Hardware and Software Support for Virtualization

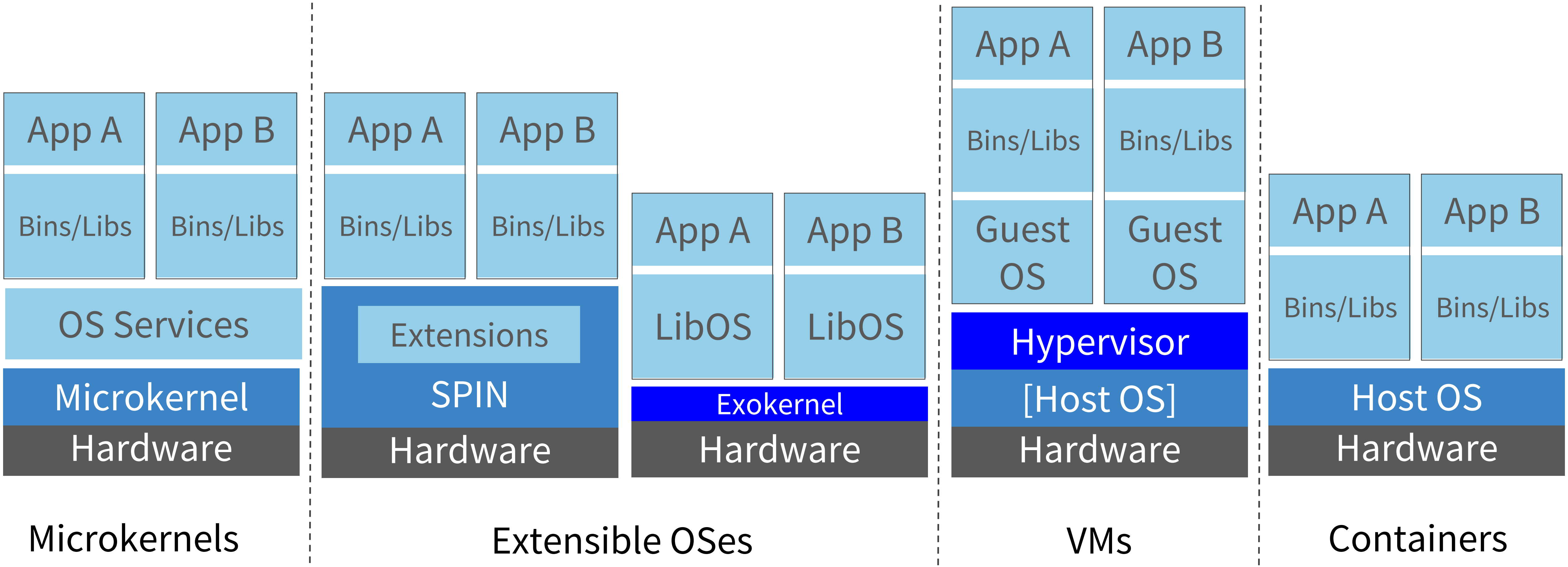
Emmett Witchel

CS380L

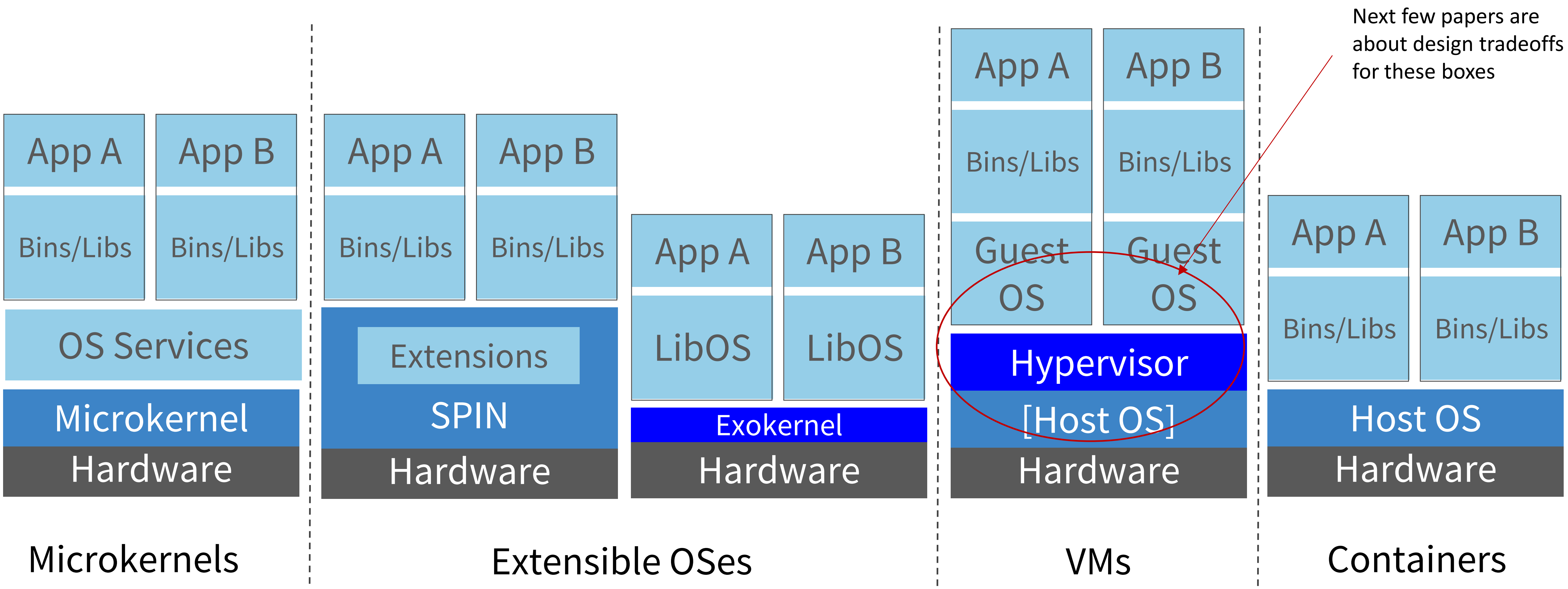
Xen faux quiz (pick 2, 5 min)

- What is the difference between an API and an ABI?
- Why are fork and exec slow in Xen?
- What is page coloring?
- What is the difference between a hypercall and a system call?
- Does Xen require device drivers in the hypervisor? Why/why not?
- Does Xen trap guest system calls? Why/why not?
- What policy does Xen use to allocate memory across domains? What advantages/disadvantages does this have?
- Why are HW physical->machine mappings readable by all VMs in Xen?
- What is the “double paging” problem?
- How does a memory balloon work? What happens when a guest OS writes to memory owned by the balloon driver?
- How do Xen memory virtualization techniques differ from ESX?
- Compare and contrast interposition techniques in ESX, Xen, Arrakis

Box drawing Potpourri: OSes, VMs, Containers



Box drawing Potpourri: OSes, VMs, Containers



Why virtualize hardware?

- Programs for one OS difficult to run on another OS.
  - Wine (winehq.org) [MLOC = millions of lines of code]
    - started in 1993, beta in 2005 / v1.0 in 2008 at 1.4 MLOC / 2014 → 2.6 MLOC
- Ever try installing two different PostgreSQL versions?
  - Shared libraries, configuration files, etc.
- But the hardware interface relatively stable.
- Virtualizing the hardware →run unmodified application with its OS.
  - Run unmodified applications (same *ABI*) from different OSes.
  - Performance isolation. OS don't cut it (QoS cross-talk).
  - Accounting: sell part of a physical machine (isolation).
  - ***Compatibility:*** VMMs have always presented a very appealing platform for practical deployment, [because they] [allow] users to securely share hardware on machines at a low performance cost, [improve] machine utilization, and [don't require] modifications to the applications. —Steven Hand
- Multiplexing, aggregation, emulation

Precisely, what is ‘Virtualization?’

