CS 134
Operating Systems

April 19, 2019

Virtualization

What is a virtual machine?

- Simulation of a computer
- Running as an application on a host computer
- Goals
  - Accurate
  - Isolated
  - Fast

Why use a virtual machine?

- To run multiple simultaneous operating systems (e.g, Windows and Linux)
- To manage big machines (allocate cores and memory at OS granularity)
- Kernel development (like QEMU and JOS)
- Better fault isolation (defense in depth)
- To package applications with a specific kernel version and environment
- To improve resource utilization

How accurate do we have to be?

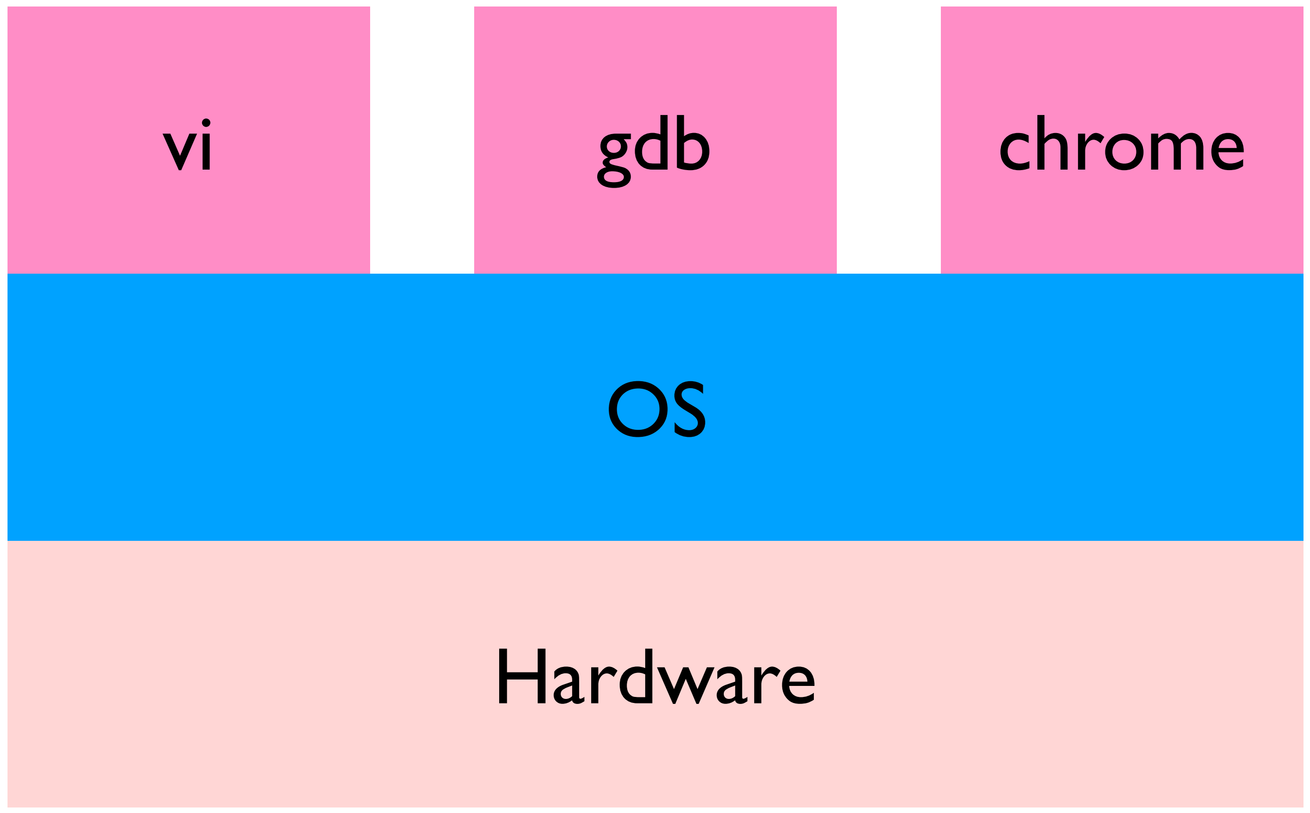
- Must handle weird quirks in existing OSes
  - Even bug-for-bug compatability
- Must maintain isolation against malicioius software
  - Guest can not break out of VM
- Must be impossible for guest to distinguish VM from real machine
  - Some VMs compromise, modifying the guest kernel to reduce accuracy requirement

