

Virtualizing CPU Ctd.

Virtualizing Memory

Yiying Zhang

Administravia

- Form your group and email TA group info by the end of this Wed (11:59pm)
- Decide your project topic and send it to the TA asap (ideally this Wed, but no later than next Mon)
- Project proposal due on 4/18!
- First quiz next Wed
- Guest lecture on 4/17

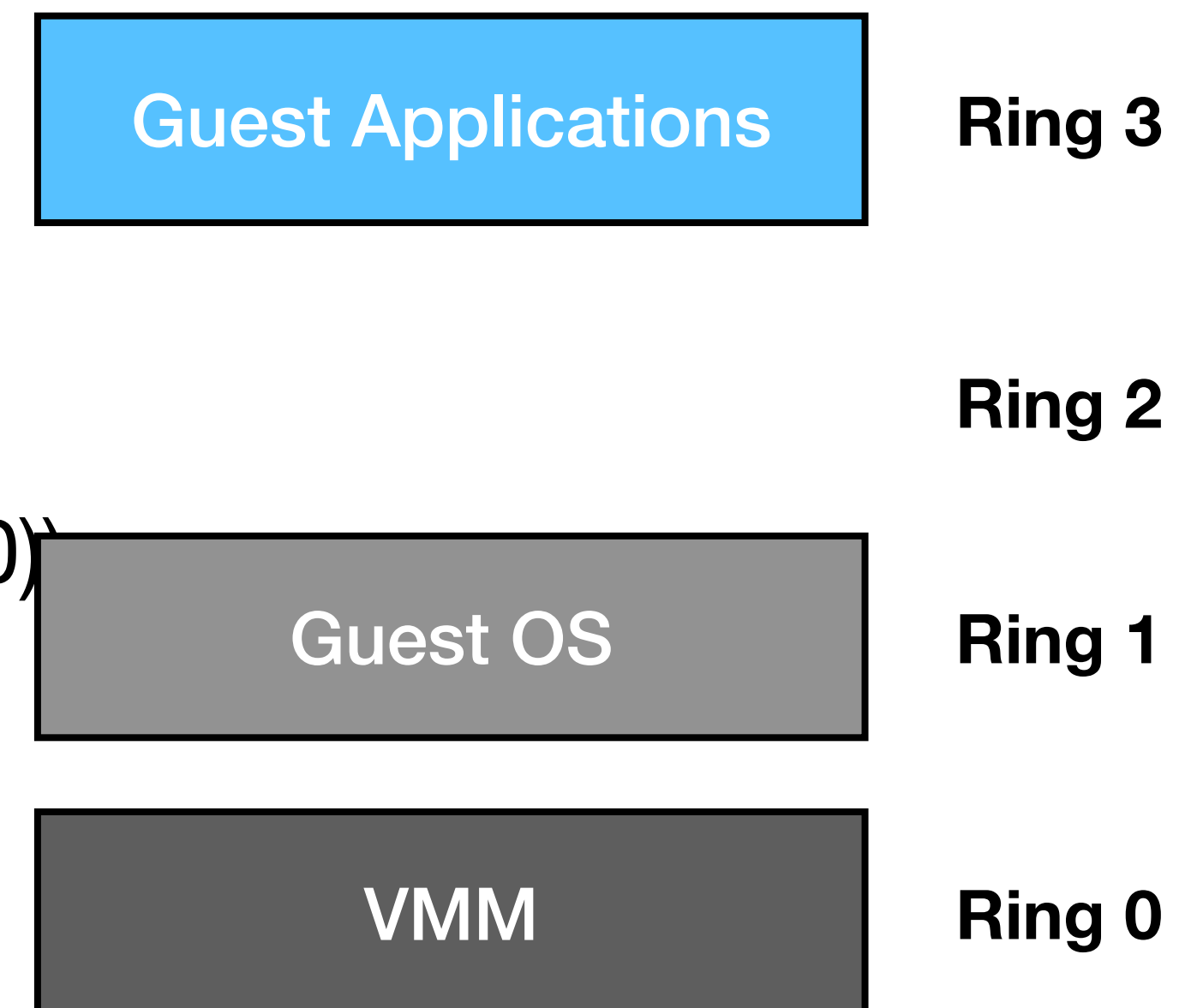
[Recap] Virtualization Approach 1: Complete Machine Emulation (Hosted Interpretation)

- VMM implements the complete hardware architecture in software
- VMM steps through VM's instructions and update emulated hardware as needed

```
while(1){  
    curr_instr = fetch(virtHw.PC);  
    virtHw.PC += 4;  
    switch(curr_instr){  
        case ADD:  
            int sum = virtHw.regs[curr_instr.reg0] +  
                      virtHw.regs[curr_instr.reg1];  
            virtHw.regs[curr_instr.reg0] = sum;  
            break;  
        case SUB:  
            //...etc...
```

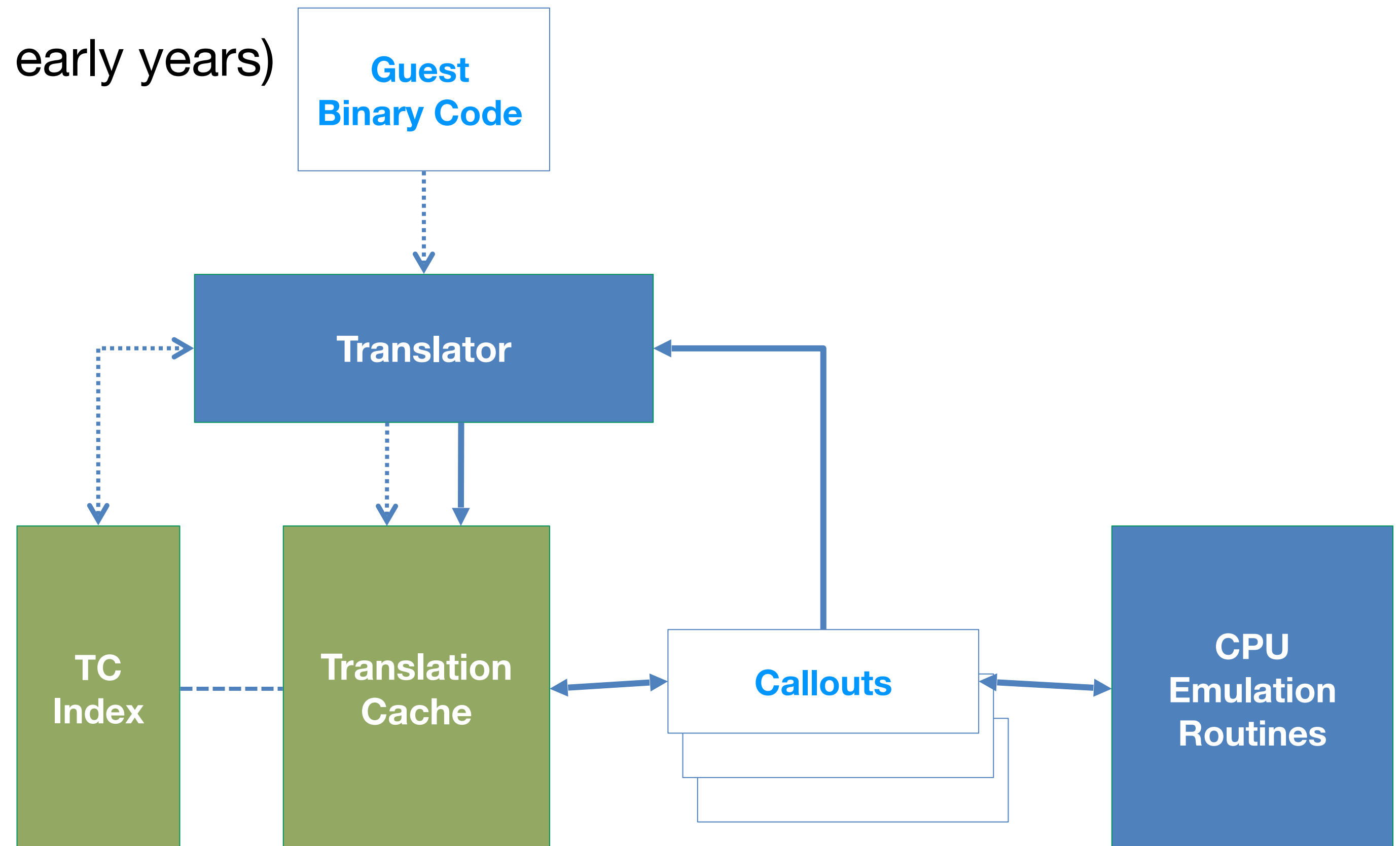
[Recap] Virtualization Approach 2: Direct Execution with Trap-and-Emulate

- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)
- Applications run in ring 3 (can't access memory owned by guest OS (ring1))
- Guest OS runs in ring 1 (can't access memory owned by VMM (ring 0))
- Cannot allow guest OS to run *sensitive instructions* directly!
 - Those that touches hardware configurations



[Recap] Virtualization Approach 3: Direct Execution with Binary Translation

- VMM dynamically rewrites instructions
- So that non-virtualizable instructions can trap to VMM
- VMware's main selling point (at least in early years)



Adaptive Binary Translation

- Binary translation can outperform classical virtualization by avoiding traps
 - *rdtsc* on Pentium 4: trap-and-emulate 2030 cycles, callout-and-emulate 1254 cycles, in-TC emulation 216 cycles
- What about non-privileged instructions accessing protected data like page table?
 - “Innocent until proven guilty”
 - Start in the innocent state and detect instructions that trap frequently
 - Retranslate non-IDENT to avoid the trap
 - Patch the original IDENT translation with a forwarding jump to the new translation

Virtualization Approach 4: Direct Execution with Hardware-Assisted Virtualization

- Adds a new mode so that sensitive operations could all be properly handled
- Other hardware support to make virtualization easier/faster

Hardware-Assisted CPU Virtualization (Intel VT-x)

- Two new modes of execution (orthogonal to protection rings)
 - VMX root mode: same as x86 without VT-x
 - VMX non-root mode: runs VM, sensitive instructions cause transition to root mode, even in Ring 0
- New hardware structure: VMCS (virtual machine control structure)
 - One VMCS for one virtual processor
 - Configured by VMM to determine which sensitive instructions cause VM exit
 - Specifies guest OS state