- Popek & Goldberg 1974: VMM properties
  - **Equivalence/Fidelity**

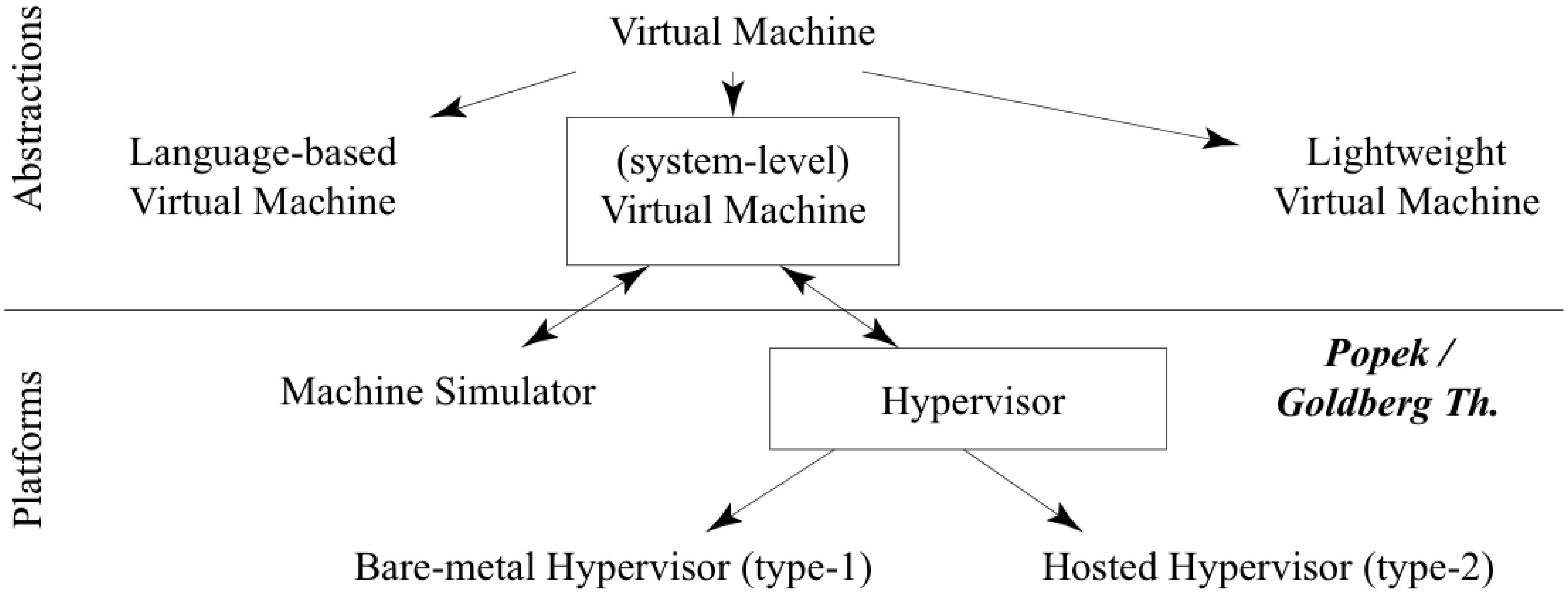
    A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
  - **Resource Control / Safety**

    The VMM must be in complete control of the virtualized resources
  - **Efficiency / Performance**

    A statistically dominant fraction of machine instructions must be executed without VMM intervention
- Bugnion, Tsafrir, Nieh 2017:

  Virtualization is the encapsulation pattern used to present the same interface as the encapsulated resource
- What is a VM vs VMM? VMM vs Hypervisor?

Types of virtual machines



- Virtual machine is an overloaded term. Know where you are.

Virtual Machine: a simulator mental model

```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
   machine.Registers[i.dst] += machine.Registers[i.src];
   break;
}
```

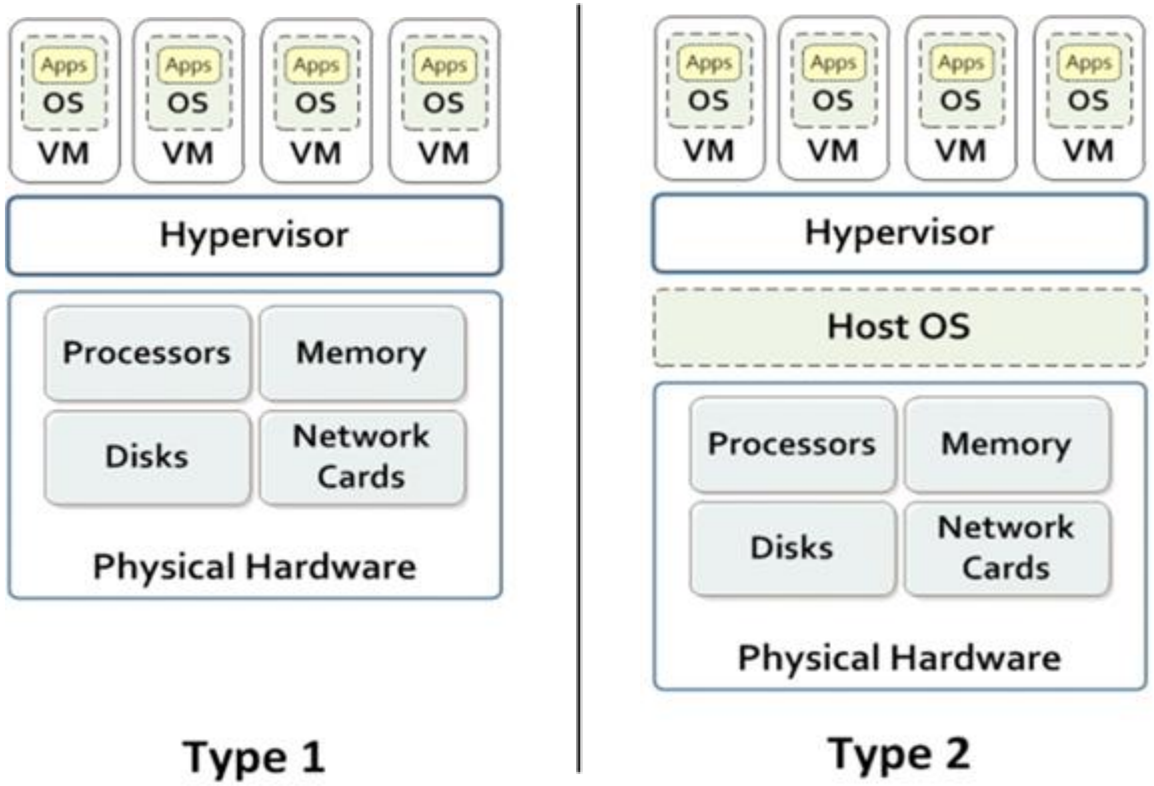
What interface is encapsulated here?
How?

Virtualization challenges

- Instructions (VT-x)
  - Virtual machine control structure (VMCS)
  - Unsafe instructions trap (VMExits expensive)
- Memory (extended page tables)
  - Hypervisor does to OS what OS does to user
- Devices (VT-d)
  - Software-defined devices
  - IOMMU (page table for devices)
  - Hardware support for virtualization (SR-IOV)

VMM Classification: Type 1 vs Type 2

- VMM implemented directly on physical hardware
- VMM performs scheduling and allocation of system's resources
- E.g., IBM VM/370, Disco, Xen, ESX Server



- VMMs built completely on top of a host OS
- Host OS provides resource allocation and standard execution environment to each "guest OS"
- KVM, User-mode Linux (UML), ESX Workstation

What makes hardware hard to virtualize?