VMs are an old idea

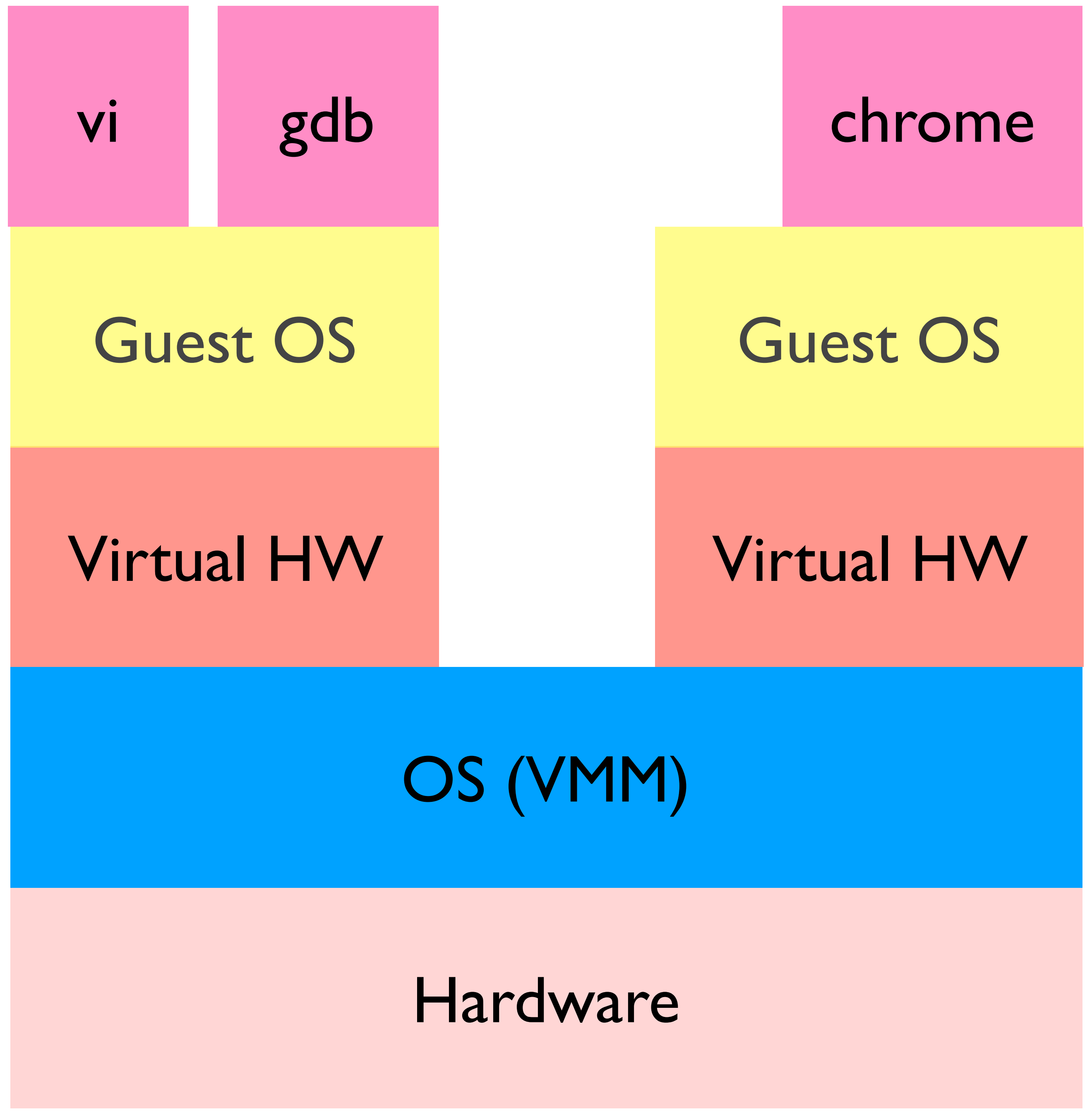
- 1960s: IBM used VMs to share big machines
- 1970s: IBM specialized CPUs for virtualization
- 1990s: VMware repopularized VMs for x86 HW
- 2000s: AMD & Intel specialized CPUs for virtualization

AMD-V, Intel VT-x

Process architecture



VM architecture



VMM aka *Hypervisor*

The abstraction provided by the VMM is the HW layer

Process vs HW

Process	HW
Non-privileged registers and instructions	All registers and instructions
Virtual memory	Virtual memory and MMU
Signals	Traps & interrupts
File system and sockets	I/O devices and DMA

Can a CPU be virtualized?

