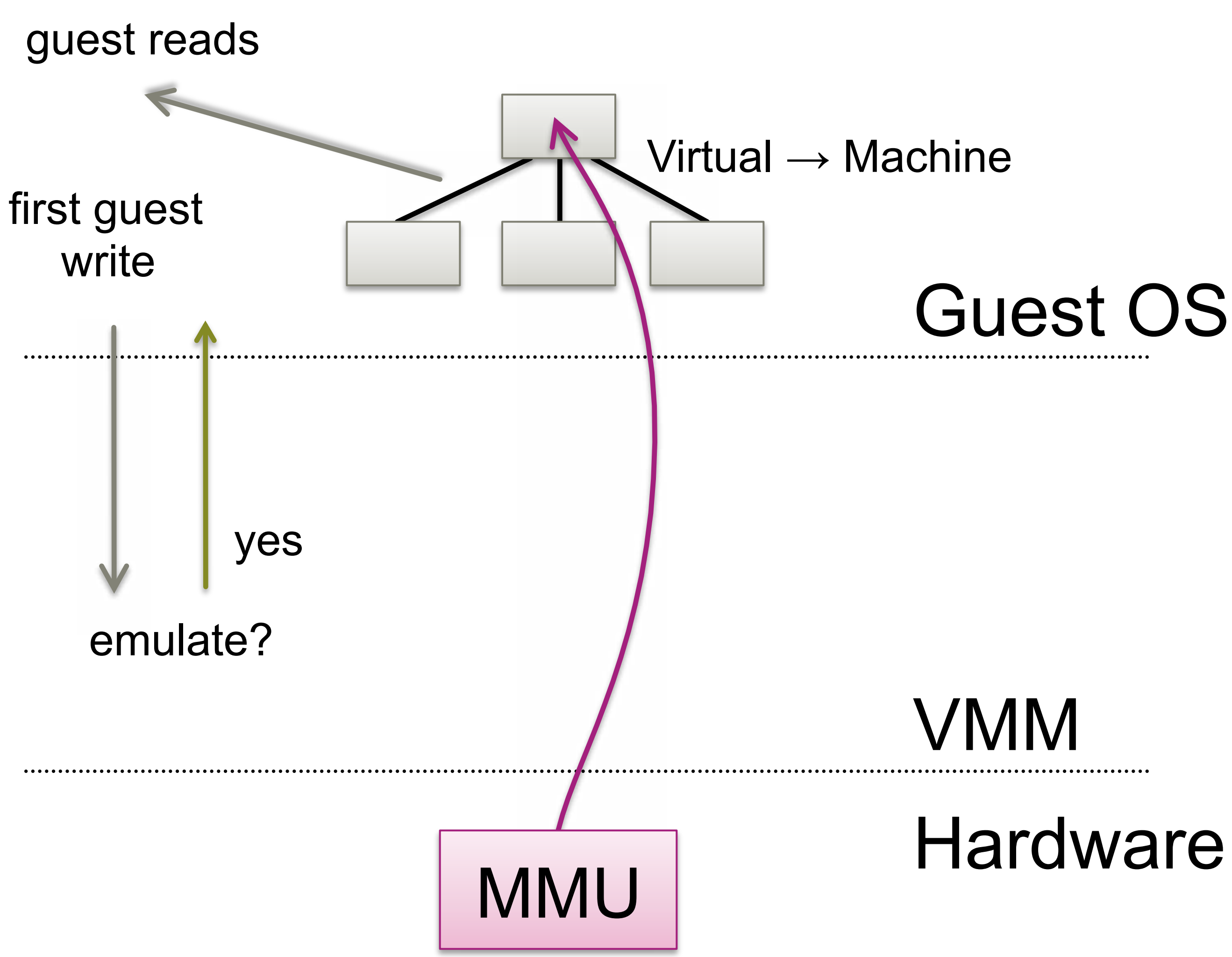
Paravirtualizing the MMU

- **Guest OSes allocate and manage own PTs**
  - Hypercall to change PT base
- **VMM must validate PT updates before use**
  - Allows incremental updates, avoids revalidation
- **Validation rules applied to each PTE:**
  - 1. Guest may only map pages it owns
  - 2. Pagetable pages may only be mapped RO
- **VMM traps PTE updates and emulates, or 'unhooks' PTE page for bulk updates**