- Direct access to physical memory
  - MIPS allows OS to access physical memory at a fixed virtual address
- Instructions that act differently at different privilege levels
  - popf, iret
- Unprivileged instructions that access privileged state
  - sgdt, sldt,
- Excessive exits to hypervisor due to difficult to virtualize instructions
  - int 0x80, a software interrupt was x86’s syscall instruction
- x86-32 segment state

Challenges for x86

- How to virtualize an ISA?
  - Generic challenges: instructions (VT-x), MMU (EPT), devices (VT-d).
  - Classical virtualization (IBM 370) used hardware.
  - Trap and emulate too slow for many architectures
  - System calls and page faults are frequent
  - Software emulation considered too slow.
- x86 challenges
  - The EFLAGS register has the interrupt enable bit.
    - If the kernel is being virtualized, it is not privileged to enable interrupts.
    - The kernel calls pushf and popf all over the place, and no instance can enable interrupts.
  - CR3 points to the base of the page table: VMM can't trust OS to write page tables.
  - Untagged TLB → frequent flushes.
  - Solved by binary translating the kernel (Disco 1997), currently solved by VT-x

Xen conclusion: full virtualization not a good tradeoff

Virtualization: Techniques & Tradeoffs

	Performance	Fidelity	Compatibility	Interposition	Complexity
Full virtualization *(Device emulation, HW Virtualization)*					
API remoting *Forwards API calls to proxy (e.g. dom0, proxy VM)*					
Paravirtualization *Adapt Guest OS or apps*					

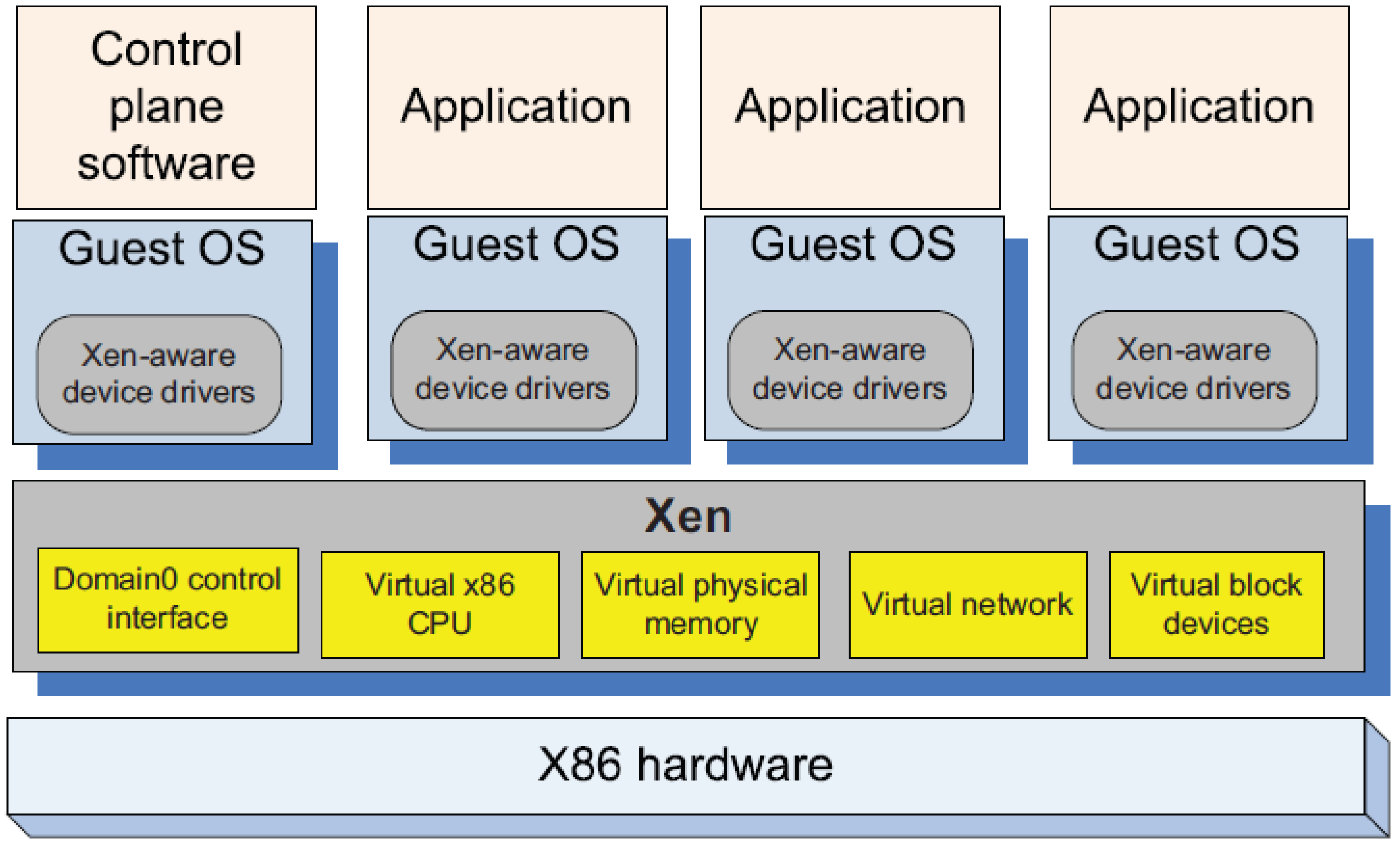
Paravirtualization: goals

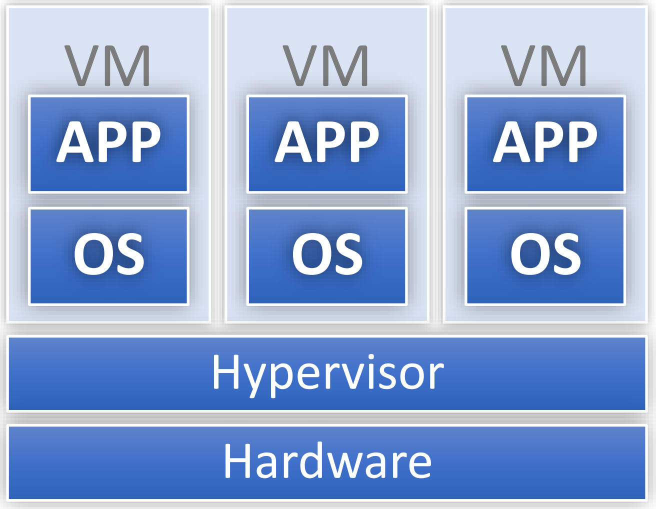
- Paravirtualization
  - idealized machine, efficient to virtualize.
  - More efficient than “full” virtualization
  - Low cost of porting an OS (weak point).
- Still need safety
  - hypervisor → portion of PA space that the guest OS cannot access
  - top 64MB, use segmentation to avoid TLB flushes.
- “*Typically only two types of exception occur frequently enough to affect system performance: system calls (which are usually implemented via a software exception), and page faults.*”