- Requirements to be "classicaly virtualizable" defined by Popek and Goldberg in 1974:

  1. **Fidelity**: Software on the VMM executes identically to its execution on hardware (barring timing effects)

  2. **Performance**: An overwhelming majority of guest instructions are executed by the hardware without the intervention of the VMM

  3. **Safety:** The VMM manages all hardware resources

Why not simulation?

- VMM interprets each instruction (e.g., Bochs)
- Maintain machine state for each register
- Emulate I/O ports and memory
- Violates *performance* requirement

Idea: execute guest instructions on real CPU whenever possible

- Works fine for most instructions
- E.g., `add %eax, %ebx`
- But privileged instructions could be harmful
- Would violate *safety* requirement

Idea: run guest kernels at CPL 3

- Ordinary instructions work fine
- Privileged instructions should trap to VMM (general protection fault)
- VMM can apply privileged operations on "virtual" state, not to real hardware
- This is called *trap-and-emulate*

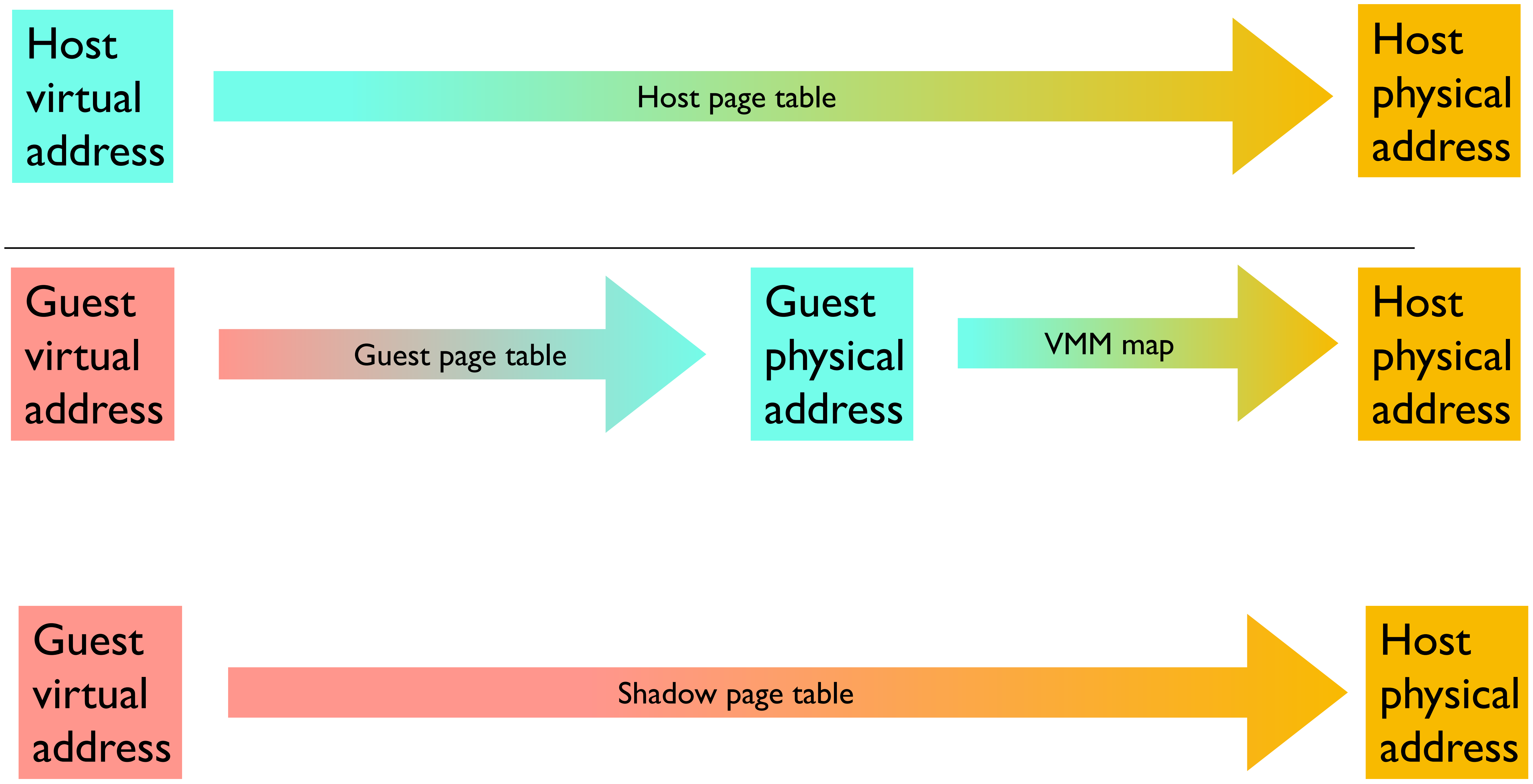
Trap-and-emulate example

- `CLI/STI`—enables and disables interrupts
- `EFLAGS IF` bit tracks current status
- VMM maintains virtual copy of `EFLAGS` register
- VMM controls hardware `EFLAGS`
  - Probably leave interrupts enabled even if guest disables them
- VMM looks at virtual `EFLAGS` to determine whether or not to interrupt guest
- VMM must make sure that guest sees only virtual `EFLAGS`

What about virtual memory?

- Want to maintain illusion that each VM has dedicated physical memory
- Guest wants to start at PA 0 and use all of RAM
- VMM needs to support many guests; they can't all use the same physical addresses
- Idea:
  - Claim RAM is smaller than real RAM
  - Keep paging enabled
  - Maintain a "shadow" copy of guest page table
  - Shadow maps VAs to different PAs than guest requests
  - Real `%CR3` register points to shadow page table
  - Virtual `%CR3` register points to guest page table

Virtualization memory diagram



Example

- Guest wants guest-physical page @ 0x10000000
- VMM map redirects guest-physical 0x10000000 to host-physical 0x20000000
- VMM traps if guest changes %CR3 or writes to guest page table
- Transfers each guest PTE to shadow page table