Comparison of Pre VT-x and Post VT-x

Guest Applications

Ring 3

Guest OS

Ring 1

Hypervisor

Ring 0

Hardware w/o VT-x

Guest Applications

VMX non-root
Ring 3

Guest OS

VMX non-root
Ring 0

Host Applications

VMX root Ring 3

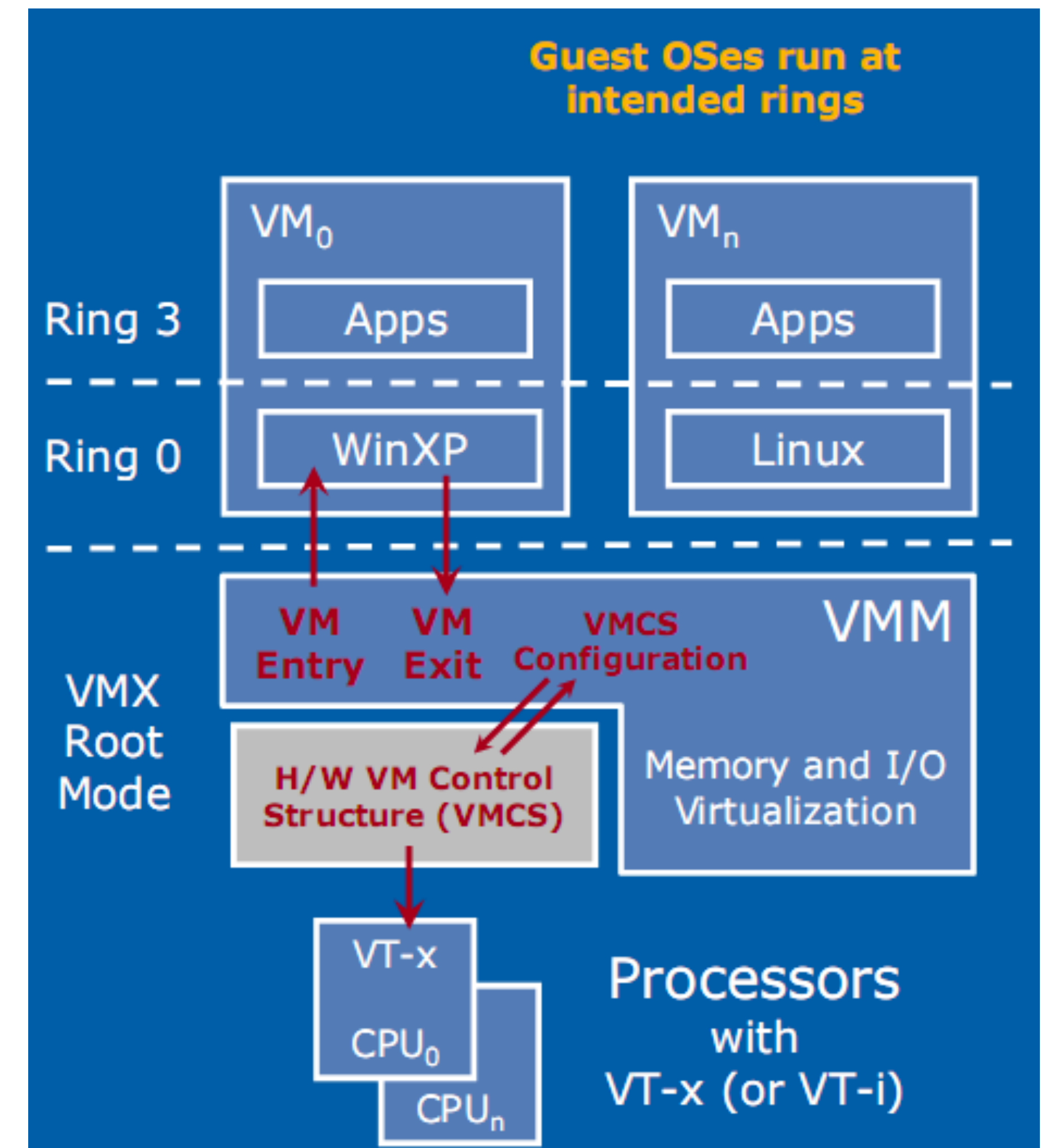
Hypervisor

VMX root Ring 0

Hardware w/ VT-x

VMX Mode Transition with Intel VT-x

- VM exit/entry (to/from root mode)
 - Registers and address space swapped in one atomic operation
 - Guest- and host-states saved and loaded to VMCS during transitions
- VM exit
 - *vmcall* instruction
 - EPT page faults (more later this lecture)
 - Interrupts
 - Sensitive instructions (configured in VMCS)
- VM entry
 - *vmlaunch* instruction: enter with a new VMCS
 - *vmresume* instruction: enter for the last VMCS
- Typical vm exit/enter takes ~200 cycles on modern CPU
- Whenever possible, sensitive instructions can be handled without an VM exit by just changing or accessing states within the VMCS



Example: Guest syscall with Hardware Virtualization

- VMM fills VMCS exception table for guest OS (including a syscall handlers)
 - and sets bit in VMCS to not exit on syscall exception
- VMM executes VM entry
- Guest application invokes a syscall
- does not trap (no VMM involvement), but go to the VMCS exception table, which jumps to the guest OS's syscall handler

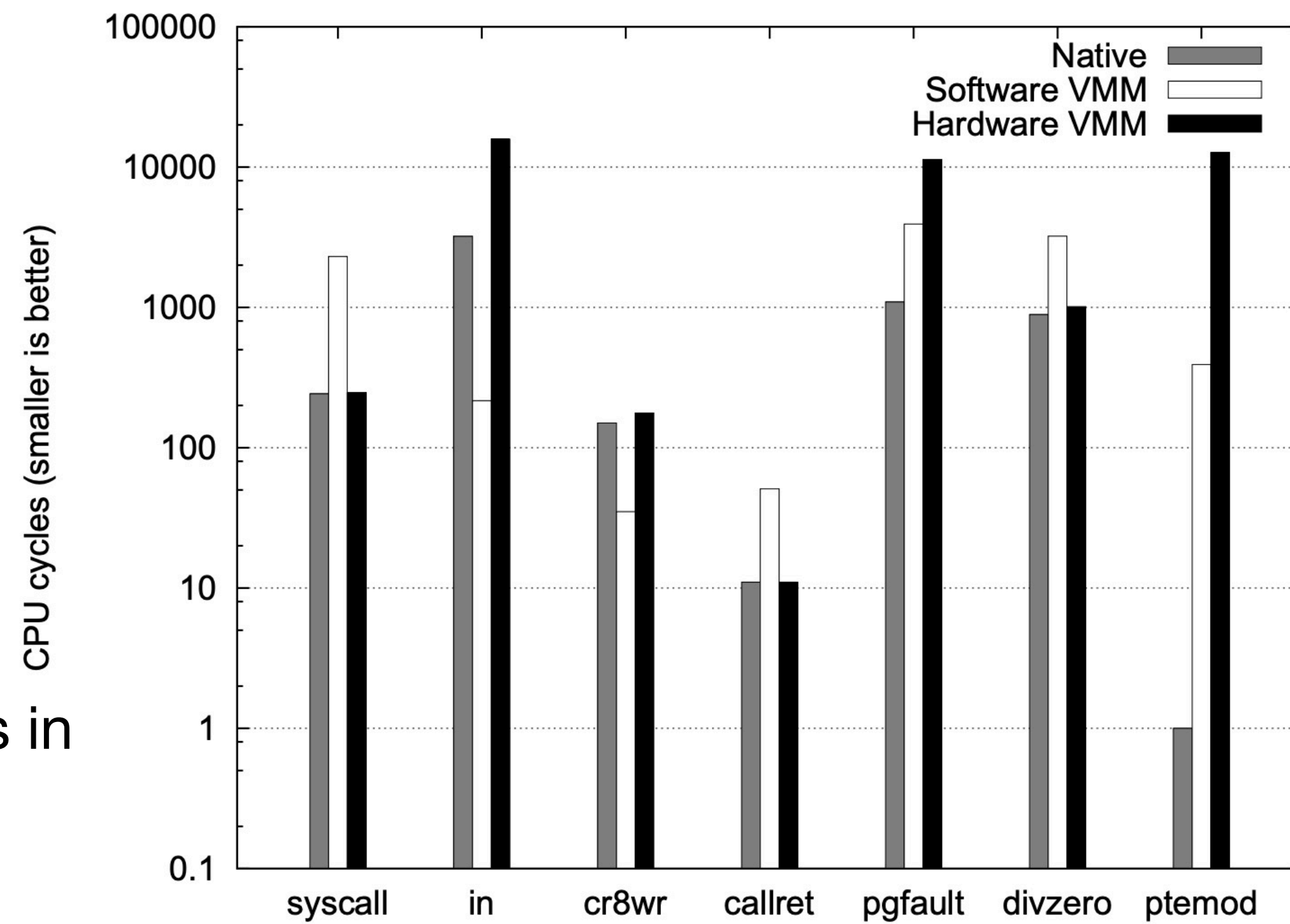
Software Binary Translation vs. Hardware-Assisted Virtualization

- Software binary translation wins in

- Trap elimination
- Emulation speed
- Callout avoidance

- Hardware-assisted virtualization wins in

- Code density
- Precise exceptions
- Syscalls



syscall, call/ret, divzero

=> native = hardware > software

in, cr8wr

=> software > native > hardware

pgfault, ptemod

=> native > software > hardware

Figure 4. Virtualization nanobenchmarks.

Virtualization Approach 5: Direct Execution with Paravirtualization

- Full virtualization (no guest OS modification)
 - Tricky and has performance overhead
- Para-virtualization: modified guest OS
 - Change (rewrite) guest OS to remove sensitive but unprivileged instructions and to use other tricks to make virtualization faster
 - Guest OS works with hypervisor (i.e., knows that it is a VM) and has some exposure to hardware
 - e.g., guest OS informs hypervisor of page table changes
 - e.g., guest OS directly calls hypervisor on system calls (*hypercalls*)
 - Guest applications are still unmodified
 - Pros and Cons?

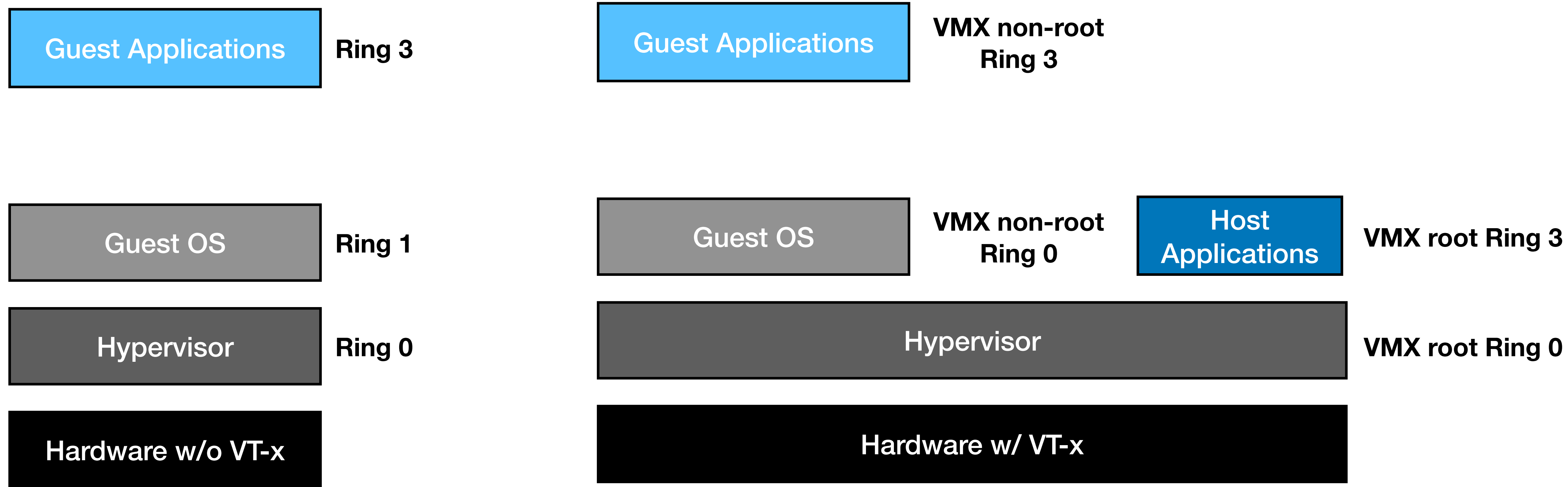
Other Virtualization Approaches

- Container: Essentially just a group of processes with some additional features (isolated namespace, isolated resources, etc.) (e.g., Docker)
- Unikernel: LibraryOS designed for a single application, running on hypervisor (as a VM) or host OS (as a process)
- Sandboxing: Limit what the applications (and libOS) can do (e.g., gVisor)
- Language-based: Running applications written in a high-level language on language runtimes (e.g., JVM)