Paravirtualization: techniques

- Small changes to the OS
  - Explicit hypercalls into the hypervisor
    - Replace privileged instructions with hypercalls
    - Changed syscall instruction. (In 2000, **`int 0x80`** was replaced by **`sysenter`** in hardware)
  - Batch updates to page tables
- Shadow paging
  - Guest: VA->Guest PA
  - Hypervisor uses its own Guest PA->Host PA maps
  - Installs VA->Host PA into TLB
- Use a “system VM” for complex functionality
  - Keeps hypervisor simple
  - Domain 0 (Dom0) does things like loads the real device drivers
  - Guest OS loads a Xen-aware driver that talks to Dom0

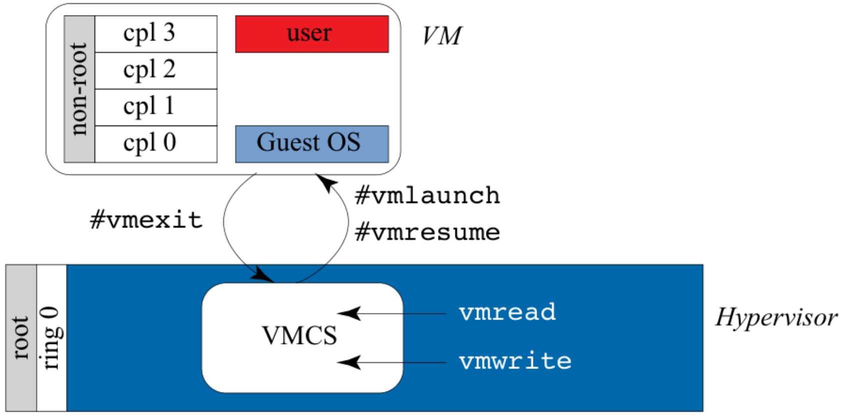
Xen System Architecture



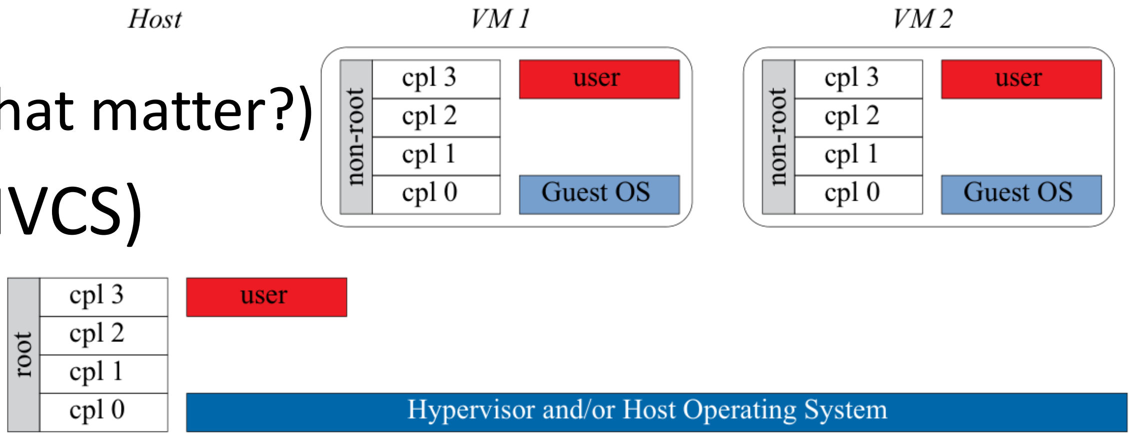


Type I / bare metal organization

VT-x: Virtualizing the CPU

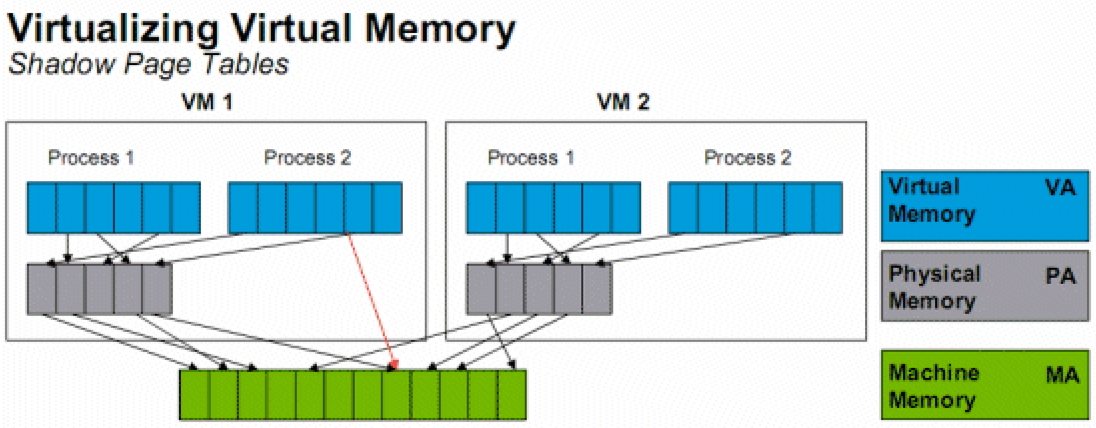


- Duplicate all architecturally visible state
  - root mode (hypervisor), non-root mode (guest)
    - Many kernel instructions (e.g., cli) work in non-root mode on non-root state
    - Instructions to access global descriptor table are privileged in root mode
    - vmcall enters root mode like sysenter enters the kernel
  - Transitions atomic: require 1 instruction, includes TLB state
    - This is expensive (~780 cycles)!
    - Must minimize VMexits
  - root mode is virtualizable! (why does that matter?)
- Virtual machine control structure (MVCS)
  - State of VM held in memory