- Uses VMM map to translate guest-physical addresses in shadow page table to *host-physical* addresses

Why can’t the VMM modify the guest page table in place?

Trap-and-emulate not possible on x86

- Two problems:
  1. Some instructions behave differently in CPL 3 instead of trapping

  2. Some register leak state that reveals if the CPU is running in CPL 3

Violates *fidelity* requirement

x86 isn’t classically virtualizable

- Problems in different behavior CPL 3 vs. CPL 0:

  - `mov %cs, %eax`
    - `%cs` contains the CPL in its two lower bits

  - `popfl/pushfl`
    - Privileged bits, including `EFLAGS.IF`, are masked out

  - `iret`
    - No ring change, so doesn’t restore `SS/ESP`

Two possible solutions

- Binary translation
  - Rewrite offending instructions to behave correctly

- Hardware virtualization
  - Extend x86 to make it classically virtualizable

Naive binary translation

- Replace all instructions that can cause violations with INT 3, which traps
- INT 3 is one byte, so can fit inside any x86 instruction without changing size/layout
- But, unrealistic
  - We don’t know, at load time, the difference between code and data or where instruction boundaries lie
  - VMware’s solution is much more sophisticated

VMware's binary translator

- Kernel translated dynamically (like a JIT compiler)
  - Idea: scan only as executed, since execution reveals instruction boundaries
  - When VMM first loads guest kernel, translate from entrypoint to first jump
  - Most instructions translate identically
- Need to translate instructions in chunks
  - Called a *basic block*
  - Either 12 instructions or a control flow instruction, whichever happens first
- Only guest kernel code is translated
  - Only if in CPL 0

Guest kernel shares address space with VMM

- Uses segmentation to protect VMM memory
- VMM loaded at high virtual addresses, translated guest kernel at low addresses
- Program segment limits to “truncate” address space, preventing all segments from accessing VMM except `%GS`
  - What if guest VM uses `%GS` selector?
  - `%GS` provides fast access to data shared between guest kernel and VMM
- Assumption: translated code can’t violate isolation
  - Can never directly access `%GS`, `%CR3`, `GDT`, etc.

Why put guest and VMM in same address space?