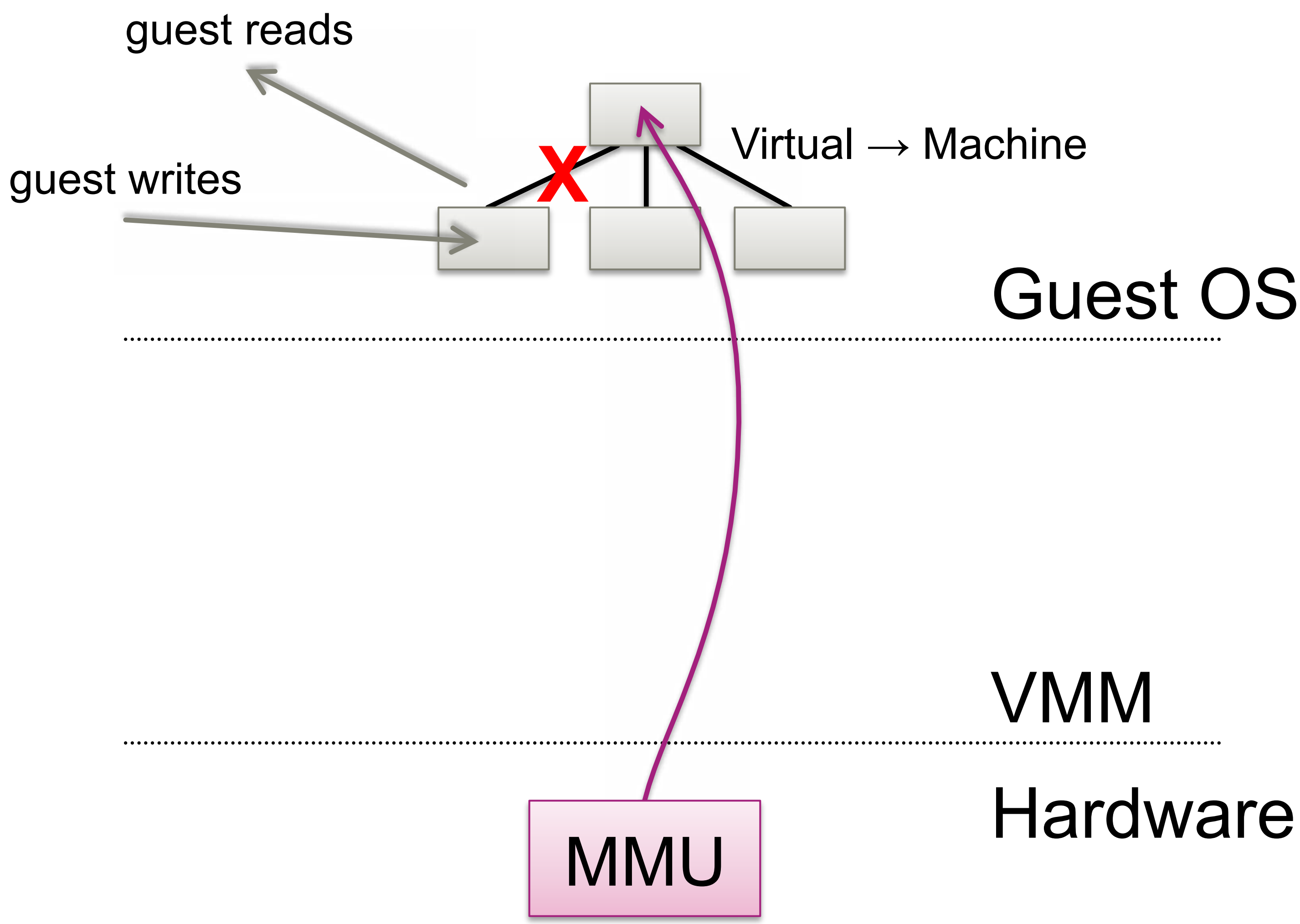
Writeable Page Tables : 1 – Write fault



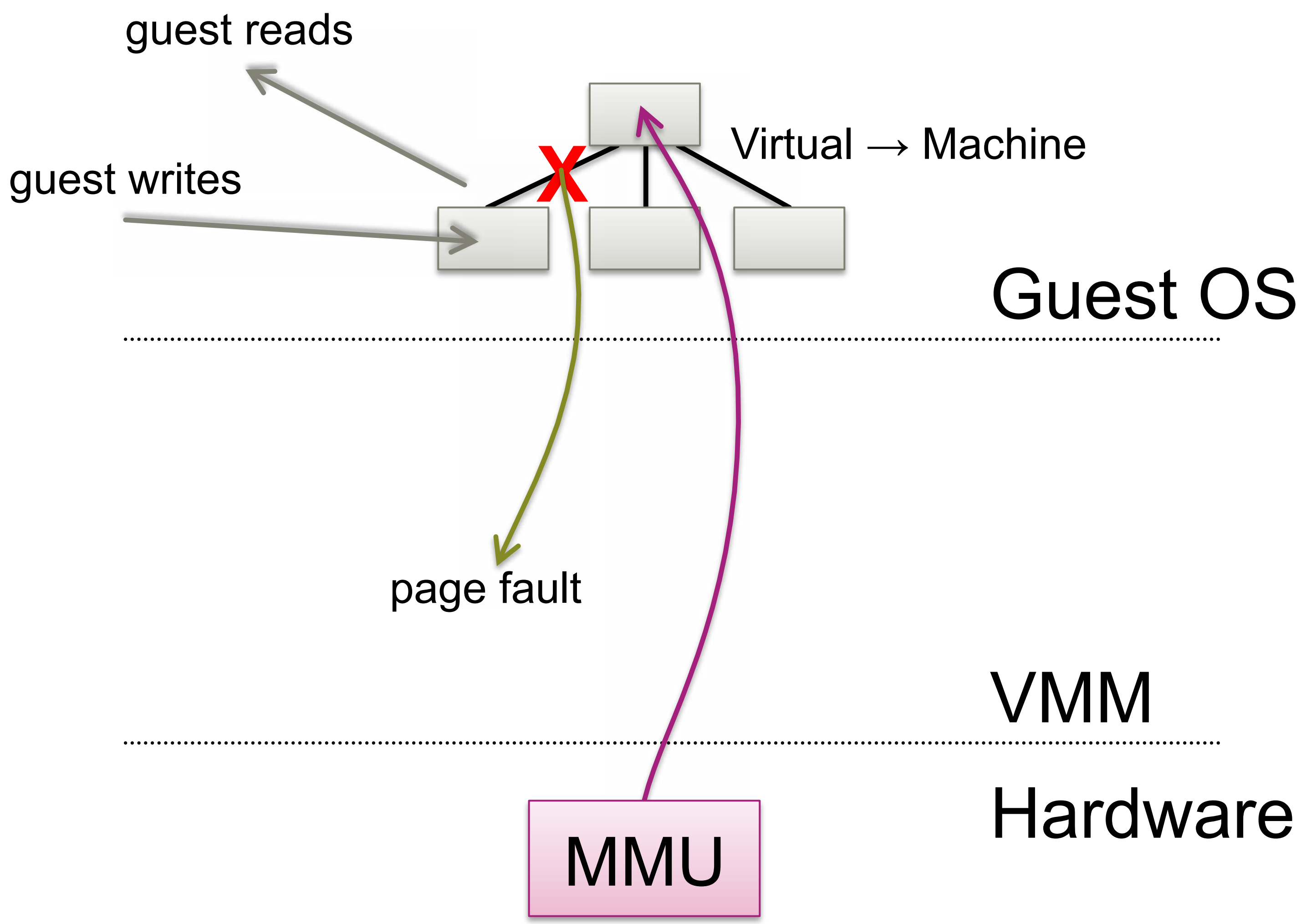
Writeable Page Tables : 2 – Emulate?