Virtualizing memory

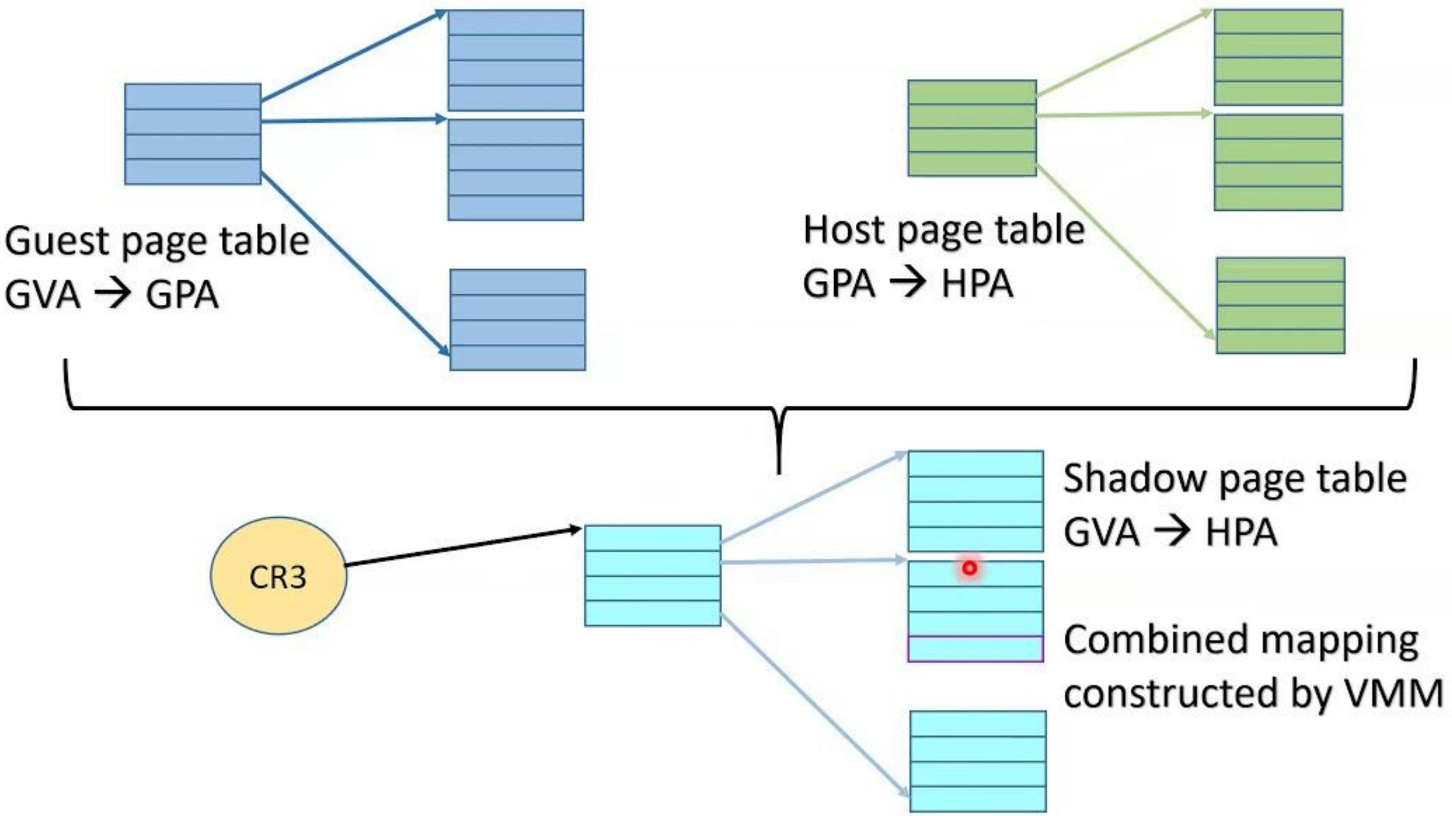
- No hardware support
  - Shadow page tables
  - Guest page tables are read-only, so trap on write
  - Hypervisor validates mappings, installs VA->Host PA
- Hardware support
  - Extended page tables (EPT) Intel
  - Nested page tables (NPT) AMD
  - Tell processor about guest and host page tables, let it do the work
    - Worst case 1 memory reference -> 24 memory references!



Shadow page tables

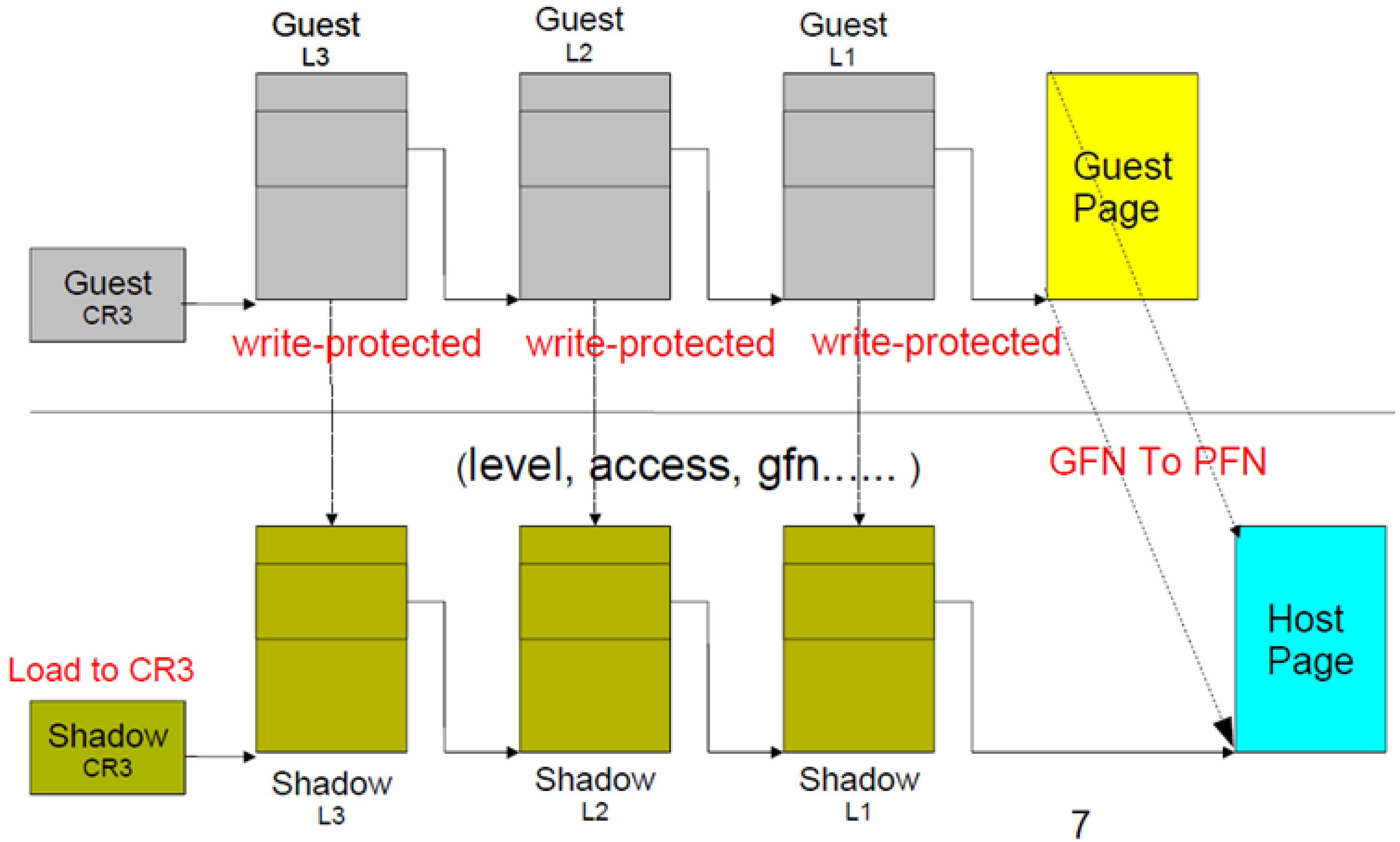
e tables

- Guest: VA->GPA
- Xen: GPA->HPA
- TLB: VA->HPA



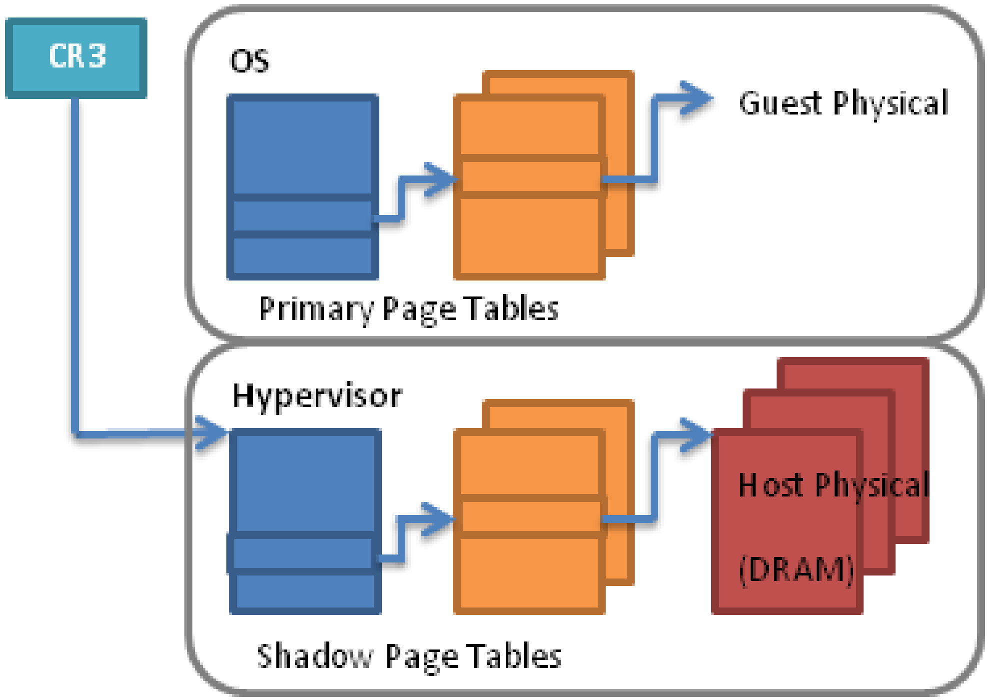
Virtual address translation

- Guest page tables write protected
- Guest PT updates cause VMexits
- VMexits are bad for performance



Shadow page tables

Is this fundamentally slow?
Why? / Why not?