- Shared state becomes inexpensive to access
  - e.g., `cli` → "`vcpu.flags.IF = 0`"
- Translated code is safe, can't violate isolation (after translation)

Binary translation example

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if ((a % i) == 0) return 0;
  }
  return 1;
}
```

```
prime: mov %ecx, %edi    # %ecx = %edi (a)
       mov %esi, 2       # %esi = 2
       cmp %esi, %ecx    # is i ≥ a?
       jge prime         # if yes, jump
nexti: mov %eax, %ecx    # set %eax = a
       cdq               # sign-extend
       idiv %esi         # a % i
       test %edx, %edx   # is remainder zero?
       jz notPrime       # jump if yes
       inc %esi          # i++
       cmp %esi, %ecx    # is i >= a?
       jl nexti          # jump if no
prime: mov %eax, $1      # return value in %eax
       ret
notPrime:
       xor %eax, %eax    # %eax = 0
       ret
```

Binary translation example (cont.)

All control flow requires indirection

translation unit (TU)

```
prime: mov %ecx, %edi    # %ecx = %edi (a)
       mov %esi, 2       # %esi = 2
       cmp %esi, %ecx    # is i ≥ a?
       jge prime         # if yes, jump
```

Translator

```
prime: mov %ecx, %edi            # IDENT
       jge [takenAddr]           # JCC
       jmp [fallthroughAddr] #JCC
```

Compiled code fragment (CCF)

Executes this jump:
  Runs translator on code at fallthroughAddr
  Normally replaces address with address of CCF
  In this case since it’s the next CCF generarted,
    elides the jump
    (and just falls through to next CCF)

Non-IDENT instructions

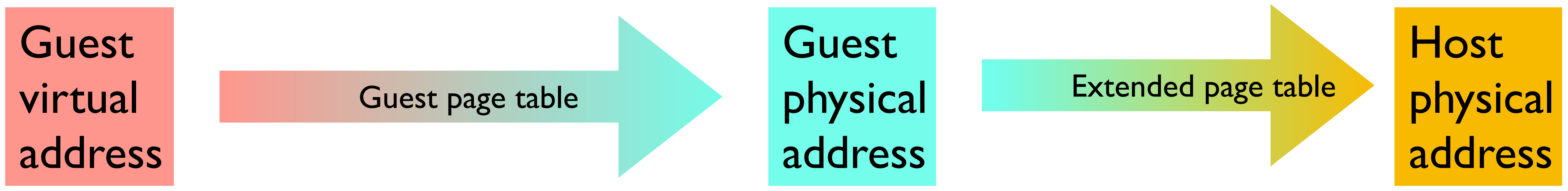
- Privileged instructions
- PC-relative addressing
  - Since code layout changes
- Direct control flow (direct JMP, CALL)
  - Since code layout changes
  - Binds target address at translation time
- Indirect control flow (RET, indirect JMP, indirect CALL)
  - Must bind target address at runtime (using a hash table lookup)

Hardware virtualization

- CPU maintains guest copy of privileged state in a special region called the Virtual Machine Control Block (VMCB)
- CPU operates in two modes:
  - VMX guest mode: runs guest kernel
    - Switch from host mode to guest mode new instruction: `vmrun`
  - VMX host: runs VMM
    - Switch from guest mode to host mode (I/O, for example)
  - Hardware saves and restores privileged register state to and from the VMCB as it switches modes
  - Each mode has its own separate privilege rings
- Net effect: hardware can run most privileged guest instructions directly without emulation