Virtualization Approaches Summary

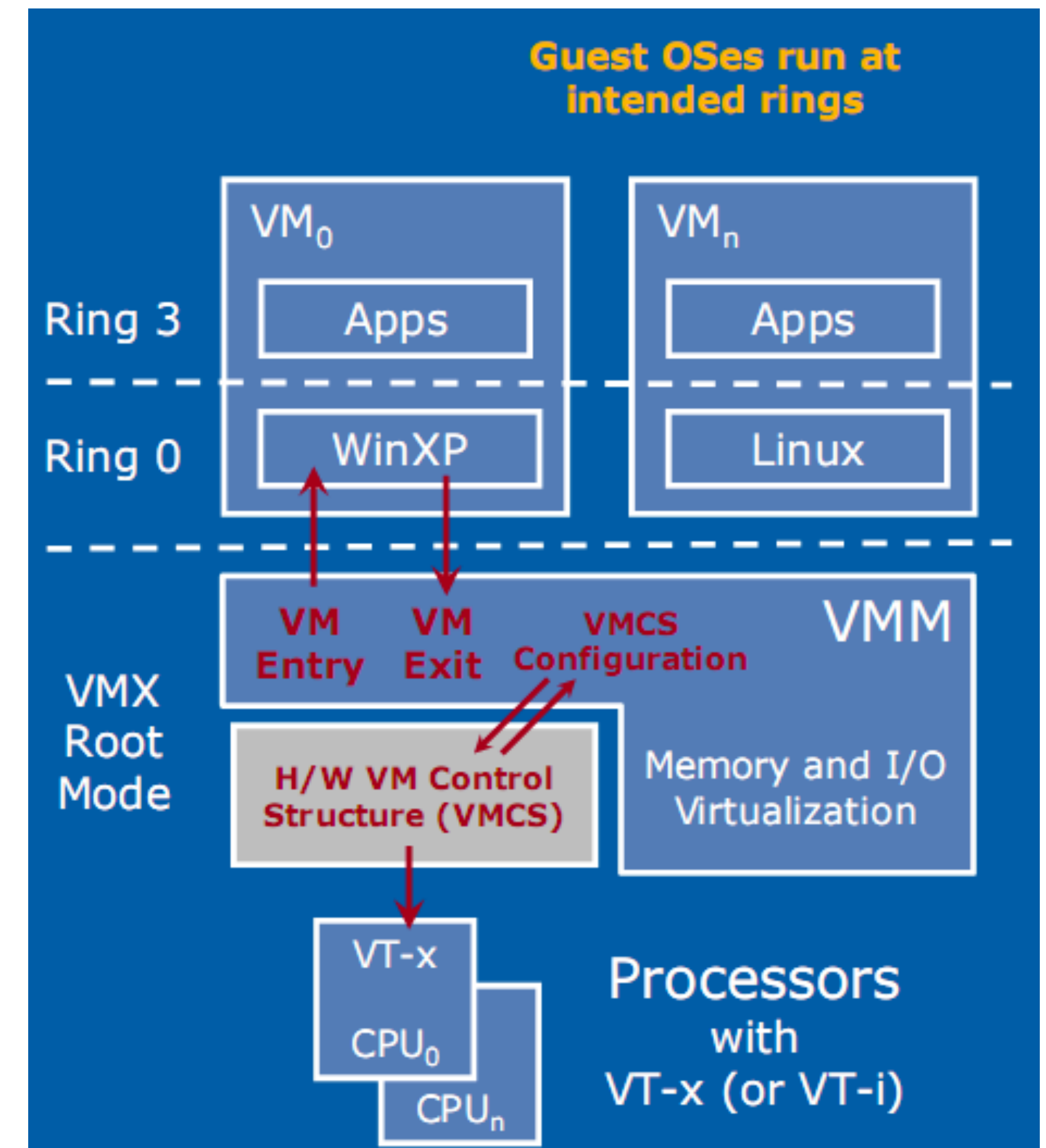
- Hosted interpretation
 - Interpret each instruction, super slow (e.g., Virtual PC on Mac)
- Direct execution with trap-and-emulate
 - Requires a virtualizable processor and only works for the same architecture
- Direct execution with binary translation
 - Works with non-virtualizable processor, but implementing VMM is tricky
- Direct execution with hardware-assisted virtualization
 - Needs new generation of hardware (which is the norm now), mode switching is still not optimized
- Direct execution with paravirtualization
 - Good performance and works with non-virtualizable processors, but require guest OS changes
- OS-level virtualization, library-level, language (app)-level, unikernels, etc.
 - More lightweight and faster to start, but less secure

Review and Finishing Off CPU



VMX Mode Transition with Intel VT-x

- VM exit/entry (to/from root mode)
 - Registers and address space swapped in one atomic operation
 - Guest- and host-states saved and loaded to VMCS during transitions
- Whenever possible, sensitive instructions only affect states within the VMCS instead of always trapping (VM exit)
- VM exit
 - *vmcall* instruction
 - EPT page faults (more next lecture)
 - Interrupts
 - Some sensitive instructions (configured in VMCS)
- VM entry
 - *vmlaunch* instruction: enter with a new VMCS
 - *vmresume* instruction: enter for the last VMCS
- Typical vm exit/enter takes ~200 cycles on modern CPU



Example: Guest syscall with Hardware Virtualization

- VMM fills VMCS interrupt table for guest OS (including a syscall handlers)
 - and sets bit in VMCS to not exit on syscalls
- VMM executes VM entry
- Guest application invokes a syscall
- does not trap (no VMM involvement), but go to the VMCS interrupt table, which jumps to the guest OS's syscall handler

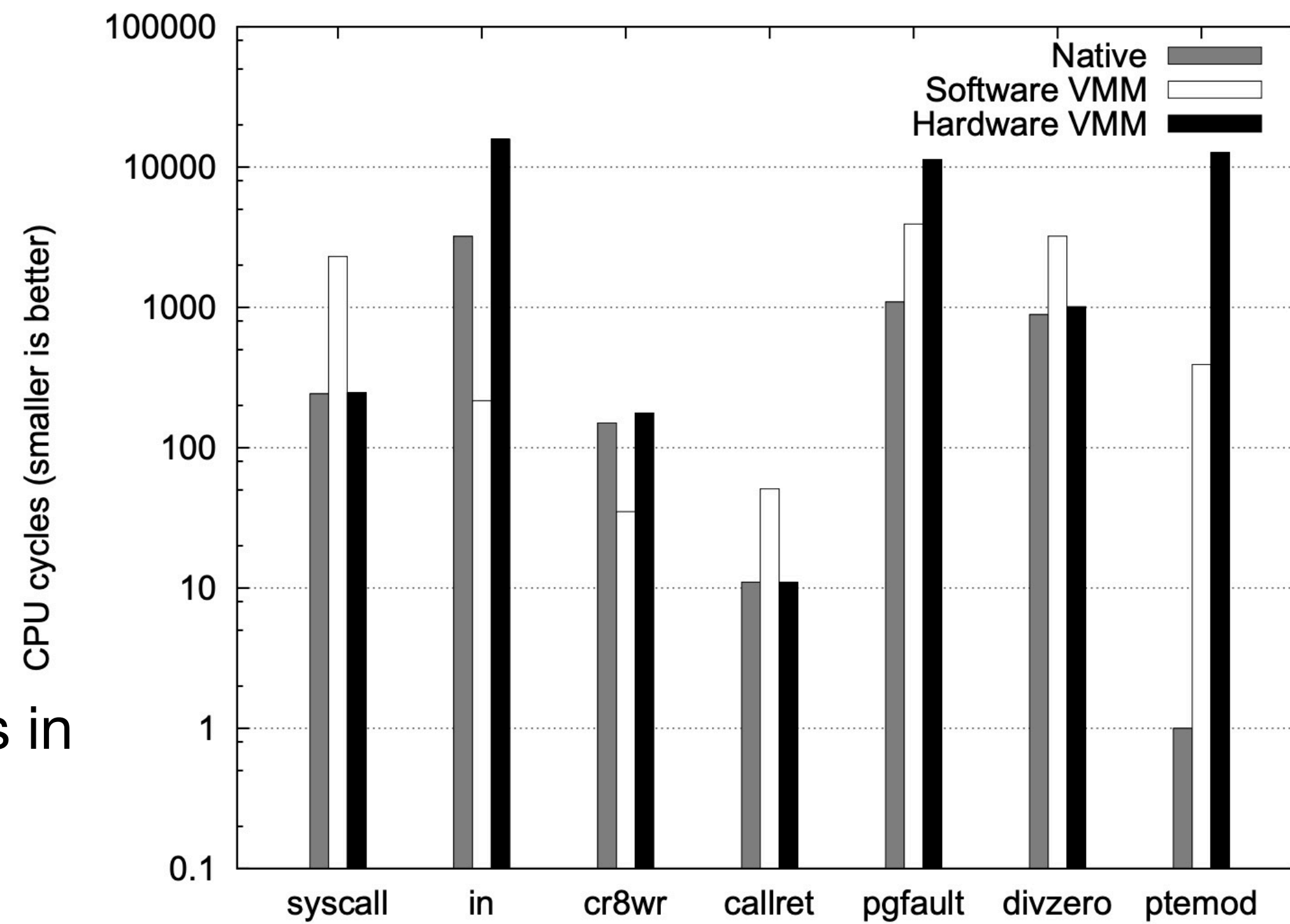
Software Binary Translation vs. Hardware-Assisted Virtualization

- Software binary translation wins in

- Trap elimination
- Emulation speed
- Callout avoidance

- Hardware-assisted virtualization wins in

- Code density
- Precise exceptions
- Syscalls



syscall, call/ret, divzero

=> native = hardware > software

in, cr8wr

=> software > native > hardware

pgfault, ptemod

=> native > software > hardware

Figure 4. Virtualization nanobenchmarks.

Virtualization Approach 5: Direct Execution with Paravirtualization

- Full virtualization (no guest OS modification)
 - Tricky and has performance overhead
- Para-virtualization: modified guest OS
 - Change (rewrite) guest OS to remove sensitive but unprivileged instructions and to use other tricks to make virtualization faster
 - Guest OS works with hypervisor (i.e., knows that it is a VM) and has some exposure to hardware
 - e.g., guest OS informs hypervisor of page table changes
 - e.g., guest OS directly calls hypervisor on system calls (*hypercalls*)
 - Guest applications are still unmodified
 - Pros and Cons?

Other Virtualization Approaches

- Container: Essentially just a group of processes with some additional features (isolated namespace, isolated resources, etc.) (e.g., Docker)
- Unikernel: LibraryOS designed for a single application, running on hypervisor (as a VM) or host OS (as a process)
- Sandboxing: Limit what the applications (and libOS) can do (e.g., gVisor)
- Language-based: Running applications written in a high-level language on language runtimes (e.g., JVM)