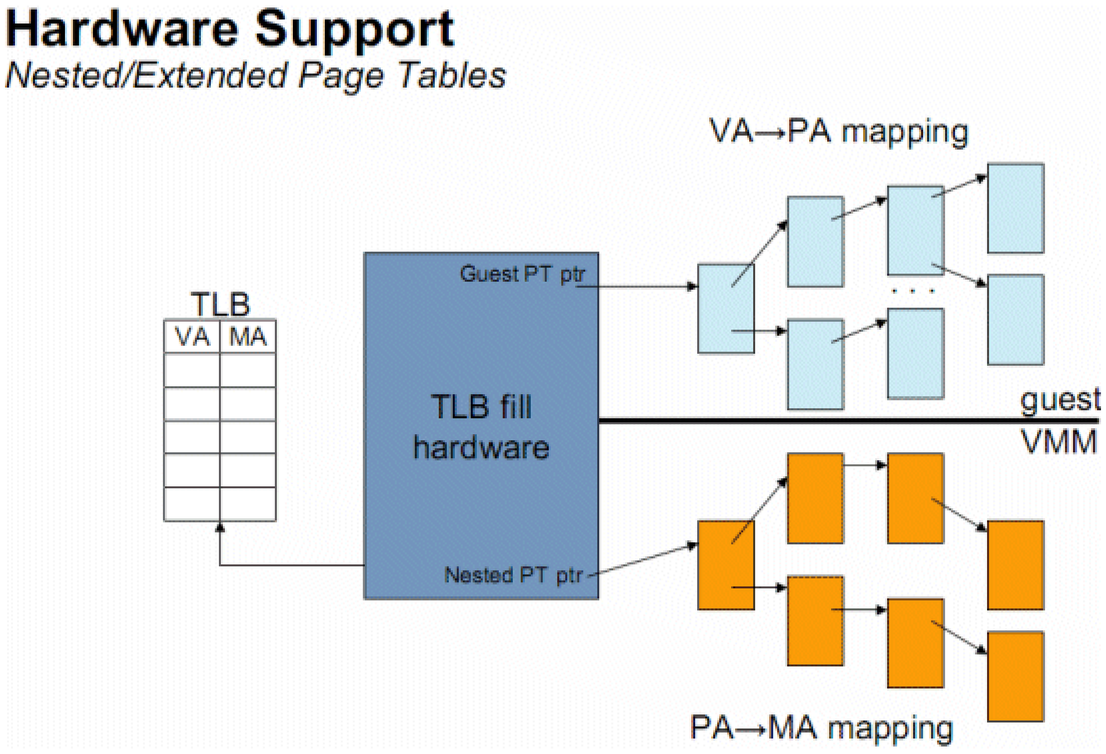


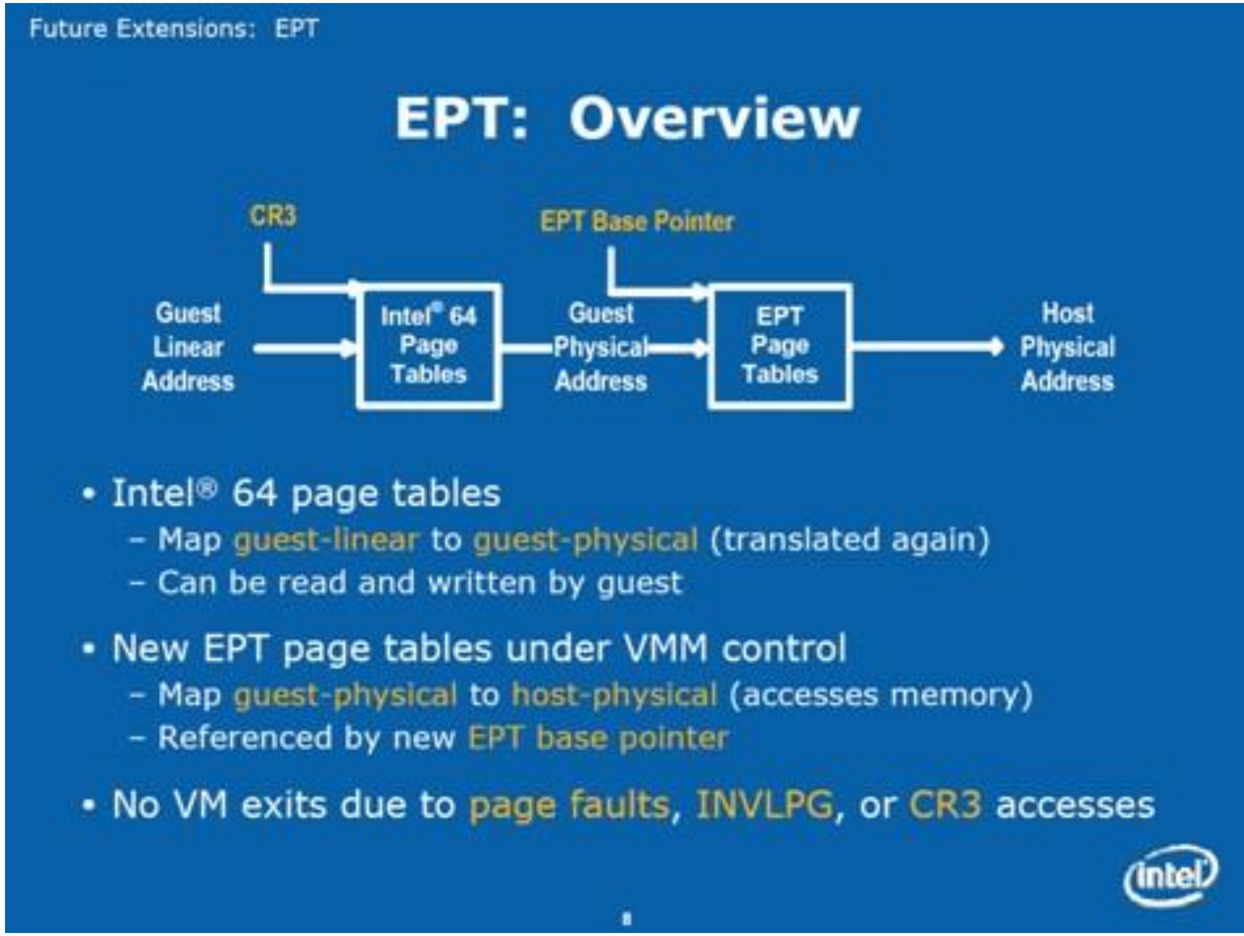
- Guest: ref unmapped
- HW: TRAP! Jump to VMM handler
- VMM: find guest OS, check shadow, setup trap regs for guest
- Guest: read cr2
- HW TRAP! Jump to VMM handler
- VMM: read cr2, write faulting address to OS reg
- Guest: alloc phys frame, write PTE
- HW: TRAP! (RO mem): → VMM handler
- VMM: alloc mem, record PA->MA, set shadow PTEs
- Guest: thinks all good, clear privilege bit, reti
- HW: TRAP! (privilege) → VMM handler
- VMM: reti to guest

Why is shadow paging slow?