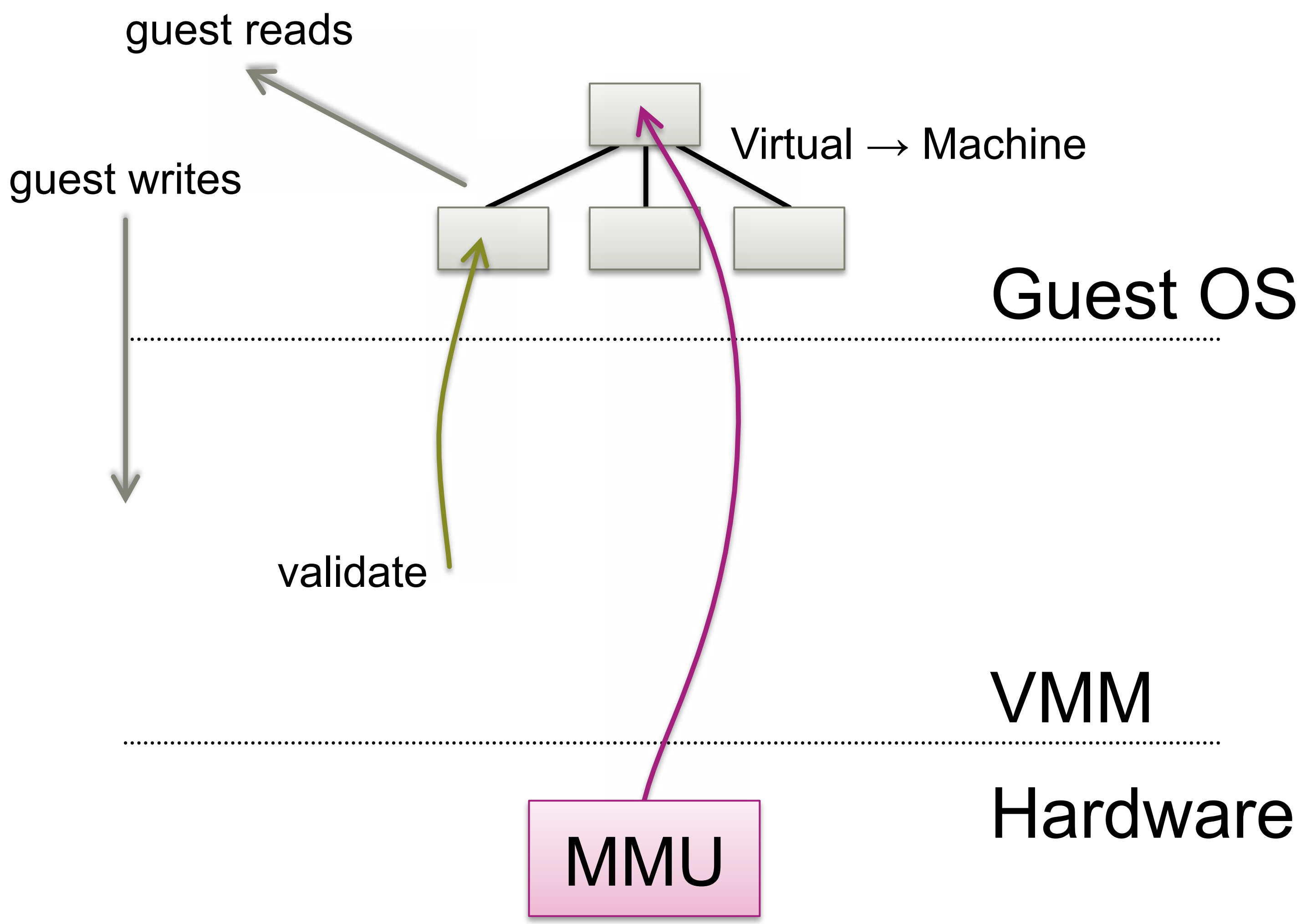
Writeable Page Tables : 3 - Unhook



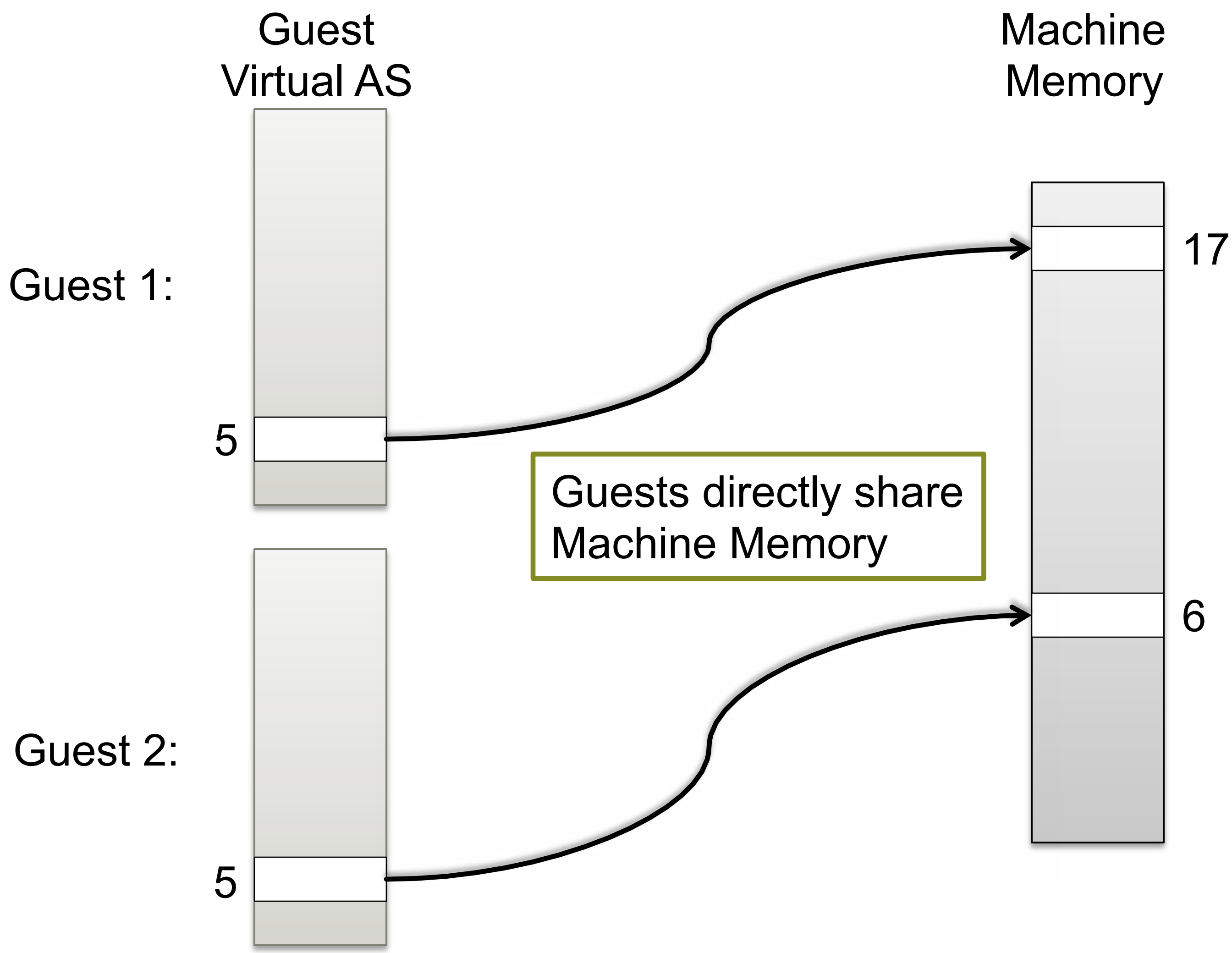
Writeable Page Tables : 4 - First Use



Writeable Page Tables : 5 – Re-hook



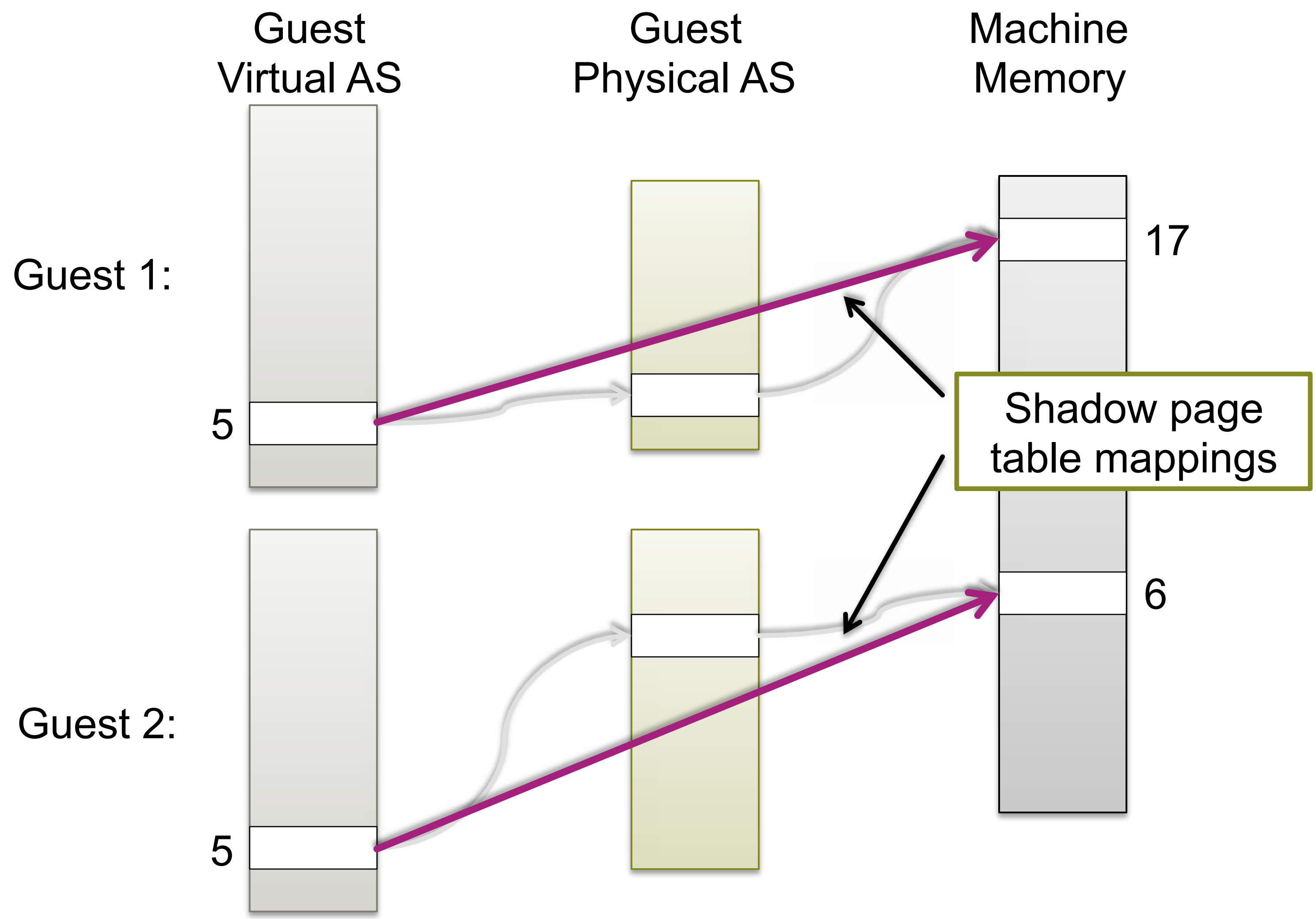
Writeable page tables require paravirtualization



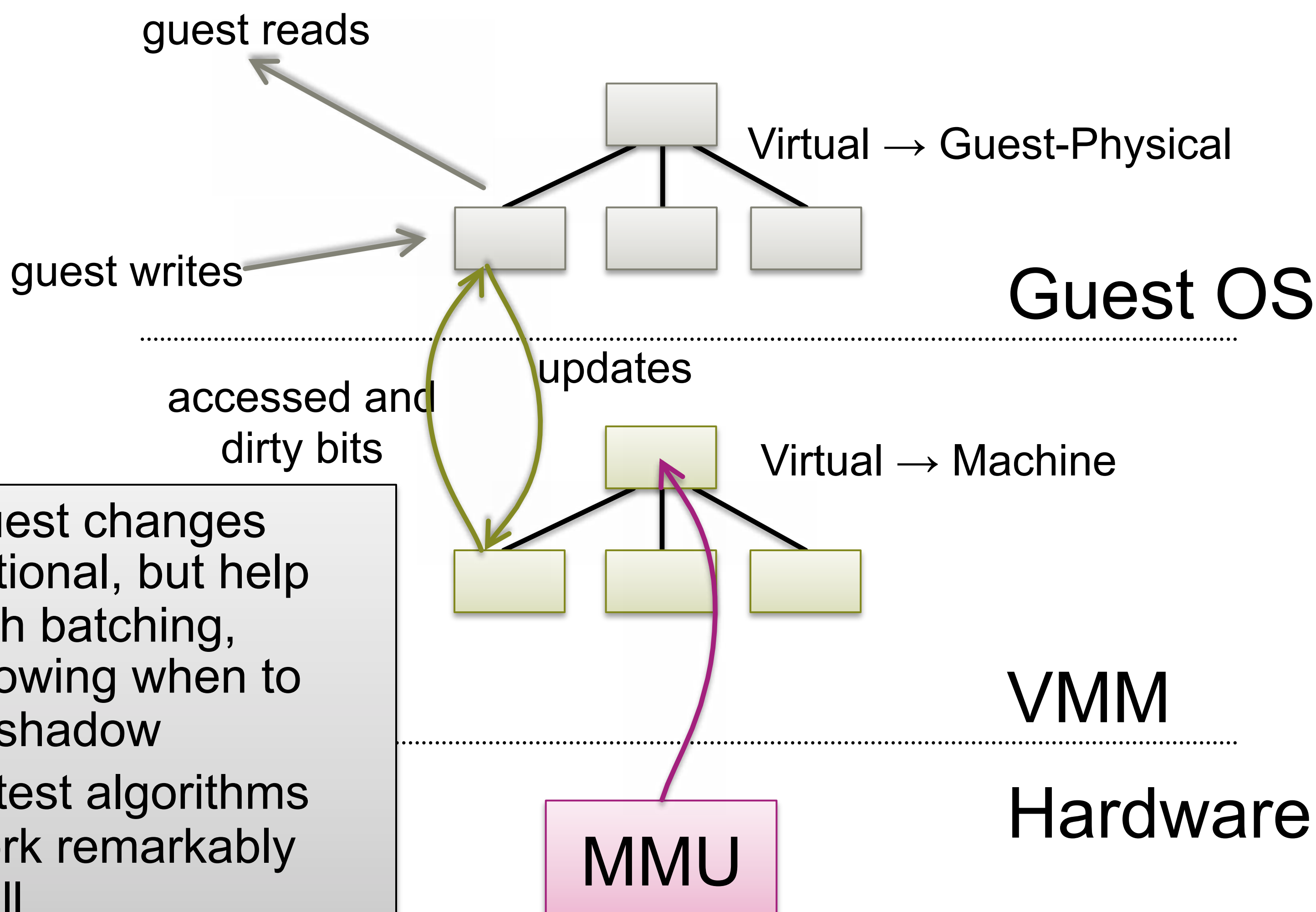
Shadow page tables

- **Guest OS sets up its own page tables**
  - Not used by the hardware!
- **VMM maintains *shadow page tables***
  - Map directly from Guest VAs to Machine Addresses
  - Hardware switched whenever Guest reloads PTBR
- **VMM must keep V→M table consistent with Guest V→P table and it's own P→M table**
  - VMM write-protects all guest page tables
  - Write ⇒ trap: apply write to shadow table as well
  - Significant overhead!

Shadow page tables



Shadow page tables



- Guest changes optional, but help with batching, knowing when to unshadow
- Latest algorithms work remarkably well

Hardware support

- **"Nested page tables"**
  - Relatively new in AMD (NPT) and Intel (EPT) hardware
- **Two-level translation of addresses in the MMU**
  - Hardware knows about:
    V→P tables (in the Guest)
    P→M tables (in the Hypervisor)
  - Tagged TLBs to avoid expensive flush on a VM entry/exit
- **Very nice and easy to code to**
  - One reason kvm is so small
- **Significant performance overhead…**

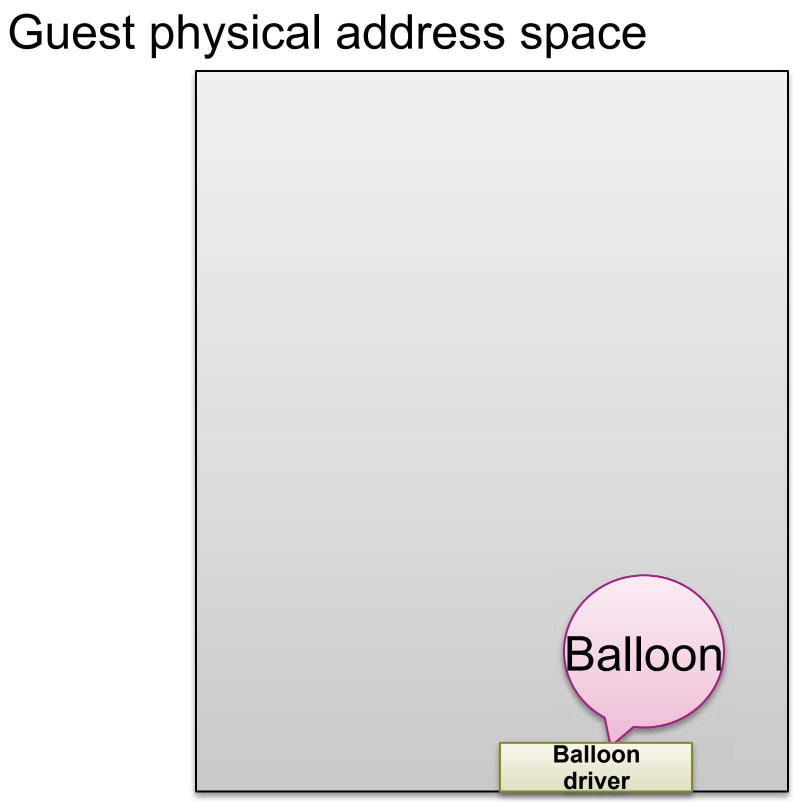
Memory allocation

- **Guest OS is not expecting physical memory to change in size!**
- **Two problems:**
  - Hypervisor wants to overcommit RAM
  - How to reallocate (machine) memory between VMs
- **Phenomenon: Double Paging**
  - Hypervisor pages out memory
  - GuestOS decides to page out physical frame
  - (Unwittingly) faults it in via the Hypervisor, only to write it out again