Virtualization Approaches Summary

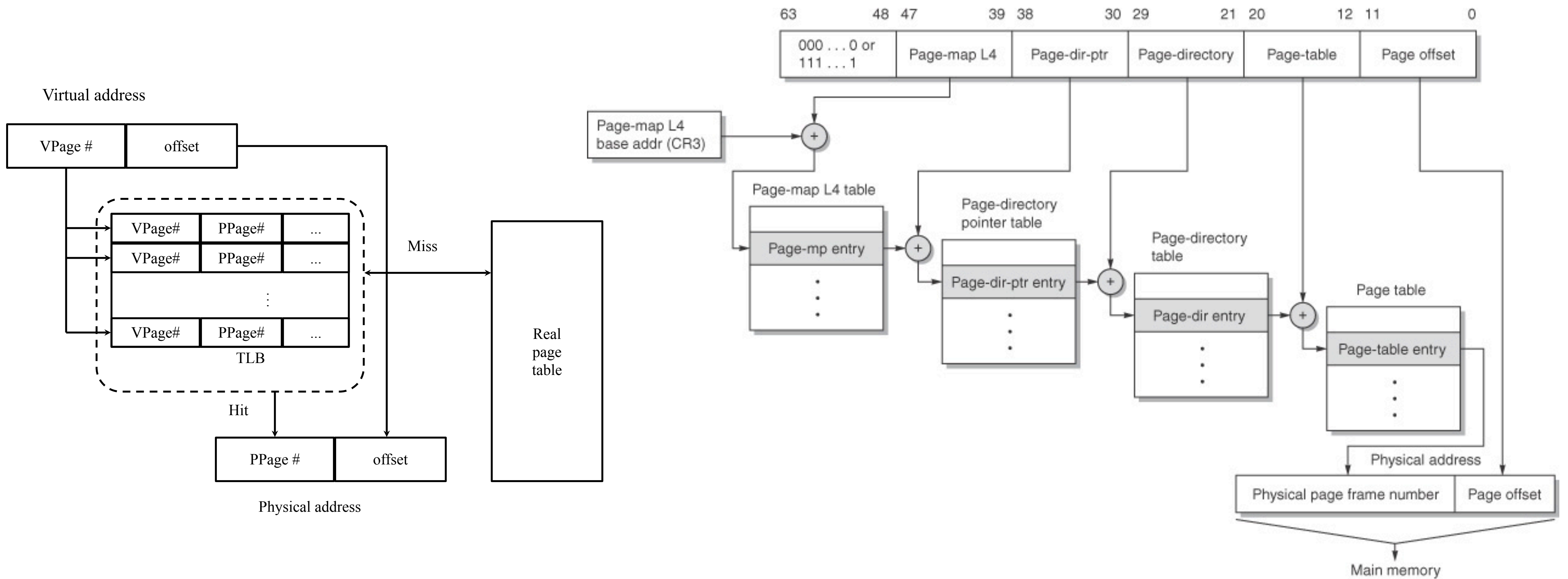
- Hosted interpretation
 - Interpret each instruction, super slow (e.g., Virtual PC on Mac)
- Direct execution with trap-and-emulate
 - Requires a virtualizable processor and only works for the same architecture
- Direct execution with binary translation
 - Works with non-virtualizable processor, but implementing VMM is tricky
- Direct execution with hardware-assisted virtualization
 - Needs new generation of hardware (which is the norm now), mode switching is still not optimized
- Direct execution with paravirtualization
 - Good performance and works with non-virtualizable processors, but require guest OS changes
- OS-level virtualization, library-level, language (app)-level, unikernels, etc.
 - More lightweight and faster to start, but less secure

Outline

- Software-based memory virtualization
- Hardware-assisted memory virtualization
- Memory management
 - Reclaiming
 - Sharing

Acknowledgment: some slides from Carl Waldspurger's OSDI'02 presentation

Review: Regular Virtual Memory System



Review: Software-controlled TLB

- On a TLB hit, MMU checks the valid bit
 - If valid, perform address translation
 - If invalid (e.g. page not in memory), MMU generates a page fault
 - OS performs page fault handling
 - Restart the faulting instruction
- On a TLB miss, HW raises exception, **traps to the OS**
 - **OS** parses page table and loads PTE into TLB
 - Needs to replace if TLB is full
 - Same as in a hit...

Review: Hardware-controlled TLB

- On a TLB hit, MMU checks the valid bit
 - If valid, perform address translation
 - If invalid (e.g. page not in memory), MMU generates a page fault
 - OS performs fault handling
 - Restart the faulting instruction
- On a TLB miss
 - MMU parses page table and loads PTE into TLB
 - Needs to replace if TLB is full
 - Same as hit ...