Virtualization memory diagram

- Hardware effectively manages two page tables
- Normal page table controlled by guest kernel
- Extended page table (EPT) controlled by VMM
- EPT didn't exist at the time of the VMware paper
-



What’s better: HW or SW virtualization?

- Software virtualization advantages
  - **Trap emulation**: most traps can be replaced with callouts
  - **Emulation speed**: BT can generate purpose-built emulation code with predecoded instruction
  - **Callout avoidance**: dometimes BT can even inline callouts
- Hardware virtualization advantages
  - **Code density**: translated code requires more instructions
  - **Precise exceptions**: BT must perform extra work to recover guest state
  - **System calls**: don’t require VMM intervention

What's better: HW or SW virtualization?

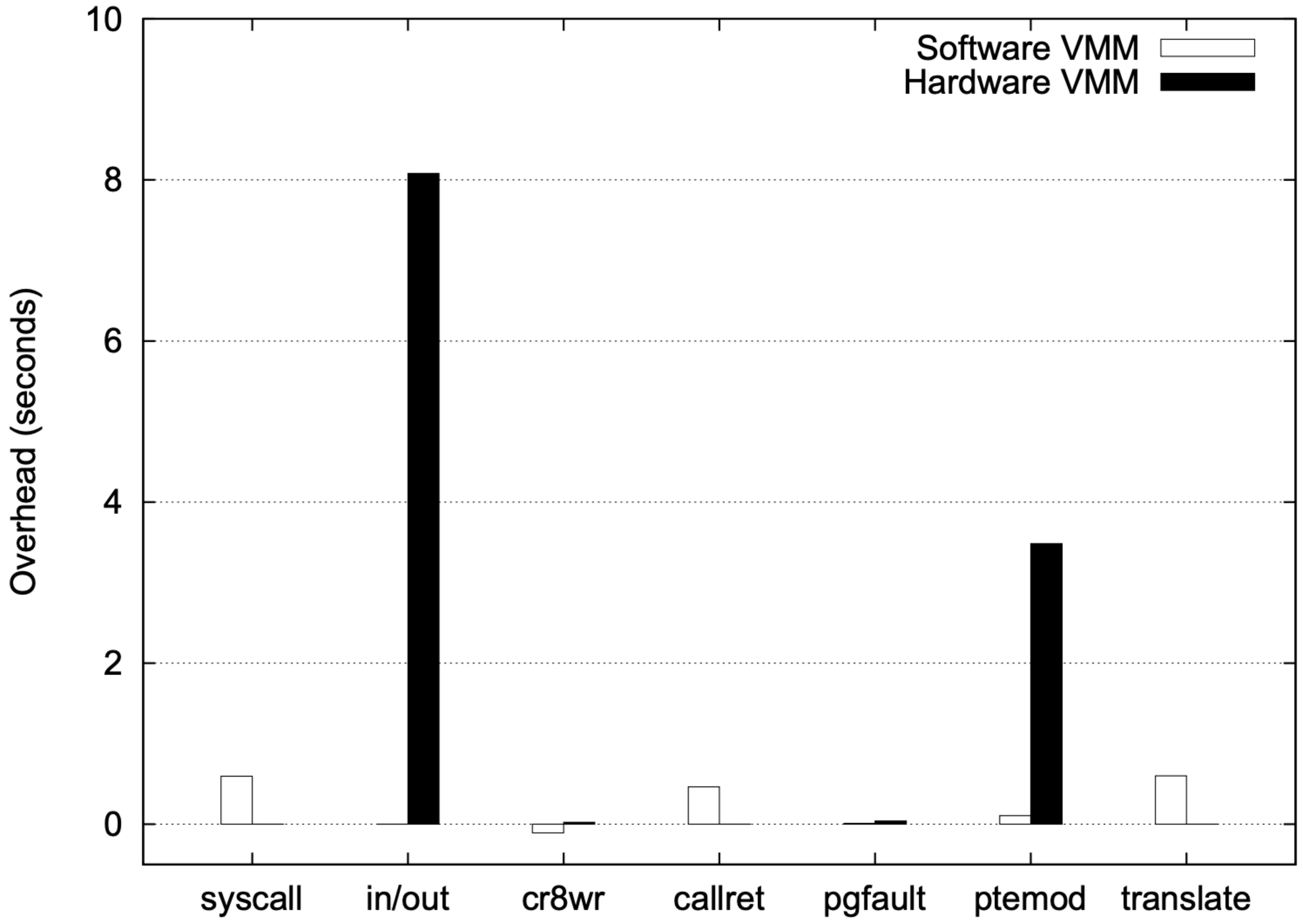


Figure 5. Sources of virtualization overhead in an **XP** boot/halt.

What’s better? Shadow page table or EPT?

- EPT is faster when page table contents change frequently
  - Fewer traps to VMM
- Shadow page table is faster when page table is stable
  - Less TLB-miss overhead
  - One page table to walk through instead of two

Conclusion

- Virtualization transformed cloud computing
- VMware made virtualization possible (through BT) on an architecture that couldn’t be virtualized (x86)
- Prompted Intel and AMD to change hardware: sometimes faster (though sometimes slower) than BT

What’s changed since the paper was written?

- HW virtualization became much faster
  - Fewer traps, better microcode, more dedicated logic
  - Almost all CPU architectures support HW virtualization
  - EPT widely available
- VMMs became commoditized
  - BT technology was hard to build
  - VMMs based on HW virtualization are much easier to implement (Xen, KVM, HyperV, etc.)
- I/O devices aren’t just emulated, they can be exposed directly
  - IOMMU provides paging protection for DMA

Questions

- How do shadow structures stay updated with primary structures?
- How are instructions that cause an exception forwarded to the VMM to handle?
- Where is the information about the registers in a virtual CPU stored: registers or memory?
- Does binary translation work for more complex virtualization (e.g., a language like Java on a JVM)?
- What is the difference between true and hidden page faults?
- Authors mention similarities to RISC/CISC debate. How is this similar?

Questions

- Any difference in security between hardware-supported or software-only virtualization?