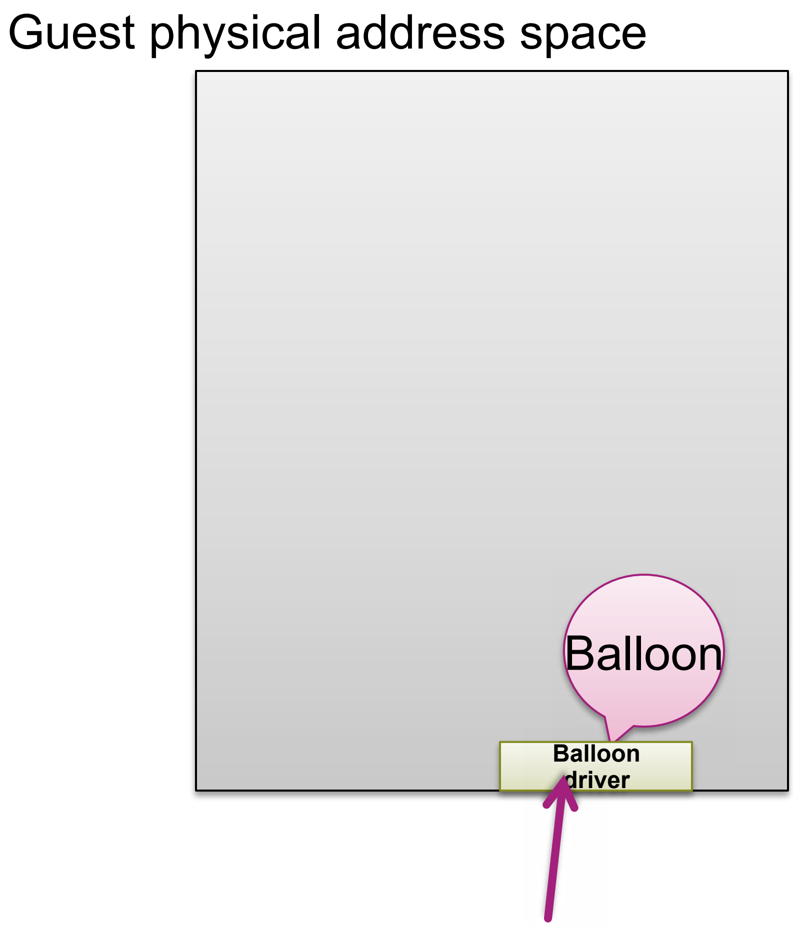
Ballooning

- **Technique to reclaim memory from a Guest**
- **Install a "balloon driver" in Guest kernel**
  - Can allocate and free kernel physical memory
    Just like any other part of the kernel
  - Uses HyperCalls to return frames to the Hypervisor, and have them returned
    Guest OS is unware, simply allocates physical memory

Ballooning: taking RAM away from a VM

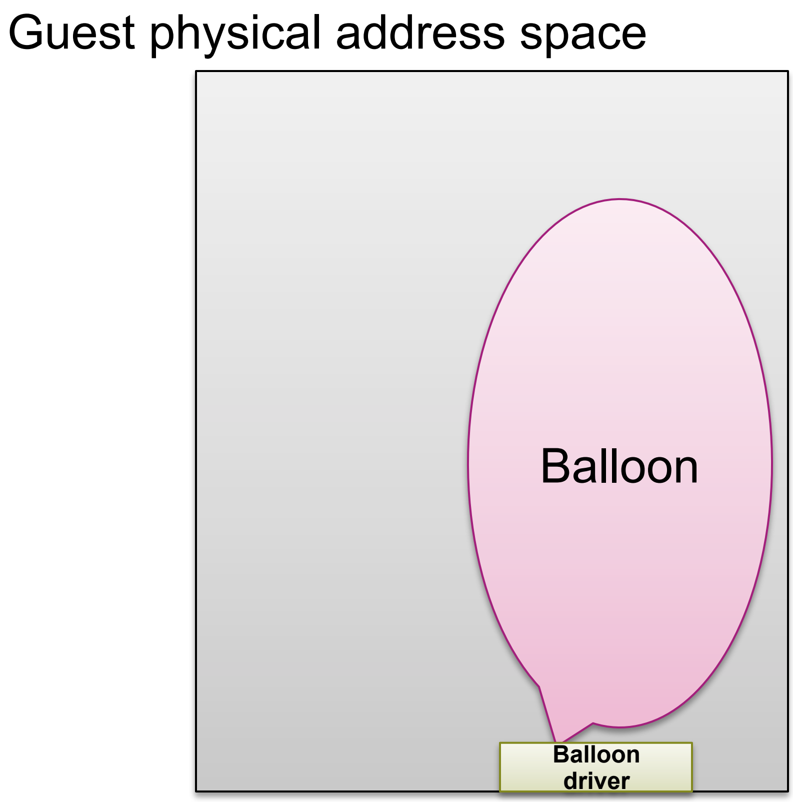


Ballooning: taking RAM away from a VM



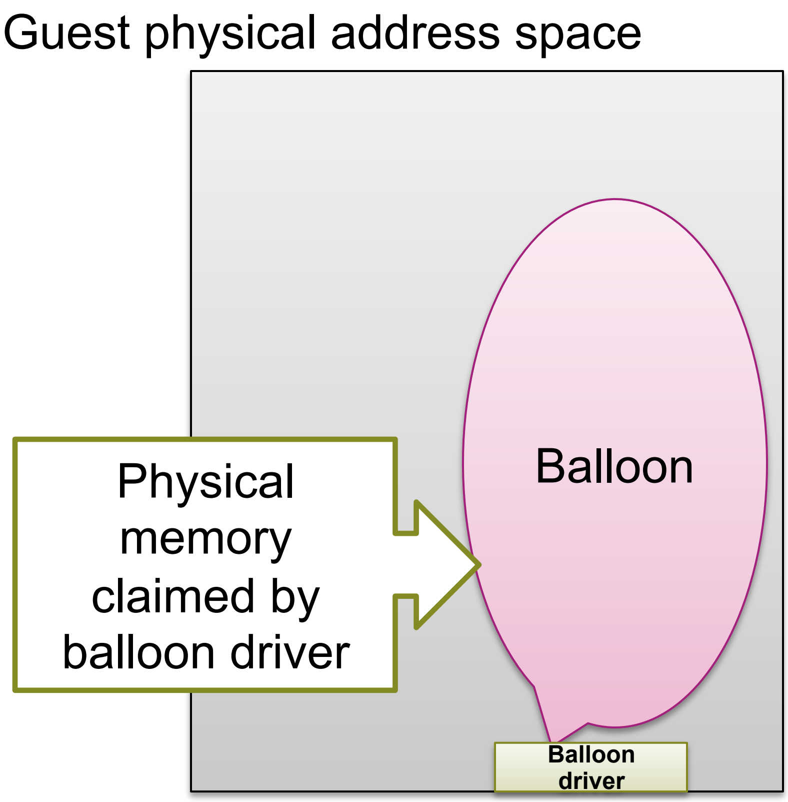
1. **VMM asks balloon driver for memory**
2.
3.
4.

Ballooning: taking RAM away from a VM



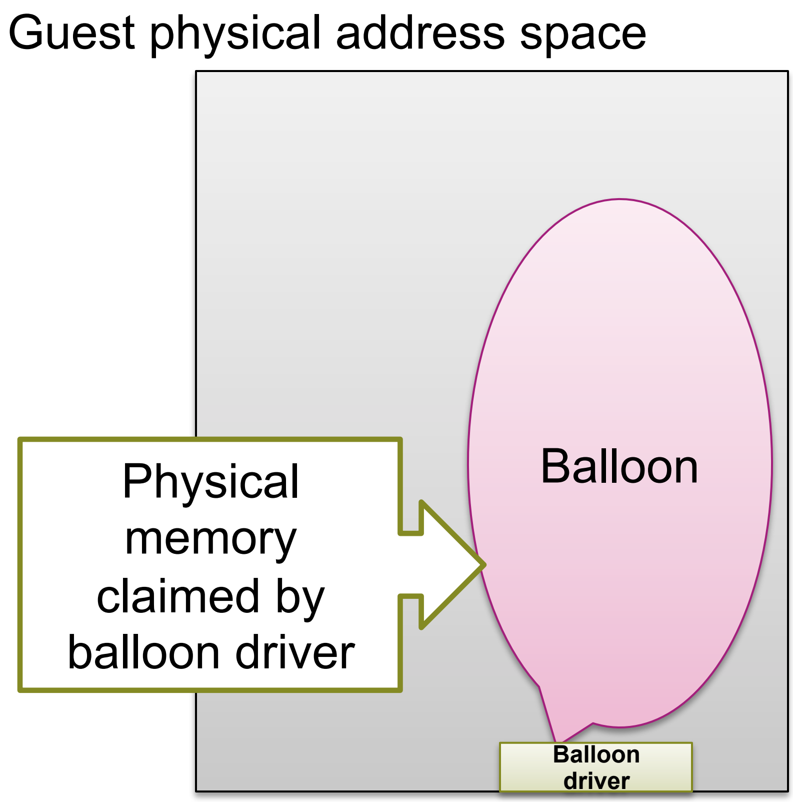
1. **VMM asks balloon driver for memory**
2. **Balloon driver asks Guest OS kernel for more frames**
   - "inflates the balloon"
3.
4.