Virtualizing Memory

- Extra level of memory addressing

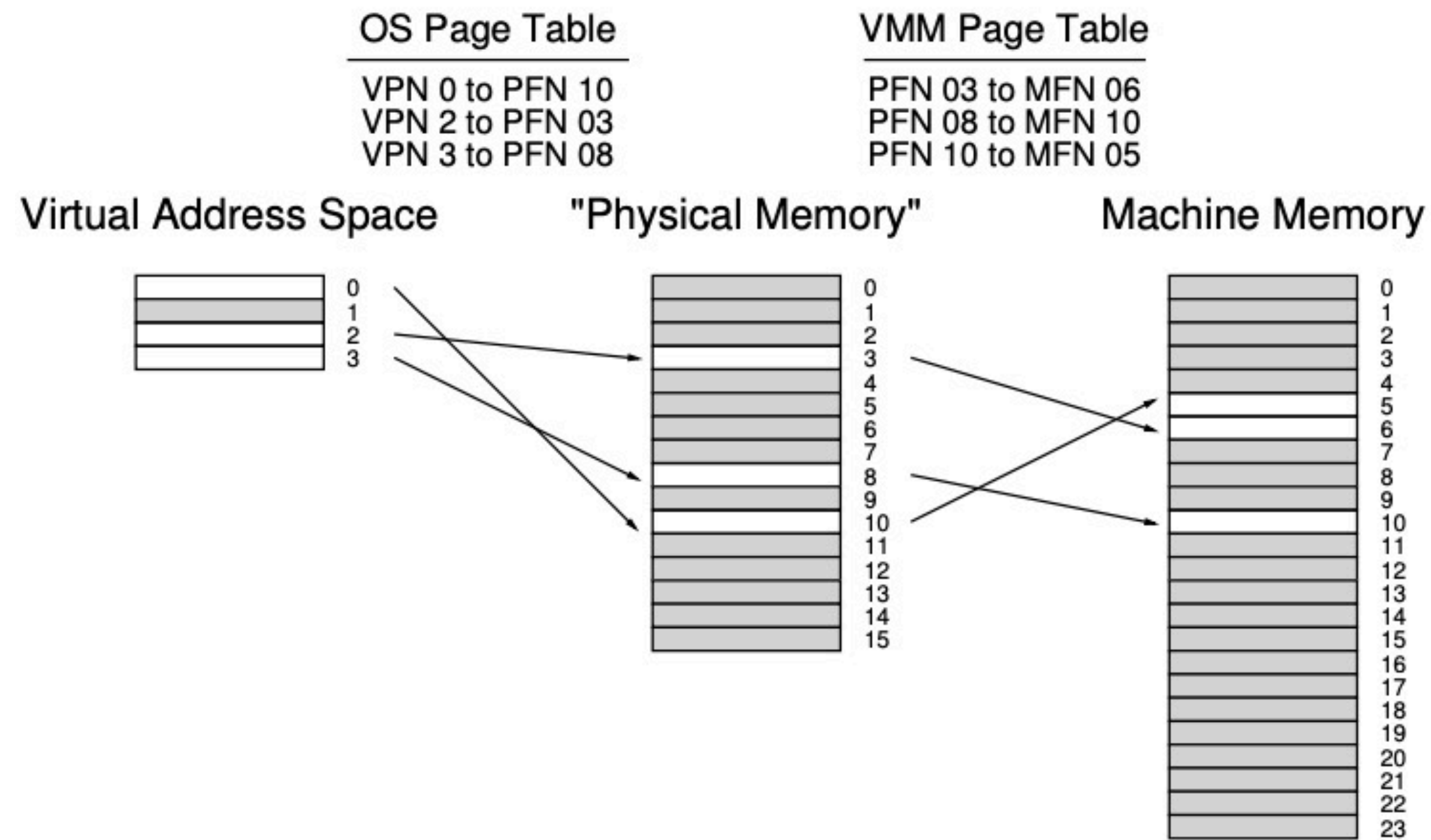


Figure B.4: VMM Memory Virtualization

Virtualizing Memory

- TLB miss flow with software-managed TLB

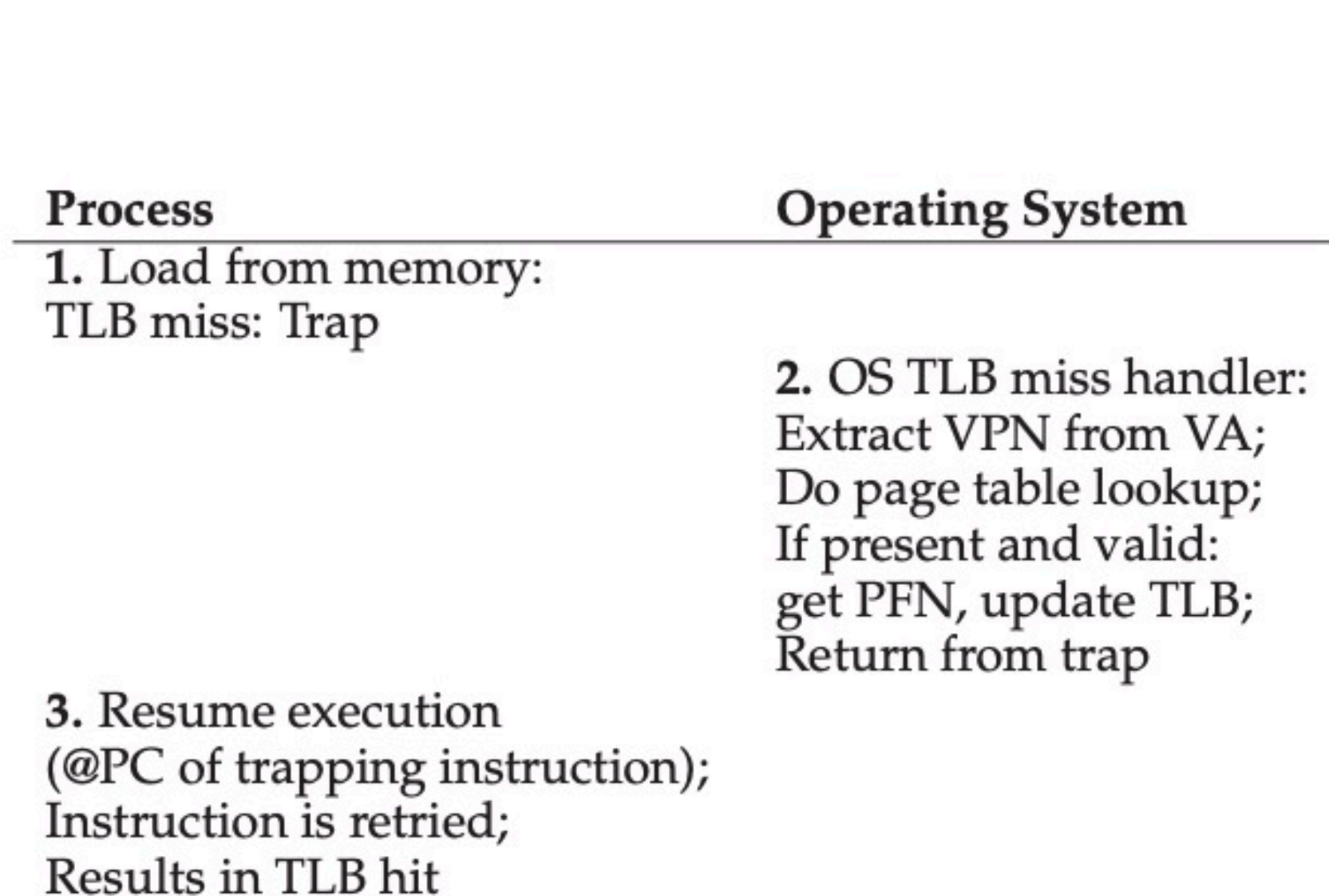


Figure B.5: TLB Miss Flow without Virtualization

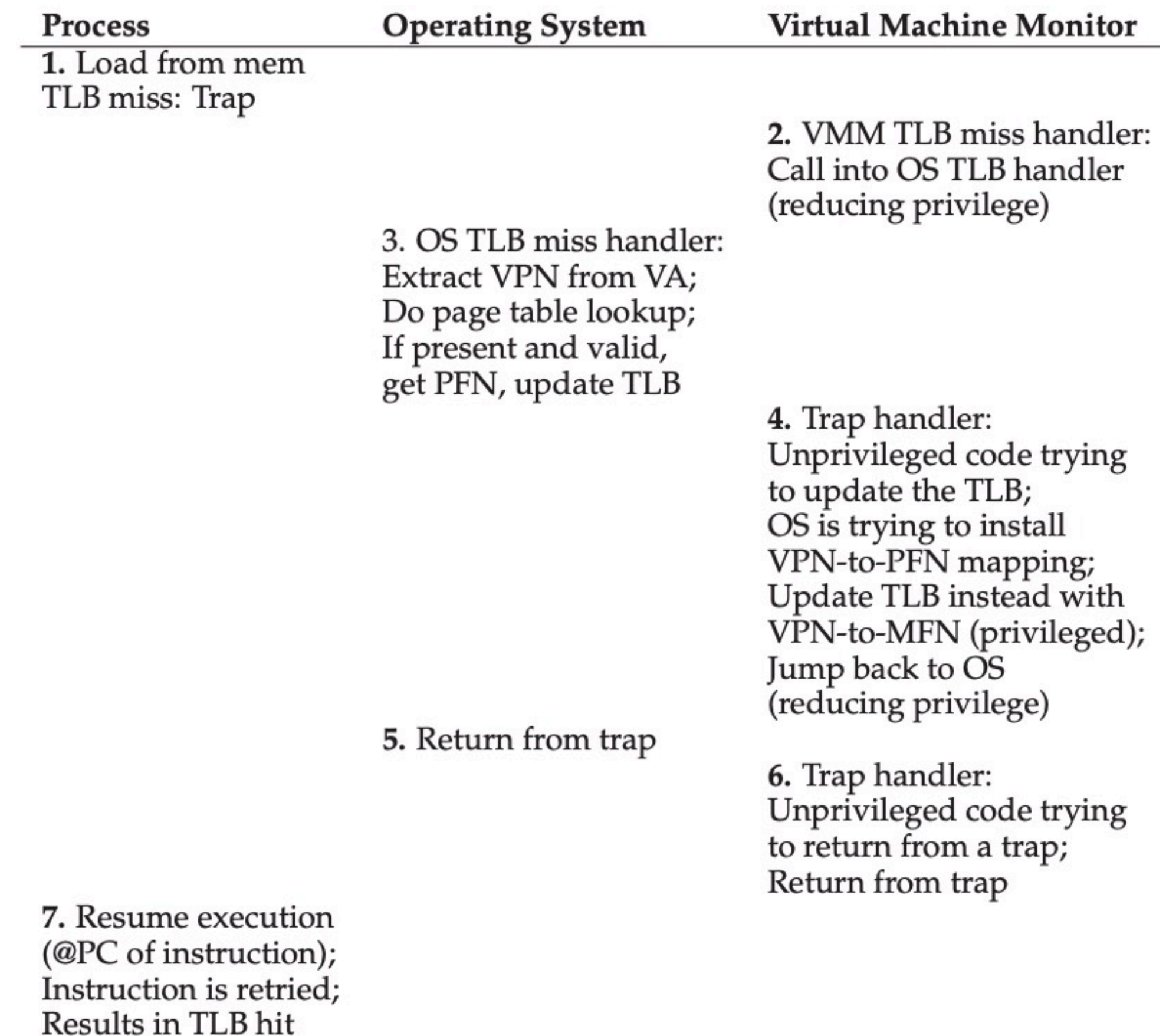
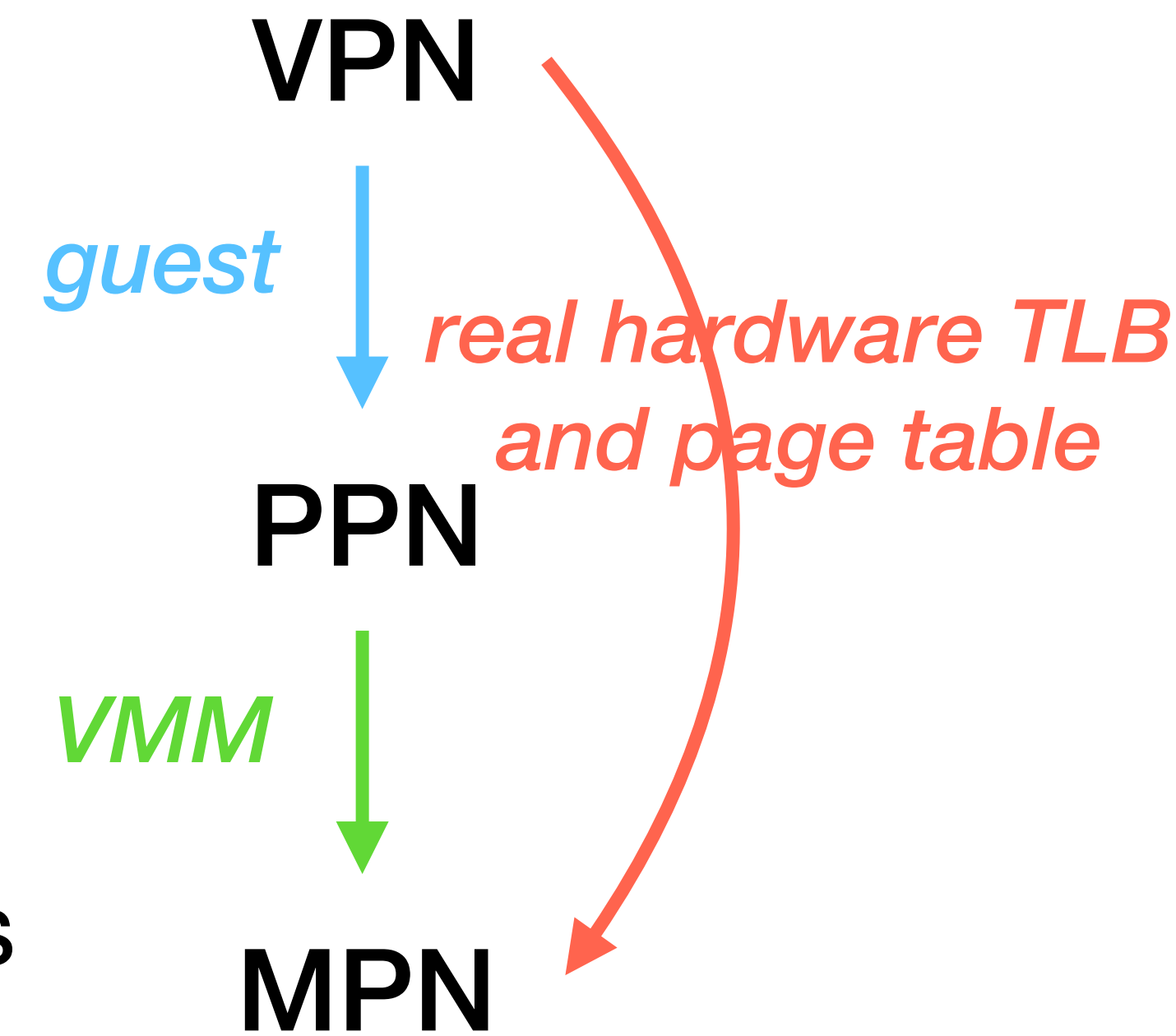


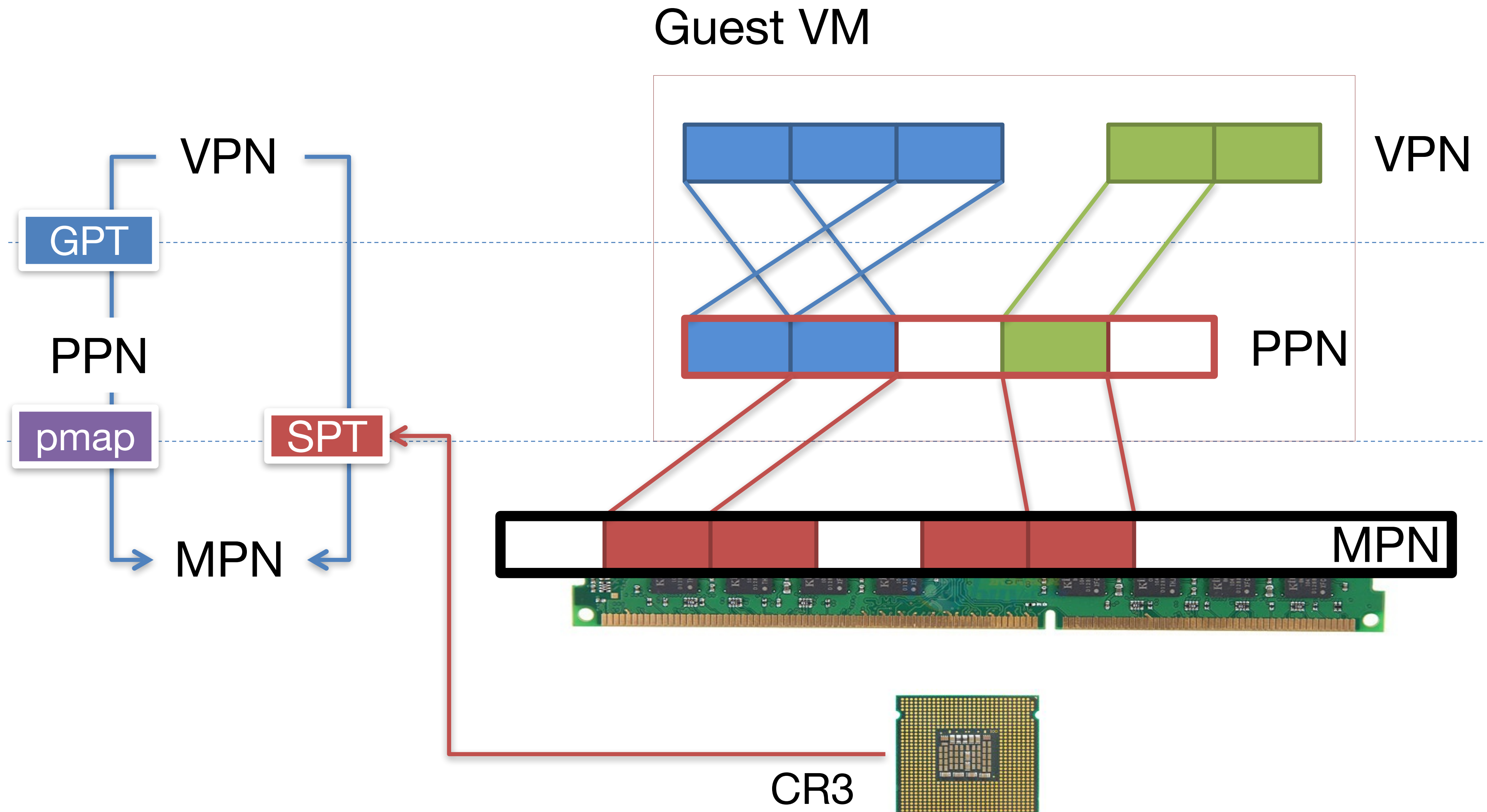
Figure B.6: TLB Miss Flow with Virtualization

Difficulty in Virtualizing Hardware-Managed TLB

- Hardware-managed TLB
 - Hardware does page table walk on each TLB miss
 - and fills TLB with the found PTE
- Hypervisor doesn't have chance to intercept on TLB misses
- Solution-1: **shadow paging**
- Solution-2: **direct paging (para-virtualization)** *(later this quarter if have time)*
- Solution-3: **new hardware**



Shadow Paging



Set Up Shadow Page Table

1. VMM intercepts guest OS setting the virtual CR3 (a sensitive operation)
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest virtual address is translated into host physical address (machine address)
4. Finally, VMM sets the real CR3 to point to the shadow page table

Set Up Shadow Page Table

```
set_cr3 (guest_page_table):  
    for VPN in 0 to 220  
        if guest_page_table[VPN] & PTE_P /* PTE_P: valid bit */  
            PPN = guest_page_table[VPN]  
            MPN = pmap[PPN]  
            shadow_page_table[VPN] = MPN | PTE_P  
        else  
            shadow_page_table = 0  
    CR3 = PHYSICAL_ADDR(shadow_page_table)
```

Question

- Assume that:
 - There are 10 VMs running on a machine
 - Each VM contains 10 applications
- **Q: How many shadow page tables in total?**
 - Shadow page tables are per application
 - Guest page tables are per application
 - pmaps are per VM

What if Guest OS Modifies Its Page Table?