- **Guest page-table writes cause traps (shadow) vs. run-through (NPT/EPT).**
  - Shadow paging must write-protect the guest's page tables.
    - Every guest PTE write triggers a fault → VM exit → hypervisor updates the corresponding shadow PTE(s) → resume.
  - With NPT/EPT the guest updates its own page tables without exits; the CPU later resolves translations using the nested tables.
- **Shadow coherence maintenance is expensive.**
  - The hypervisor must keep multiple **shadow** page tables consistent with the guest's view (per address space / per vCPU variants, global pages, split large pages, etc.).
  - Any change (CR3 load, INVLPG, context switch, page reclaim) can force shadow rebuilds, shootdowns, and extra TLB flushes—each a VM exit and cross-CPU IPI.
- **Accessed/Dirty (A/D) and permissions emulation.**
  - In shadow mode the CPU sets A/D bits in the **shadow** PTEs, not the guest's. Synchronizing those bits back to the guest view (or emulating them) requires extra exits and bookkeeping.
  - NPT/EPT expose hardware A/D and fine-grained permissions in the nested tables; no emulation round-trips are needed.

Nested page tables

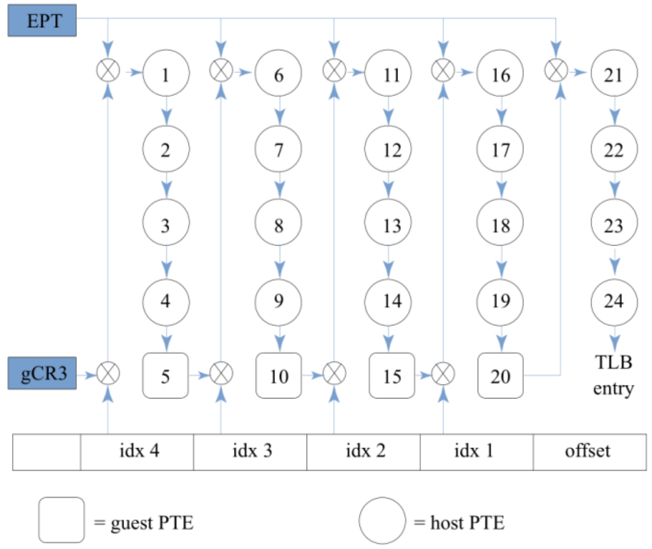