Ballooning: taking RAM away from a VM



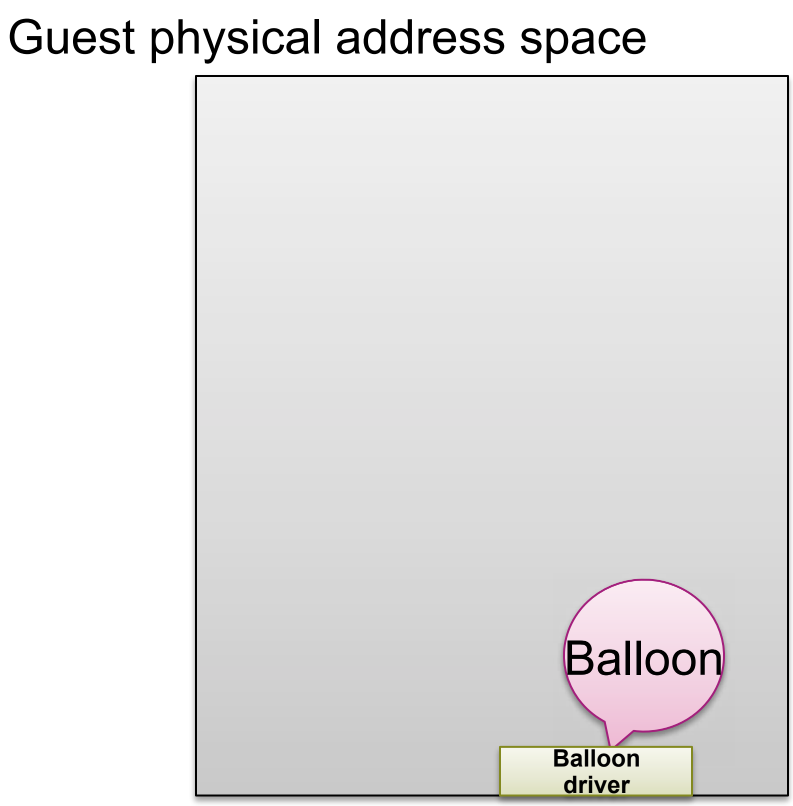
1. **VMM asks balloon driver for memory**
2. **Balloon driver asks Guest OS kernel for more frames**
   - "inflates the balloon"
3. **Balloon driver sends physical frame numbers to VMM**
4.

Ballooning: taking RAM away from a VM



1. **VMM asks balloon driver for memory**
2. **Balloon driver asks Guest OS kernel for more frames**
   - “inflates the balloon”
3. **Balloon driver sends physical frame numbers to VMM**
4. **VMM translates into machine addresses and claims the frames**

Returning RAM to a VM



1. **VMM converts machine address into a physical address previously allocated by the balloon driver**
2. **VMM hands PFN to balloon driver**
3. **Balloon driver frees physical frame back to Guest OS kernel**
   - "deflates the balloon"

Virtualizing Devices

- **Familiar by now: trap-and-emulate**
  - I/O space traps
  - Protect memory and trap
  - "Device model": software model of device in VMM
- **Interrupts → upcalls to Guest OS**
  - Emulate interrupt controller (APIC) in Guest
  - Emulate DMA with copy into Guest PAS
- **Significant performance overhead!**

Paravirtualized devices

- **“Fake” device drivers which communicate efficiently with VMM via hypercalls**
  - Used for block devices like disk controllers
  - Network interfaces
  - “VMware tools” is mostly about these

- **Dramatically better performance!**

Networking

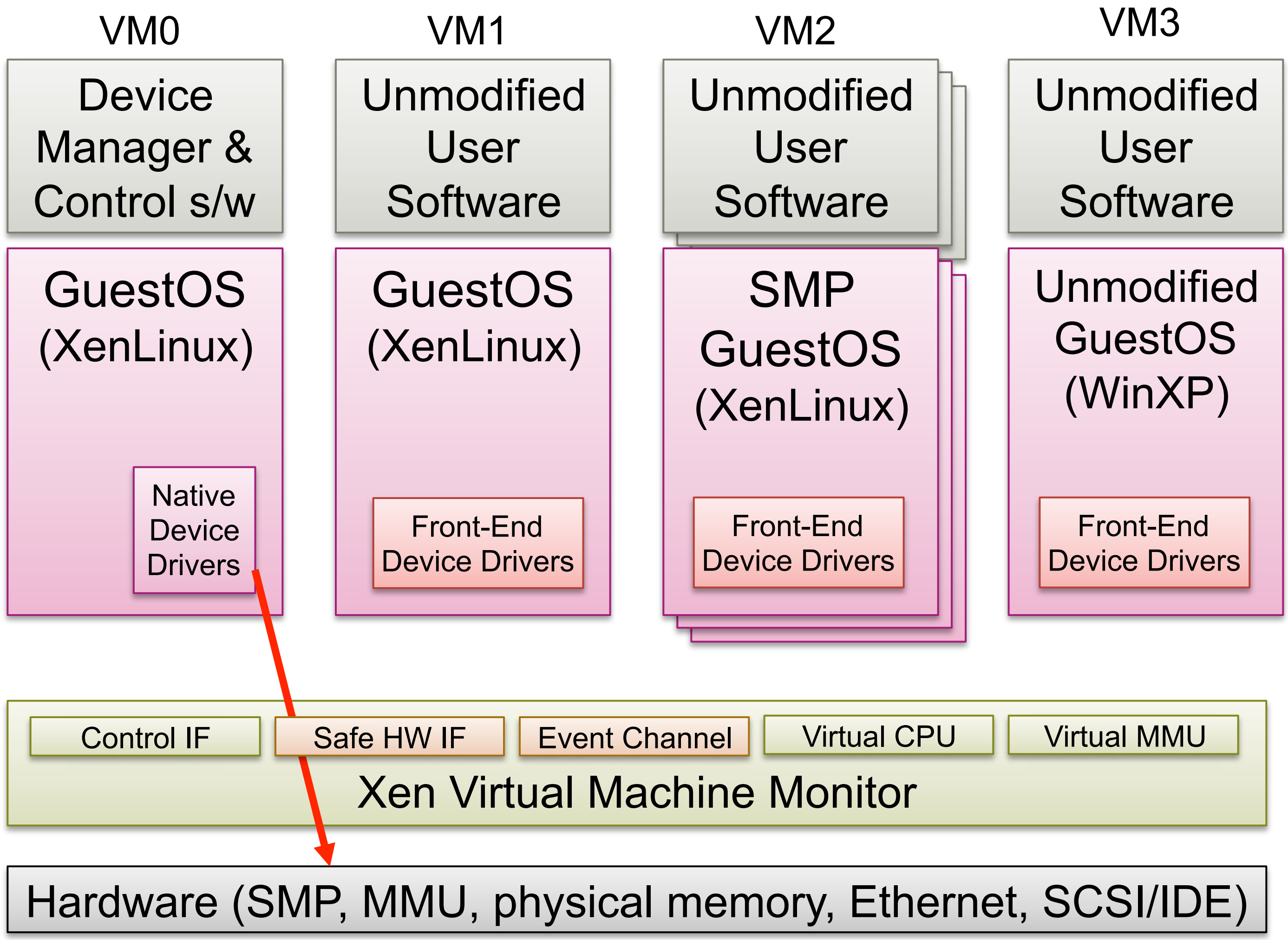
- **Virtual network device in the Guest VM**
- **Hypervisor implements a "soft switch"**
  - Entire virtual IP/Ethernet network on a machine
- **Many different addressing options**
  - Separate IP addresses
  - Separate MAC addresses
  - NAT
- **Etc.**

Where are the real drivers?

1. **In the Hypervisor**
   - E.g. VMware ESX
   - Problem: need to rewrite device drivers (new OS)
2. **In the console OS**
   - Export virtual devices to other VMs
3. **In "driver domains"**
   - Map hardware directly into a "trusted" VM