- Should not allow it to happen directly
 - Since CR3 is now pointing to the shadow page table
 - Need to synchronize the shadow page table with guest page table
- VMM needs to intercept when guest OS modifies page table, and updates the shadow page table accordingly
 1. Mark the guest table pages as read-only (by setting the corresponding PTEs' permission bits in the shadow page table)
 2. If guest OS tries to modify its page tables, it triggers page fault
 3. VMM handles the page fault by updating shadow page table

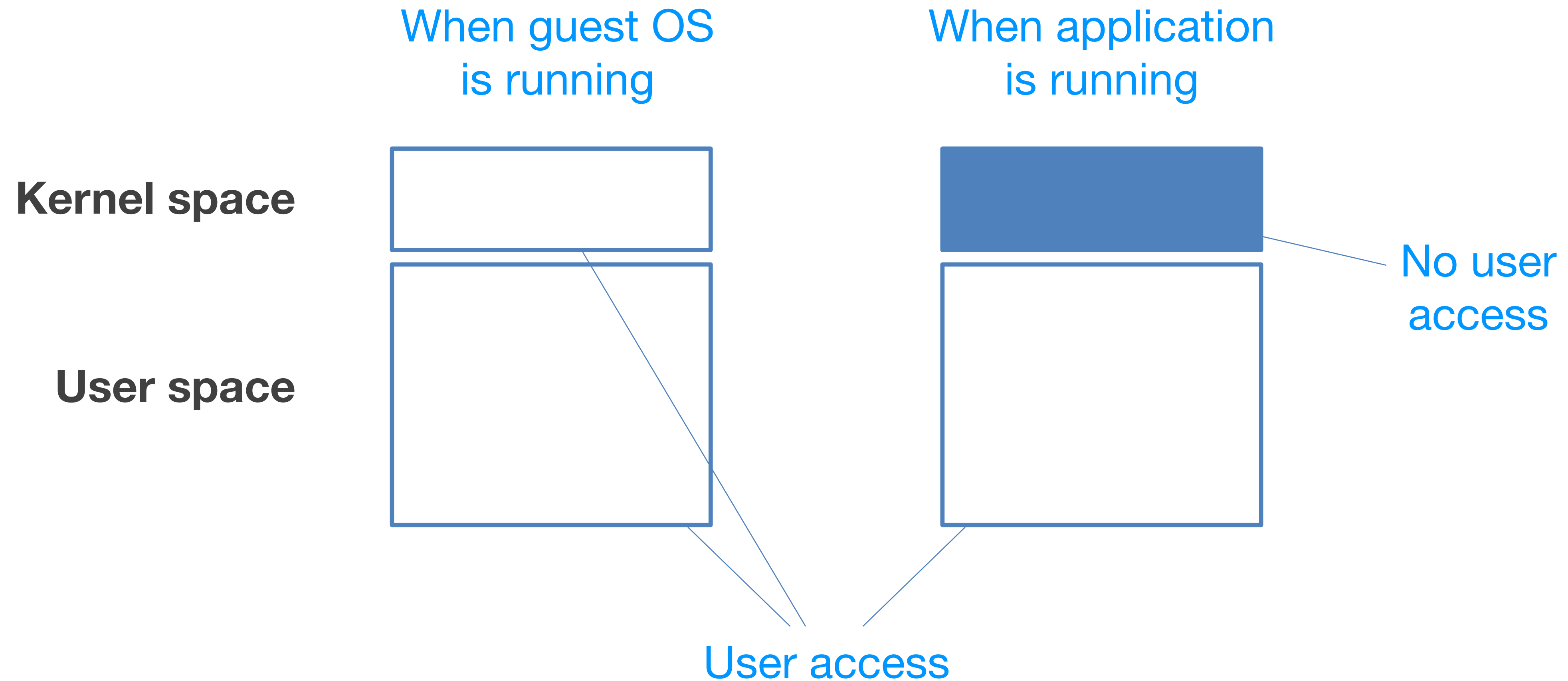
Dealing with Page Faults

- When page fault occurs, traps to VMM
- If present bit is 0 in the guest page table entry, guest OS needs to handle the fault (VMM forwards the fault to guest OS)
 - Guest OS load page from virtual disk to guest physical memory and sets present bit to 1
 - Guest OS returns from page fault, which traps into VMM again
 - VMM sees that present is 1 in guest PTE and creates entry in shadow page table
 - VMM returns from the original page fault
- If present is 1: guest OS thinks page is present (but VMM may have swapped it out), VMM handles transparently
 - VMM locates the corresponding physical page, loads it in memory if needed
 - VMM creates entry in shadow page table
 - VMM returns from the original page fault

What if a Guest App Access its Kernel Memory?

- How do we selectively allow / deny access to kernel-only pages?
- One solution: split a shadow page table into two tables
 - Two shadow page tables, one for user, one for kernel
 - When guest OS switches to guest applications, VMM will switch the shadow page table as well, vice versa

Two Memory Views of Guest VM



The Same Question

- Assume that:
 - There are 10 VMs running on a machine
 - Each VM contains 10 applications
- **Q:** Now, how many shadow page tables in total?

Pros and Cons of Shadow Paging

- Pros
 - When shadow PT is established, memory accesses are very fast
- Cons
 - Maintaining consistency between guest PTs and shadow PTs involve VMM traps, can be costly
 - TLB flush on every “world switch”
 - Memory space overhead to maintain *pmap*

Sol-2: Direct Paging (Para-virtualization)

- Modify the guest OS
 - No PPN is needed, just VPN and MPN
 - Guest OS directly manages its MPN space
 - Use hypercall to let the VMM update the page table
 - The hardware CR3 will point to guest page table
- VMM will check all the page table operations
 - The guest page tables are read-only to the guest

Sol-2: Direct Paging (Para-virtualization)

- Positive
 - Easy to implement and more clear architecture
 - Better performance: guest can batch to reduce trap
- Negatives
 - Not transparent to the guest OS
 - The guest now knows much info, e.g., MPN
 - May use such info to trigger *rowhammer* attacks

Outline

- Software-based memory virtualization
- Hardware-assisted memory virtualization
- Memory management
 - Reclaiming
 - Sharing

Hardware-Assisted Memory Virtualization

- Hardware support for memory virtualization
 - Intel EPT (Extended Page Table) and AMD NPT (Nested Page Table)
 - EPT: a per VM table translating PPN $\rightarrow$ MPN, referenced by EPT base pointer
 - EPT controlled by the hypervisor, guest page table (GPT) controlled by guest OS (both exposed to hardware)
 - Hardware directly walks GPT + EPT (for each PPN access during GPT walk, needs to walk the EPT to determine MPN)
 - No VM exits due to page faults, INVLPG, or CR3 accesses