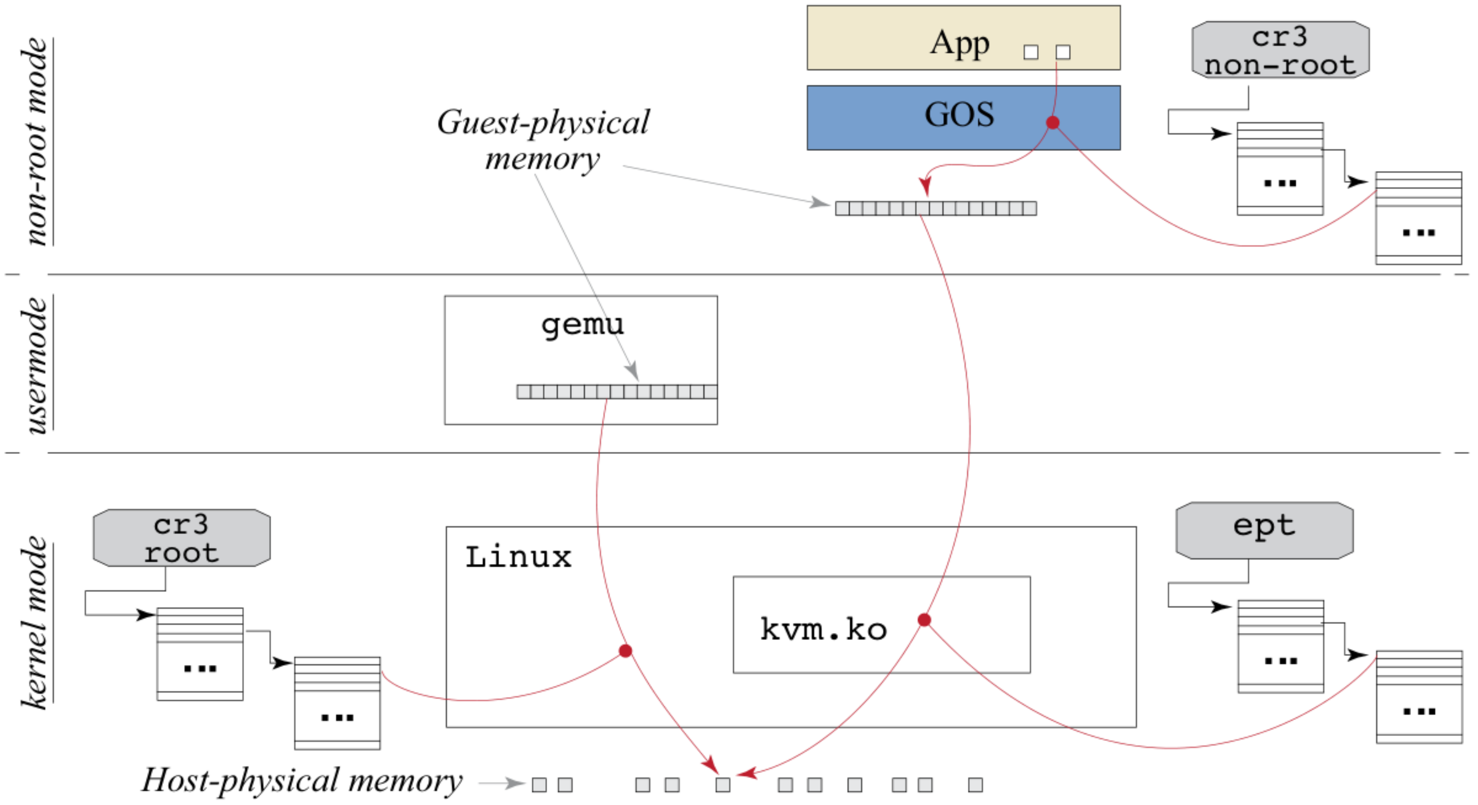


- No VMExit for guest PT writes

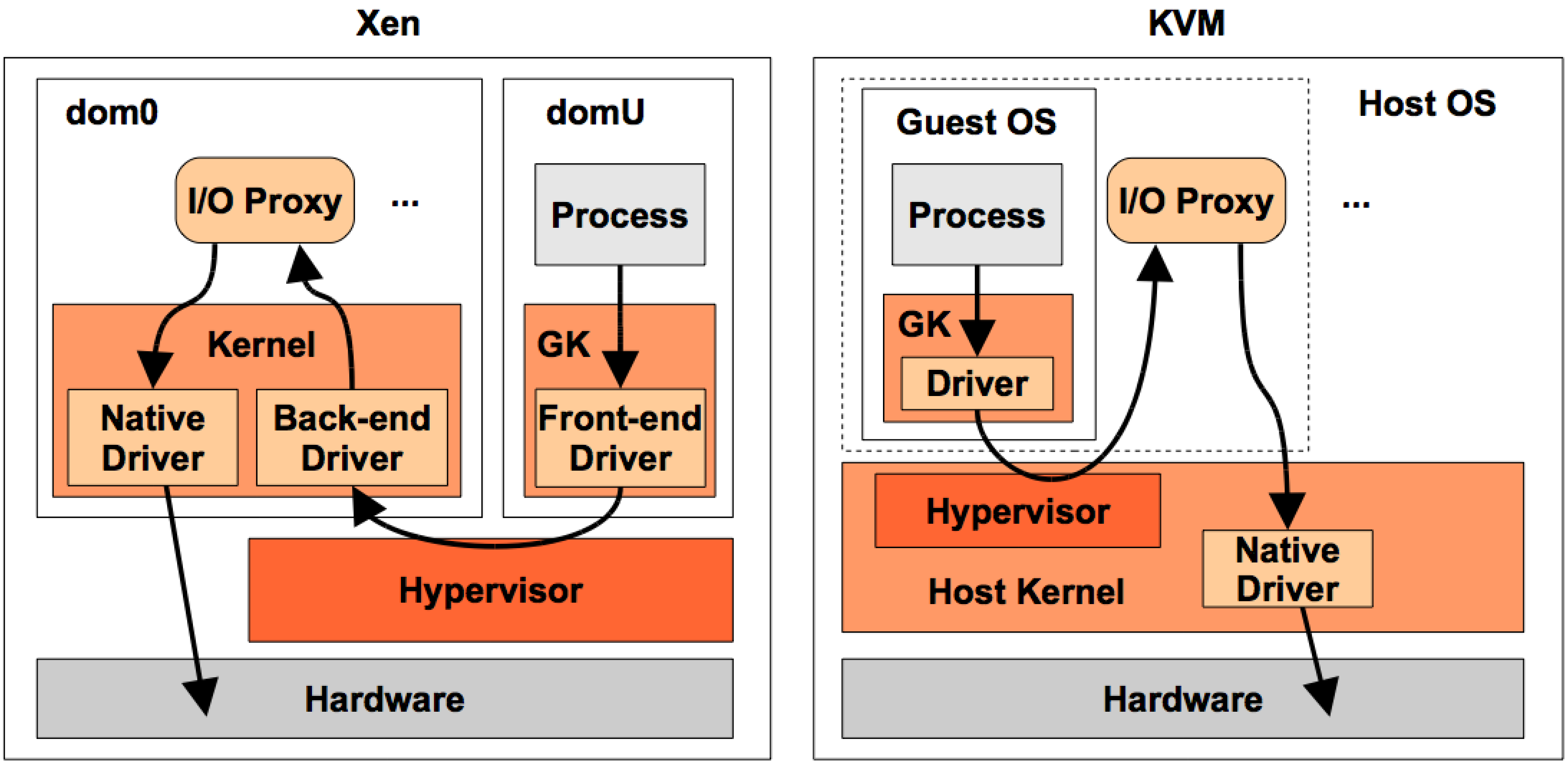
Nested page tables

- Worst case: 24 memory references to translate virtual page
  - Pages larger than 4KB are important
- qemu allocates host memory in 1 chunk
  - Host OS in control
  - qemu devices can access
- Host swapping qemu memory is complicated



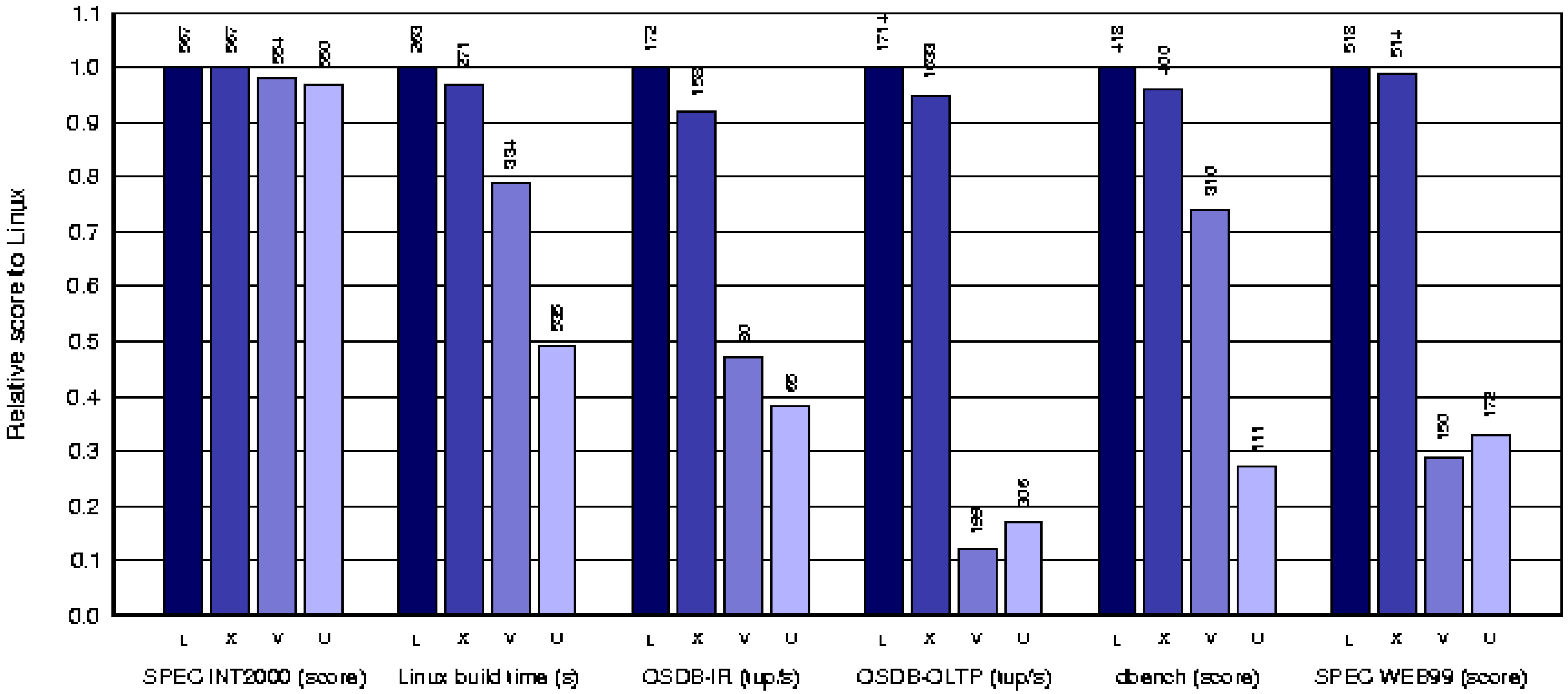


Device drivers in Xen and KVM



Performance of Xen (2003)

- Why does the first set of bars have the least slowdown?



Modern Perspective

- Hardware support for virtualization is dominant
  - KVM is distributed as part of Linux
  - Memory overheads still an issue
  - Device virtualization current frontier
- But Xen lives on!
  - Current Linux kernel supports Dom0 and user domains
  - Performance & security

- 2003: Initial release of Xen
- 2005 was a significant year for Virtualization
  - Intel introduces VT-x, quickly utilized by Xen
  - Narrows performance gap between HVM and PVM
- 2006: Amazon opens up public beta of EC2
- 2007: Live migration for HVM guests
- 2008: PCI pass-through (VT-d) and ACPI S3 support
- 2011: Xen support for Dom0 and DomU is added to the Linux kernel