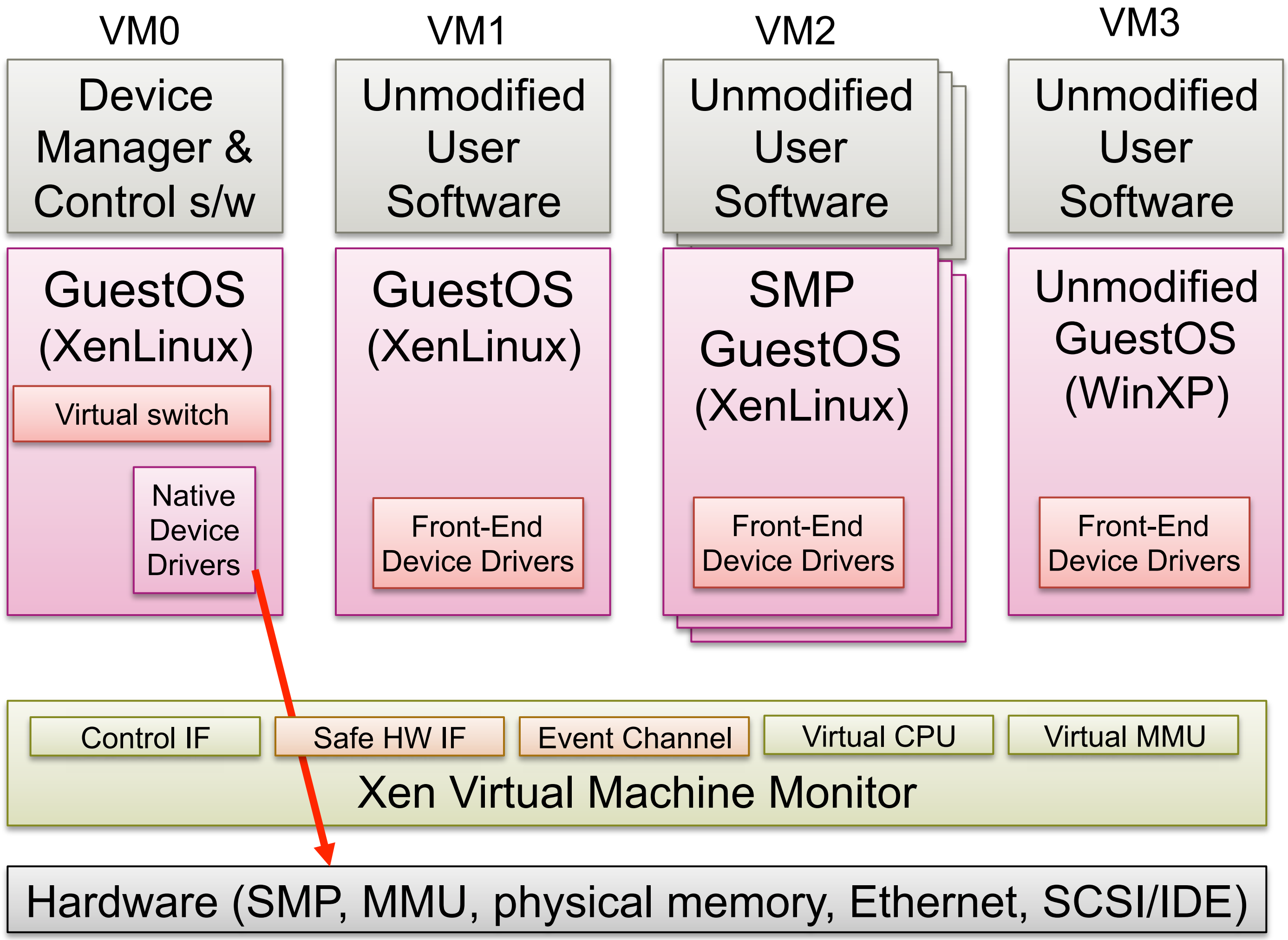
     Device Passthrough
   - Run your favorite OS just for the device driver
   - Use IOMMU hardware to protect other memory from driver VM
4. **Use "self-virtualizing devices"**

Xen 3.x Architecture



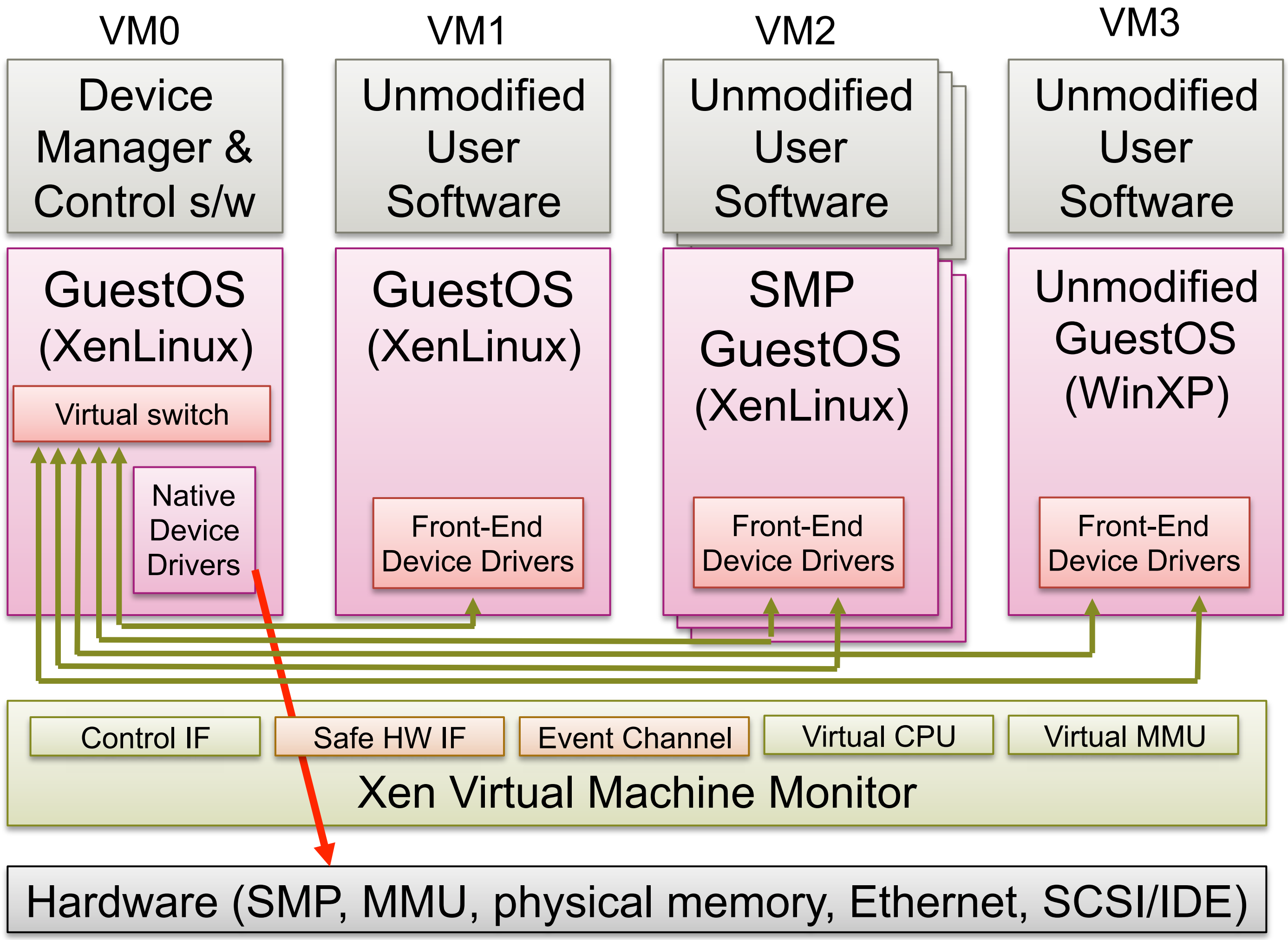
Thanks to Steve Hand for some of these diagrams

Xen 3.x Architecture



Thanks to Steve Hand for some of these diagrams

Xen 3.x Architecture



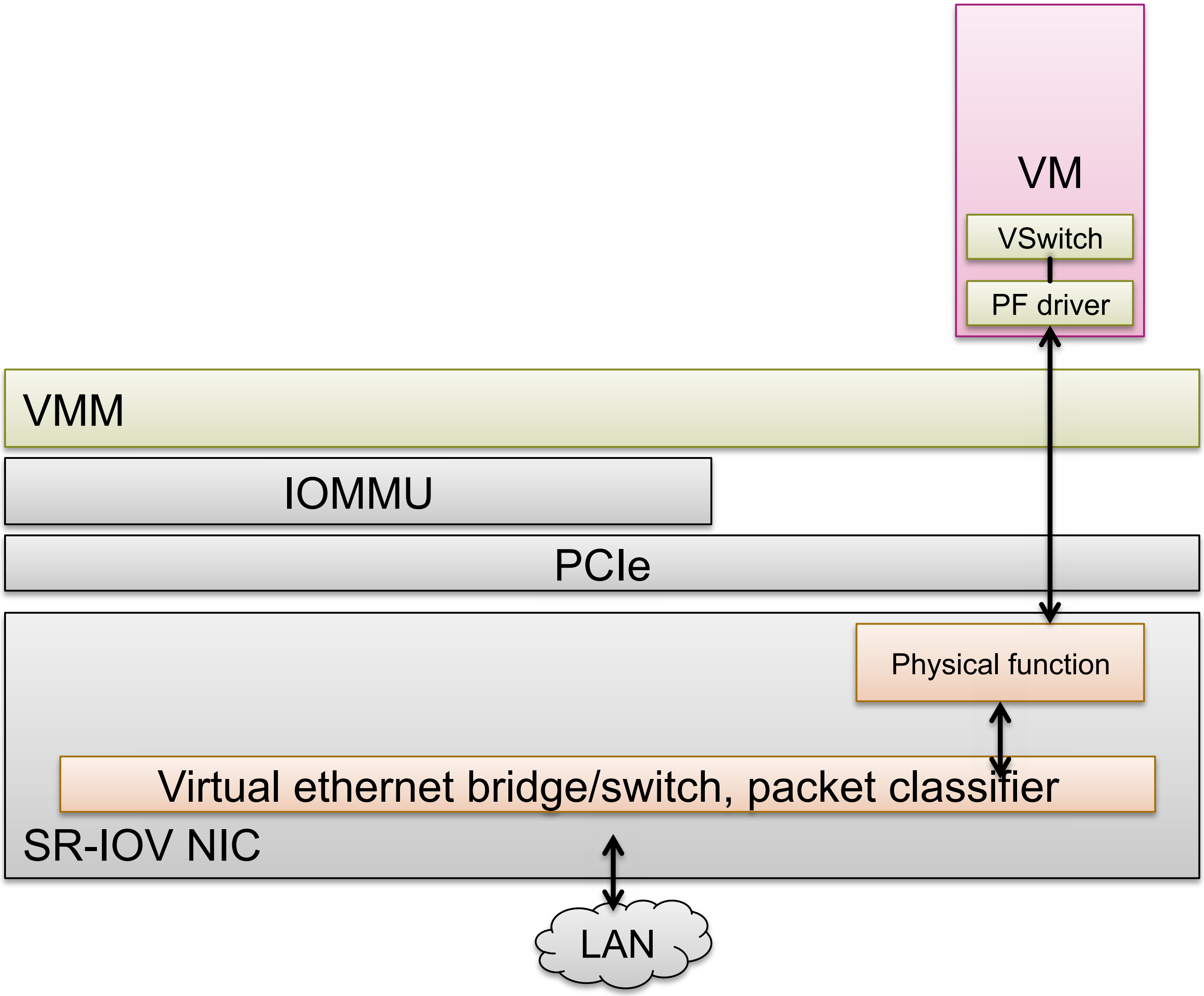
Thanks to Steve Hand for some of these diagrams

Remember this card?