Guest VM

VMM

VPN

PT

PPN

EPT

MPN

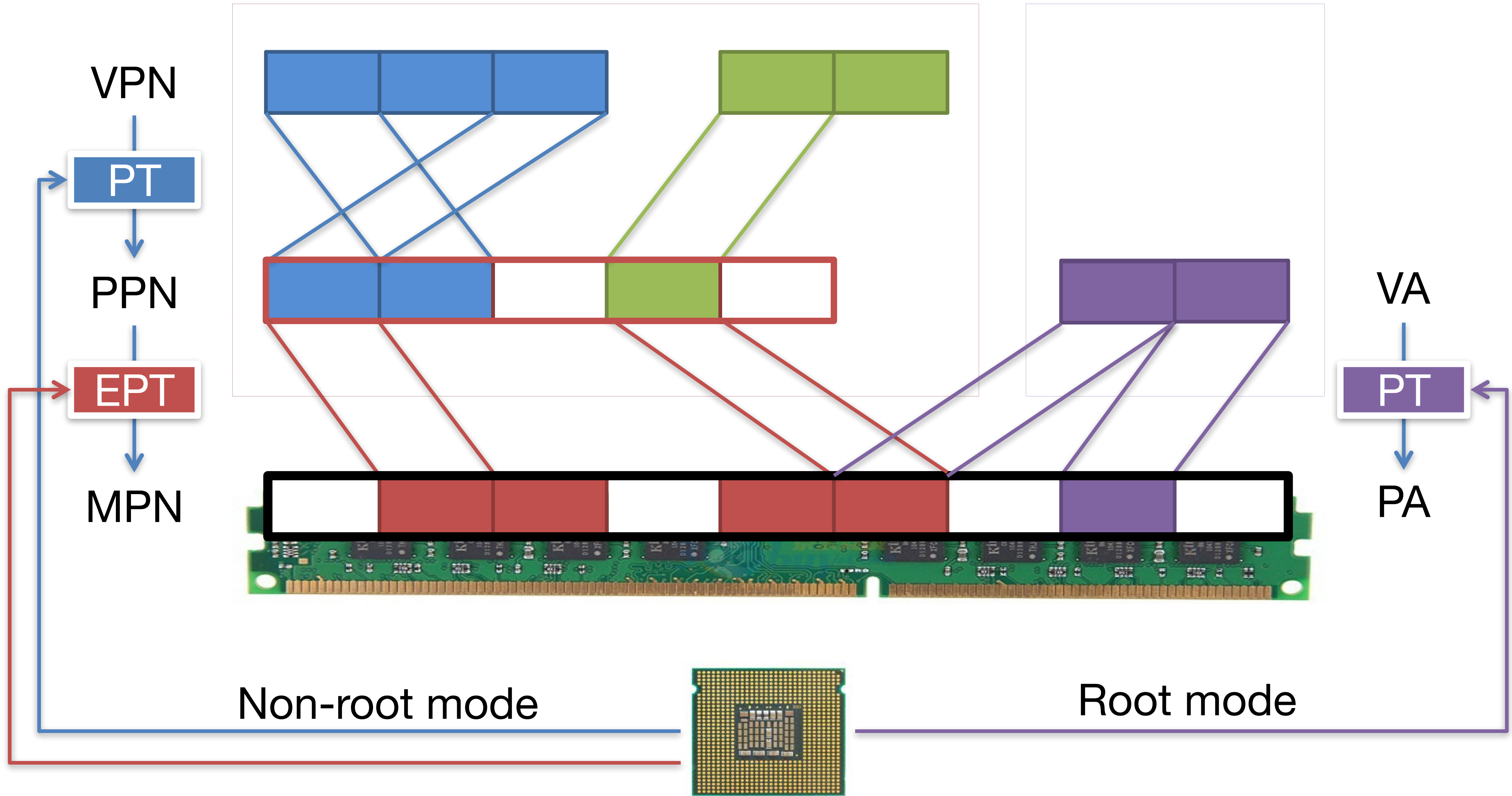
VA

PT

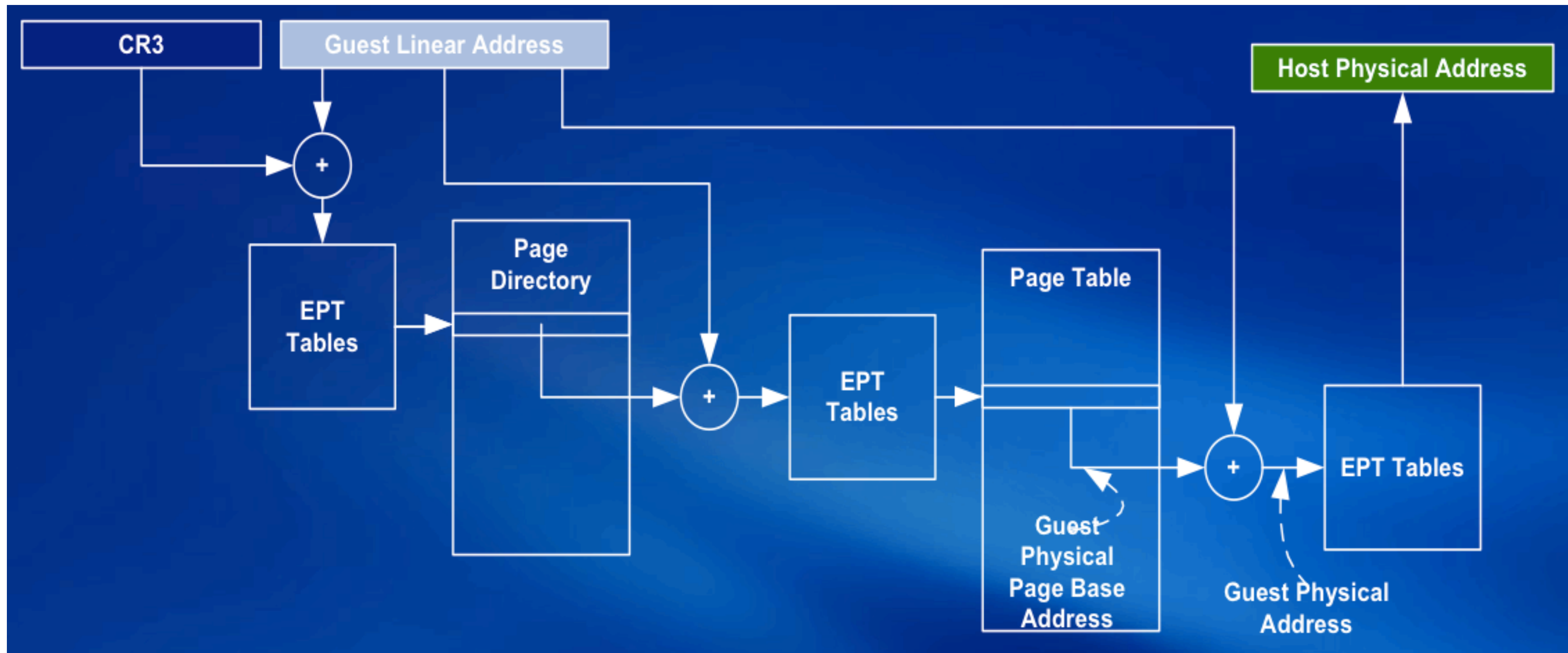
PA

Non-root mode

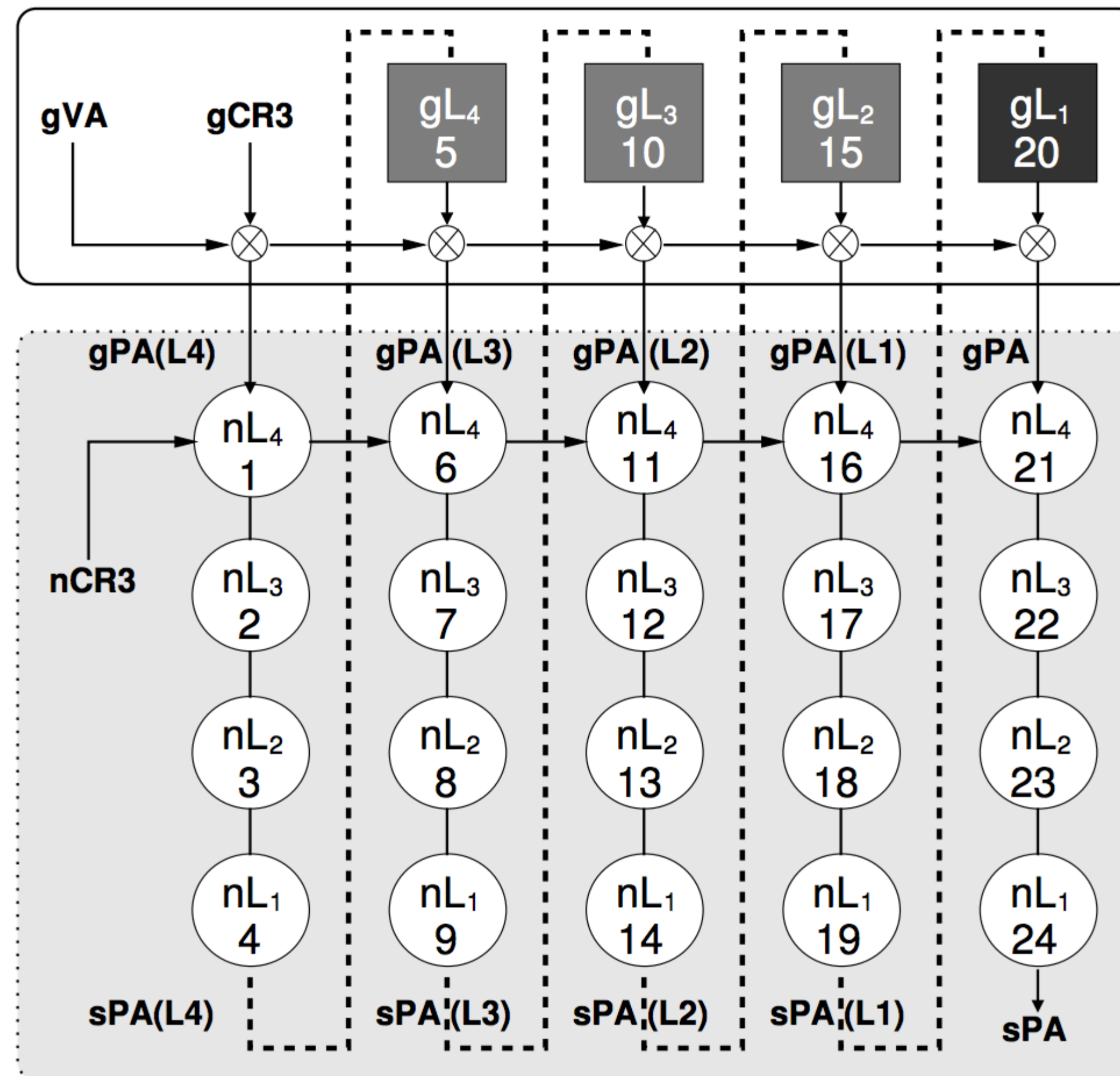
Root mode



EPT Translation: Details



EPT Increases Memory Access



sPA: machine address
nCR3: root of EPT table
nL_k: EPT table level

One memory access from the guest VM may lead up to **20 memory accesses!**

Pros and Cons of EPT

- Pros
 - Simplified VMM design (all handled by hardware)
 - Guest PT changes do not trap, minimize VM exits
- Cons
 - TLB miss is costly: can involve many memory accesses to finish the walk!

Outline

- Software-based memory virtualization
- Hardware-assisted memory virtualization
- Memory management
 - Reclaiming
 - Sharing

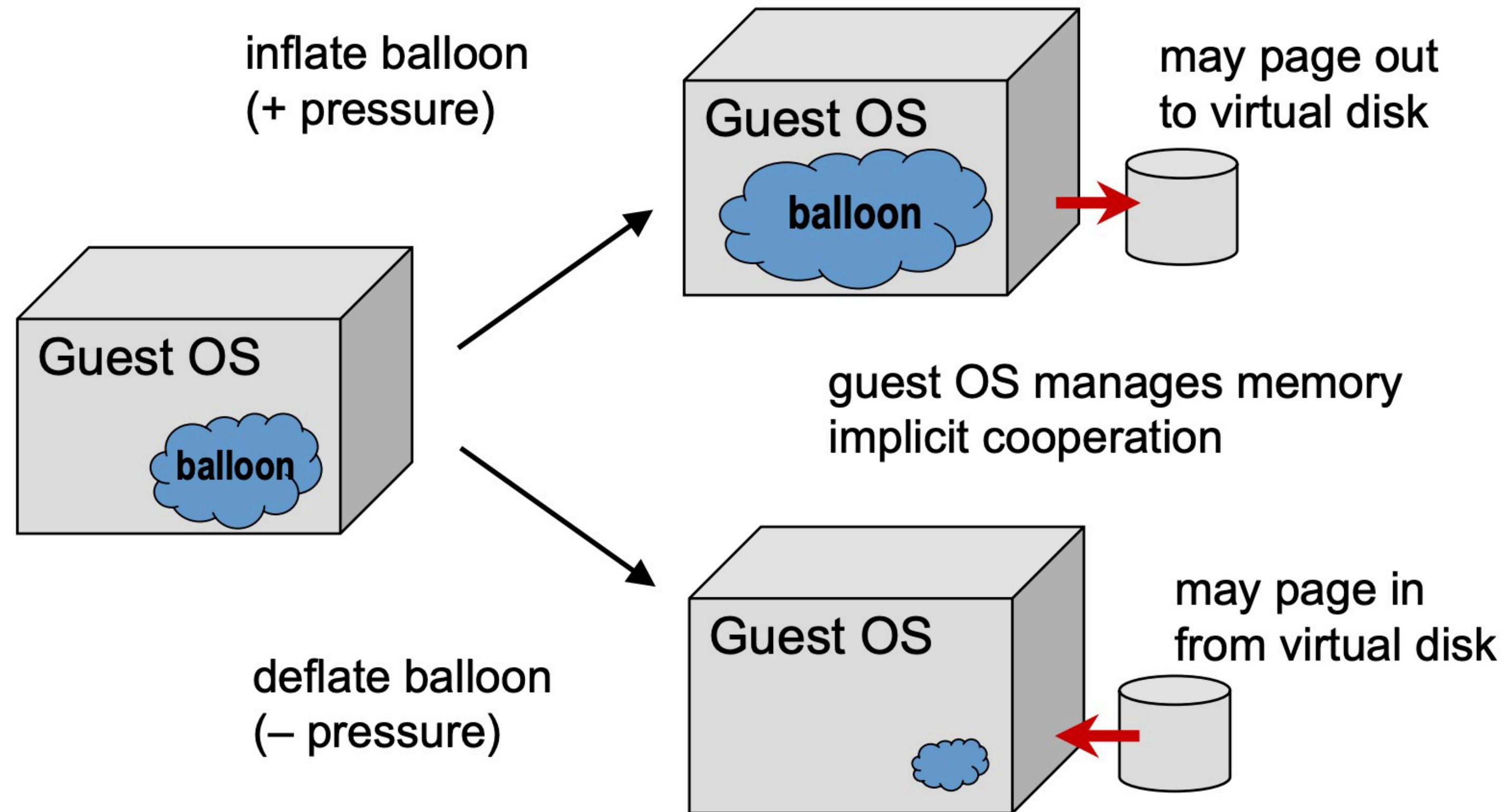
Reclaiming Memory

- ESX (and other hypervisors) allow overcommitment of memory
 - Total memory size of all VMs can exceed actual machine memory size
 - ESX must have some way to reclaim memory from VMs (and swap to disk)

Reclaiming Memory

- Traditional: add transparent swap layer
 - Requires “meta-level” decisions: which page from which VM to swap
 - Best data to guide decisions known only by guest OS
 - Guest and meta-level policies may clash, resulting in *double paging*
- Alternative: implicit cooperation
 - Coax guest OS into doing its own page replacement
 - Avoid meta-level policy decisions

Ballooning



Ballooning Details

- Guest drivers
 - Inflate: guest decides which pages to page out, PPNs communicated to hypervisor via balloon driver
 - Use standard Windows/Linux/BSD kernel APIs
- Performance benchmark
 - Linux VM, memory-intensive dbench workload
 - Compare 256 MB with balloon sizes 32 - 128 MB vs. static VMs
 - Overhead 1.4% - 4.4%