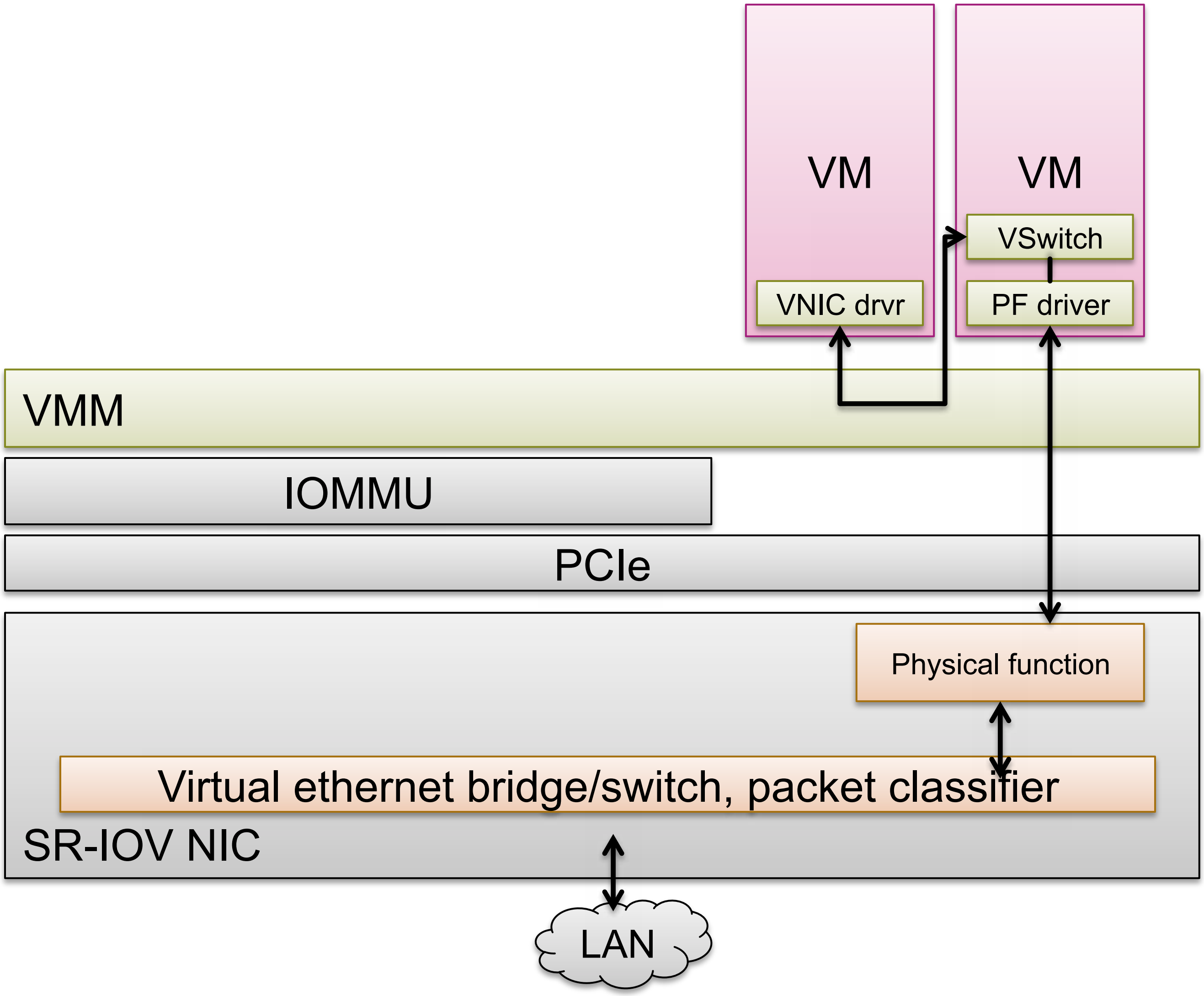
SR-IOV

- **Single-Root I/O Virtualization**
- **Key idea: dynamically create new “PCIe devices”**
  - Physical Function (PF): original device, full functionality
  - Virtual Function (VF): extra “device”, limited funtionality
  - VFs created/destroyed via PF registers
- **For networking:**
  - Partitions a network card’s resources
  - With direct assignment can implement passthrough

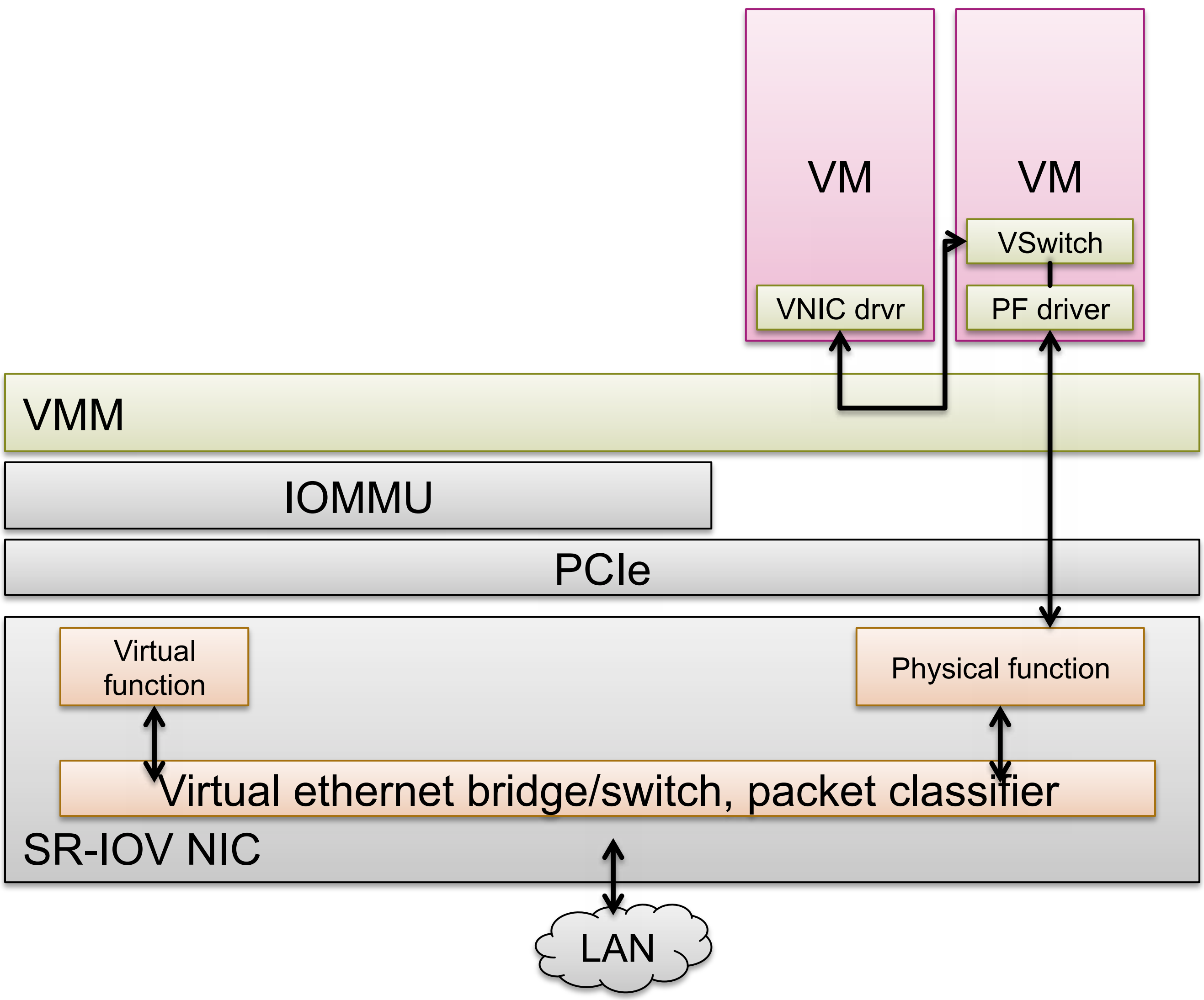
SR-IOV in action



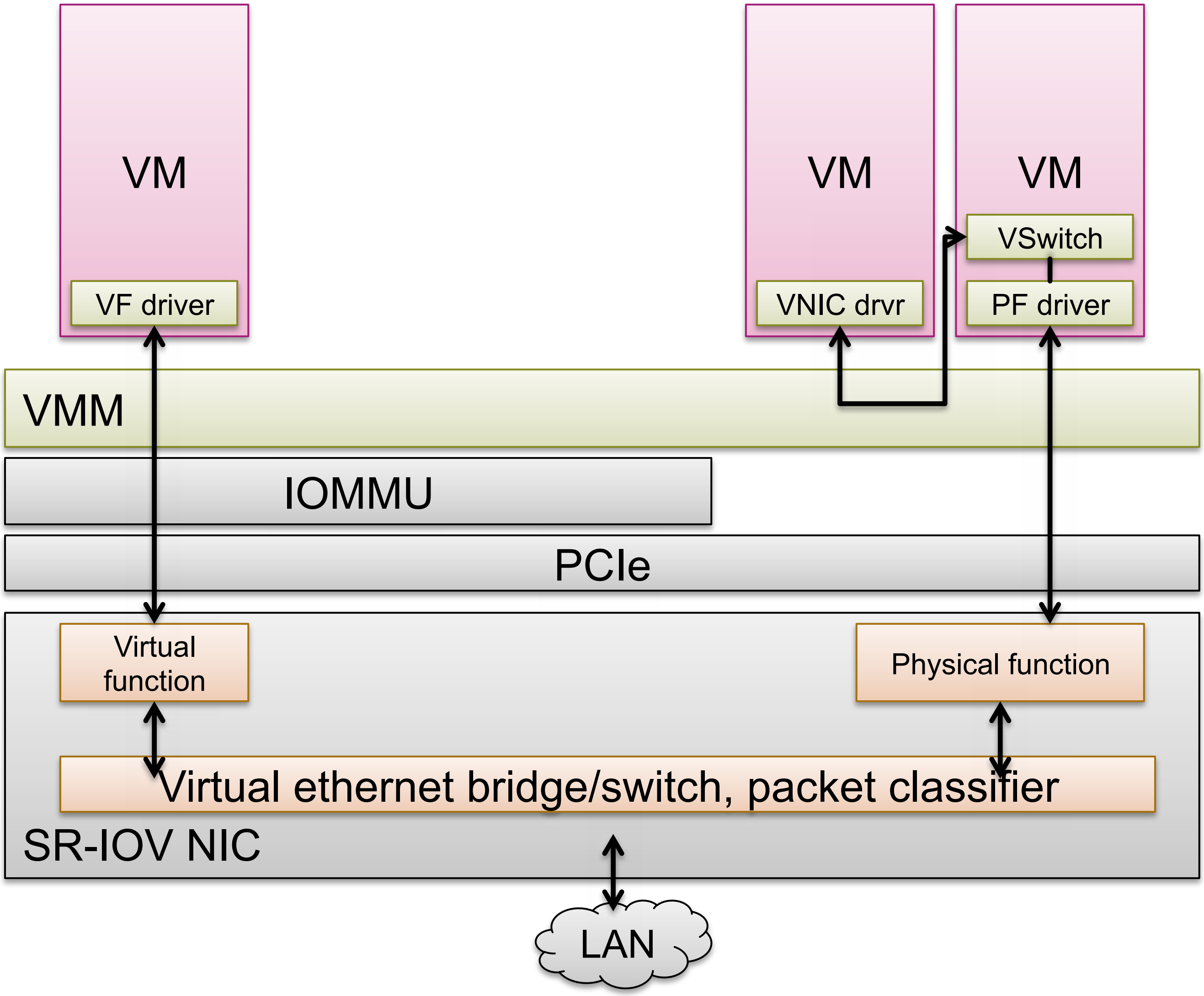
SR-IOV in action



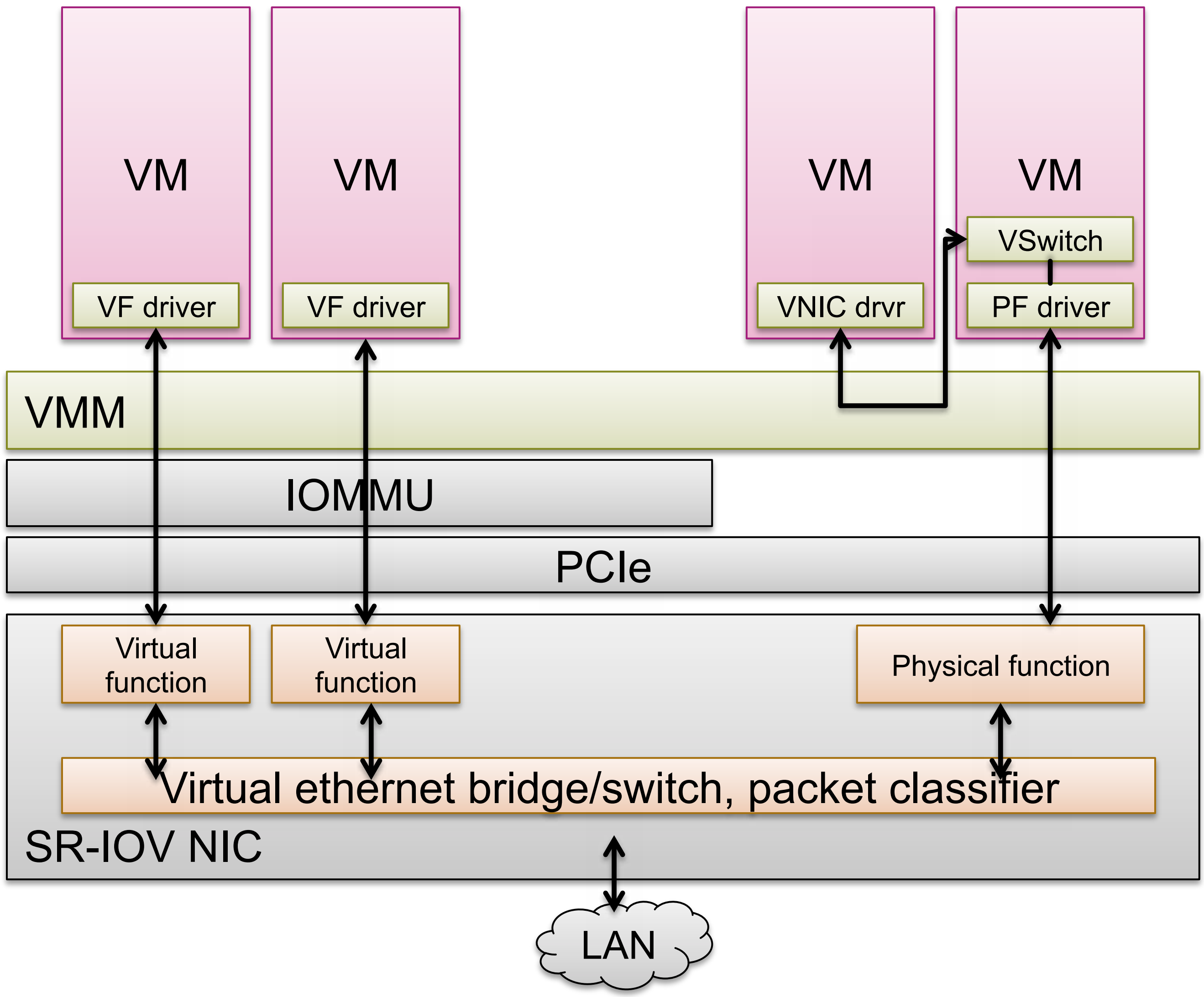
SR-IOV in action



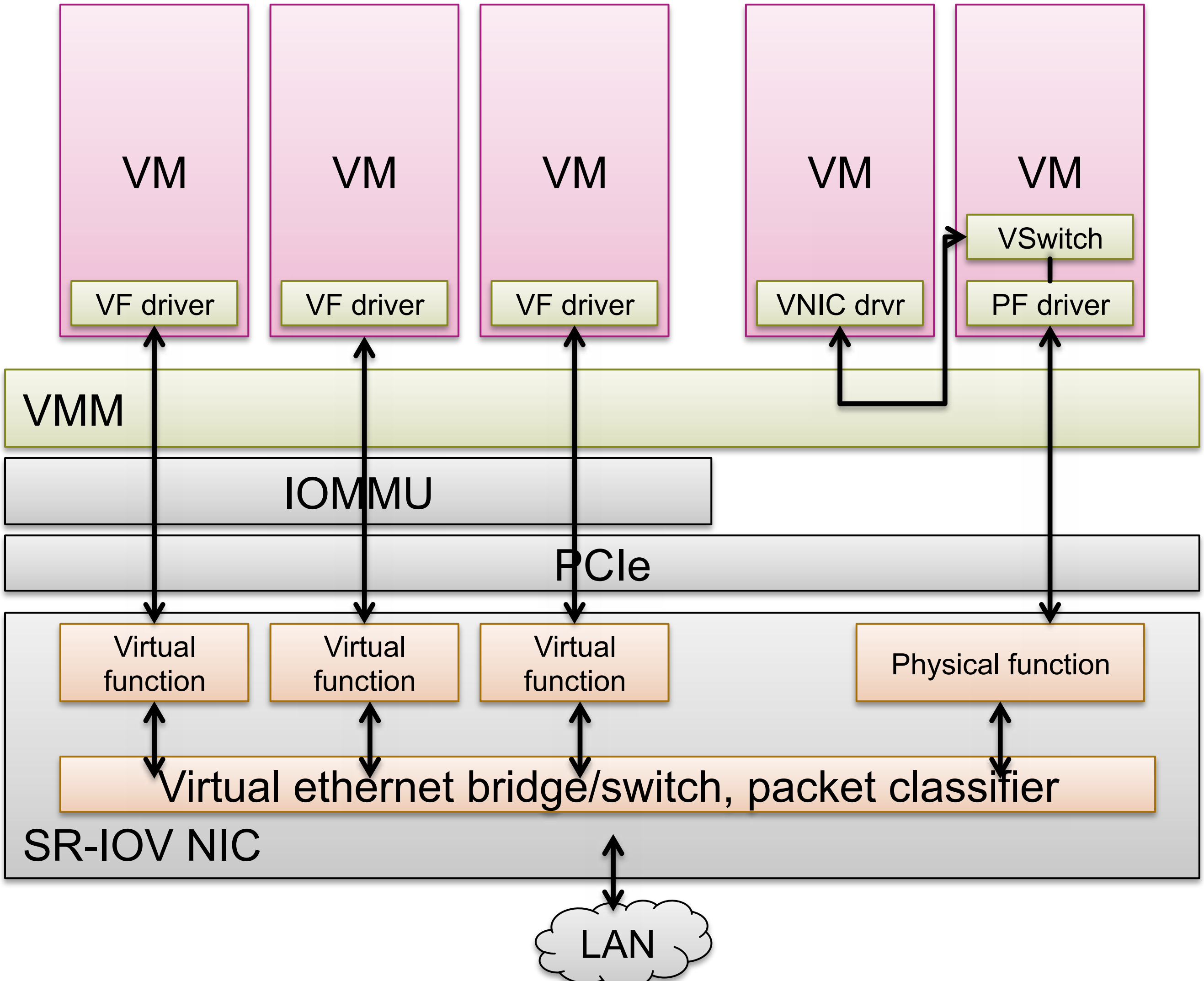
SR-IOV in action



SR-IOV in action



SR-IOV in action



Self-virtualizing devices

- **Can dynamically create up to 2048 distinct *PCI devices* on demand!**
  - Hypervisor can create a virtual NIC for each VM
  - Softswitch driver programs "master" NIC to demux packets to each virtual NIC
  - PCI bus is virtualized in each VM
  - Each Guest OS appears to have "real" NIC, talks direct to the real hardware

Next week

Reliable storage

OS Research/Future™