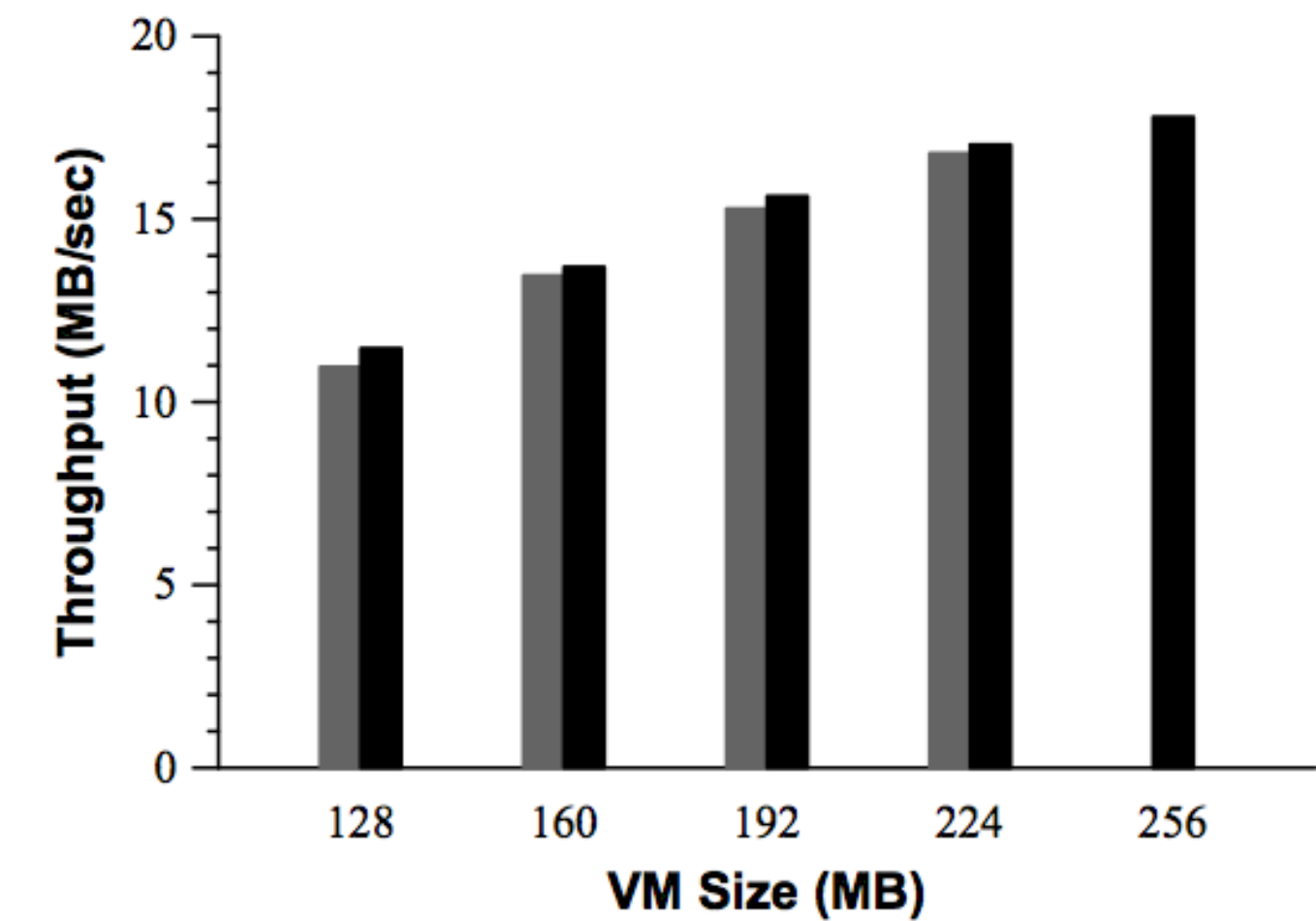
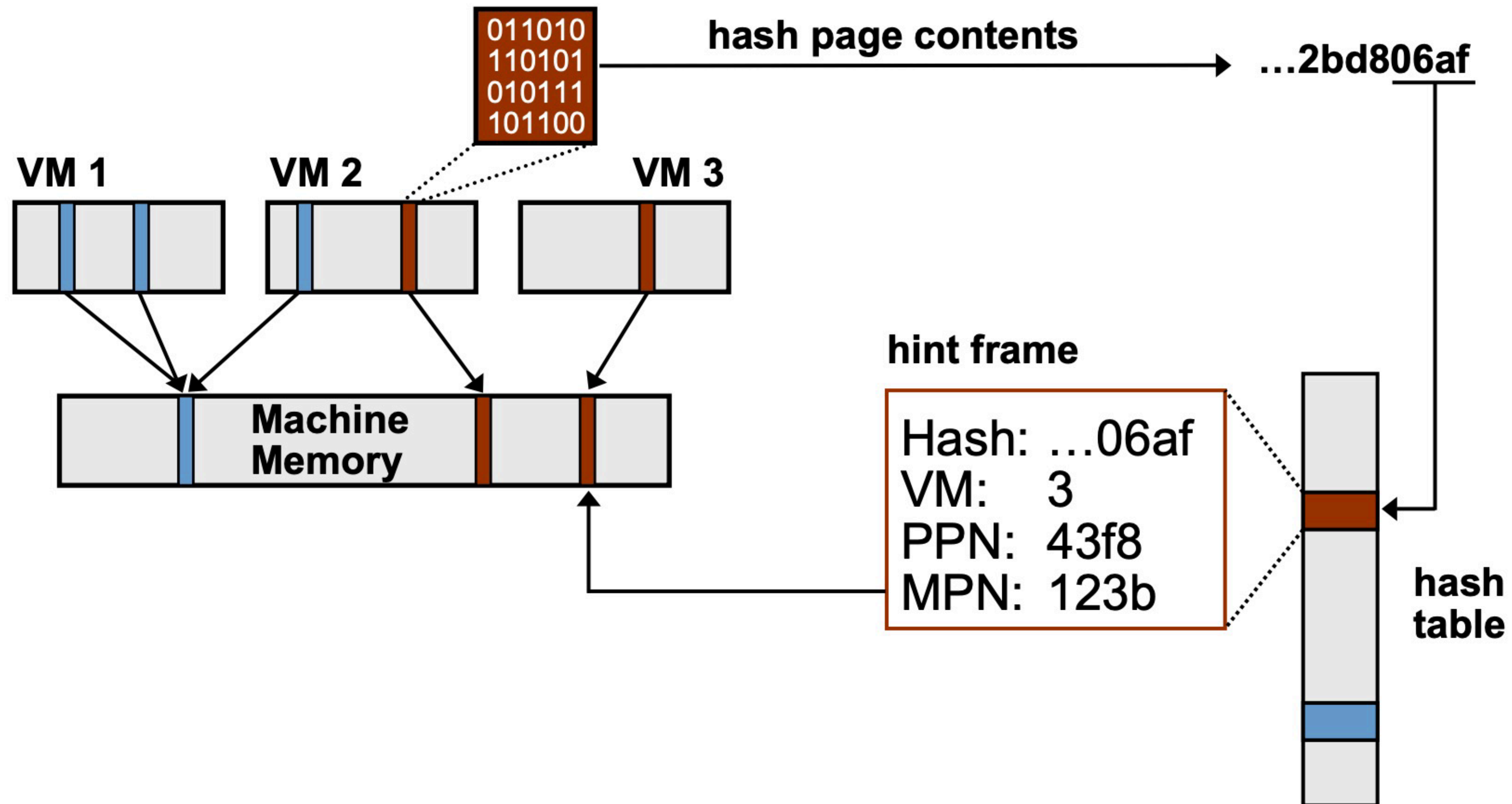


Figure 2: **Balloon Performance.** Throughput of single Linux VM running dbench with 40 clients. The black bars plot the performance when the VM is configured with main memory sizes ranging from 128 MB to 256 MB. The gray bars plot the performance of the same VM configured with 256 MB, ballooned down to the specified size.

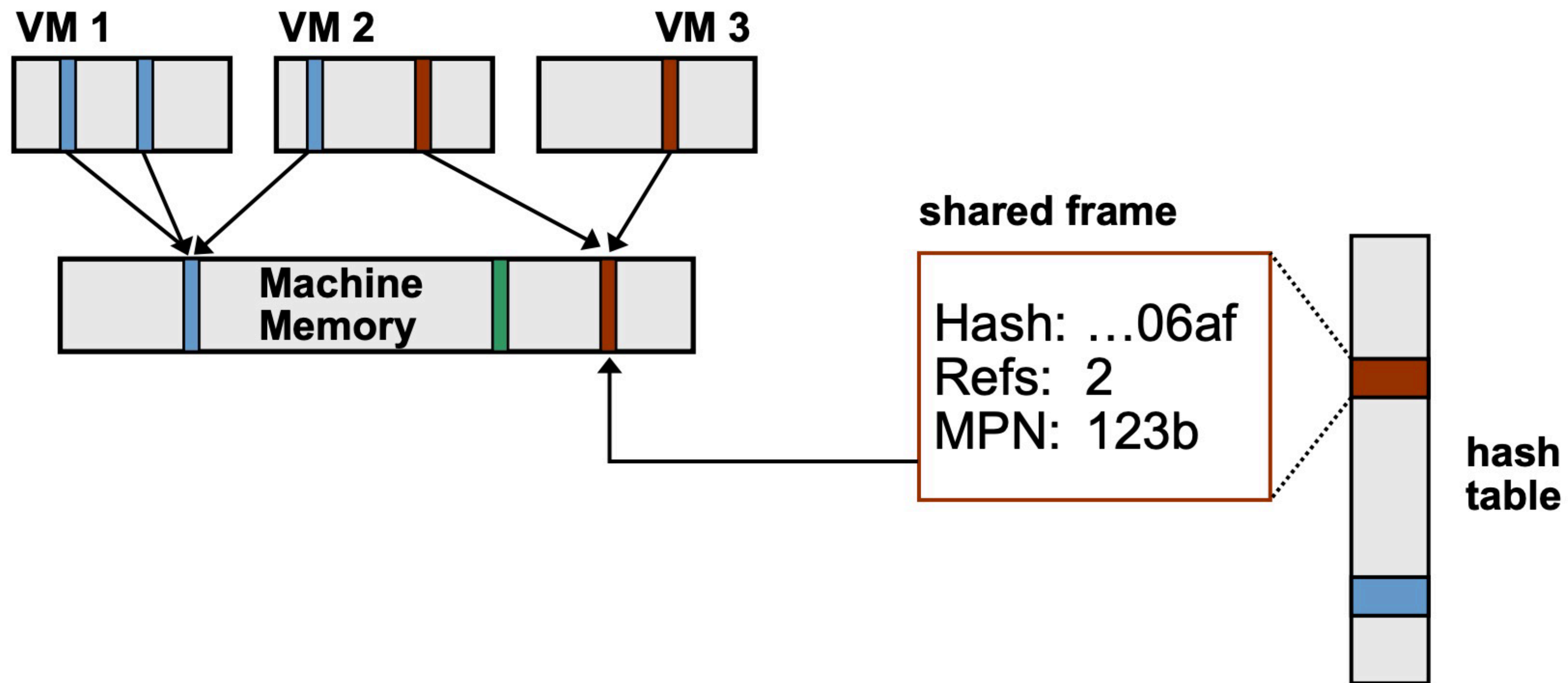
Memory Sharing

- Motivation
 - Multiple VMs running same OS, apps
 - Collapse redundant copies of code, data, zeros
- Transparent page sharing
 - Map multiple PPNs to single MPN (copy-on-write)
 - Pioneered by Disco, but required guest OS hooks
- New twist: content-based sharing
 - General-purpose, no guest OS changes
 - Background activity saves memory over time

Page Sharing: Scan Candidate PPN



Page Sharing: Successful Match



Question

- What is the benefit of keeping a "hint" entry for each scanned (but unshared) page (as compared to not maintaining anything for the page)

Real-World Page Sharing Results

		Total	Saved	
Workload	Guest Types	MB	MB	%
Corporate IT	10 Windows	2048	673	32.9
Nonprofit Org	9 Linux	1846	345	18.7
VMware	5 Linux	1658	120	7.2

Corporate IT – database, web, development servers (Oracle, Websphere, IIS, Java, etc.)

Nonprofit Org – web, mail, anti-virus, other servers (Apache, Majordomo, MailArmor, etc.)

VMware – web proxy, mail, remote access (Squid, Postfix, RAV, ssh, etc.)

Conclusion

- Software and hardware solutions for memory virtualization both have pros and cons
- More things to take care of besides the basic mechanism of memory virtualization
 - Allocation, sharing, overcommitment and reclamation