Virtualization

Adam Belay <abelay@mit.edu>
What is a virtual machine

• Simulation of a computer
• Running as an application on a host computer
• Accurate
• Isolated
• Fast
Why use a virtual machine?

• To run multiple operating system (e.g. Windows and Linux)
• To manage big machines (allocate cores and memory at O/S granularity)
• Kernel development (e.g. like QEMU + 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 compatibility
• Must maintain isolation with malicious 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
Process Architecture

Hardware

OS

vi
gcc
firefox
• What if the process abstraction looked just like HW?
Comparing a process and HW

Process
• Non privileged registers and instructions
• Virtual memory
• Signals
• File system and sockets

Hardware
• All registers and instructions
• Virt. mem. and MMU
• Traps and interrupts
• I/O devices and DMA
Can a CPU be virtualized?

Requirements to be “classically 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 property
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 enables even if VM runs CLI
- VMM looks at virtual EFLAGS register to decide when to interrupt guest
- VMM must make sure guest only sees virtual EFLAGS
What about virtual memory?

• Want to maintain illusion that each VM has dedicated physical memory
• Guest wants to start at PA 0, use all of RAM
• VMM needs to support many guests, they can’t all really 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 PA than guest requests
  • Real %CR3 points to shadow table
  • Virtual %CR3 points to guest page table
Virtualization memory diagram

Host Virtual Address → Host Page Table → Host Physical Address
Virtualization memory diagram

**Host Virtual Address** → **Host Page Table** → **Host Physical Address**

**Guest Virtual Address** → **Guest PT** → **Guest Physical Address** → **VMM Map** → **Host Physical Address**

**Guest Virtual Address** → **Shadow Page Table** → **Host Physical Address**
Example:

• Guest wants *guest-physical* page @ 0x1000000
• VMM map redirects *guest-physical* 0x1000000 to *host-physical* 0x2000000
• 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* page addresses in page table to *host-physical* addresses
Why can’t the VMM modify the guest page table in-place?
Need shadow copy of all privileged state

• So far discussed EFLAGS and page tables
• Also need GDT, IDT, LDTR, %CR*, etc.
Unfortunately trap-and-emulate is not possible on x86

Two problems:

1. Some instructions behave differently in CPL 3 instead of trapping
2. Some registers leak state that reveals if the CPU is running in CPL 3
   • Violates *fidelity* property
x86 isn’t classically virtualizable

Problems -> CPL 3 versus CPL 0:

- mov %cs, %ax
  - %cs contains the CPL in its lower two bits
- popfl/pushfl
  - Privileged bits, including EFLAGS.IF are masked out
- iretq
  - No ring change, so doesn’t restore SS/ESP
Two possible solutions

1. Binary translation
   - Rewrite offending instructions to behave correctly

2. Hardware virtualization
   - CPU maintains shadow state internally and directly executes privileged guest instructions
Strawman binary translation

- Replace all instructions that 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
  - Don’t know 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
  • idea: scan only as executed, since execution reveals instruction boundaries
  • when VMM first loads guest kernel, rewrite from entry to first jump
  • Most instructions translate identically

• Need to translate instructions in chunks
  • Called a basic block
  • Either 12 instructions or the control flow instruction, whichever occurs first

• Only guest kernel code is translated
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 kernel instruction 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 kernel and VMM in same address space?
Why put guest kernel 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
Translation example

• All control flow requires indirection

Original: isPrime()

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

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

End of basic block
```
Translation example

• All control flow requires indirection
• Original: isPrime()

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

Translated: isPrime()’

```
mov %ecx, %edi  # IDENT
mov %esi, $2
 cmp %esi, %ecx
 jge [takenAddr]  # JCC
 jmp [fallthrAddr]
...
```
Translation example

• Brackets represent continuations
• First time they are executed, jump into BT and generate the next basic block
• Can elide “jmp [fallthraddr]” if it’s the next address translated
• Indirect control flow is harder
  • “(jmp, call, ret) does not go to a fixed target, preventing translation-time binding. Instead, the translated target must be computed dynamically, e.g., with a hash table lookup. The resulting overhead varies by workload but is typically a single-digit percentage.” – from paper
Hardware virtualization

- CPU maintains guest-copy of privileged state in special region called the virtual machine control structure (VMCS)
- CPU operates in two modes
  - VMX non-root mode: runs guest kernel
  - VMX root mode: runs VMM
  - Hardware saves and restores privileged register state to and from the VMCS as it switches modes
  - Each mode has its own separate privilege rings
- Net effect: Hardware can run most privileged guest instructions directly without emulation
What about MMU?

- Hardware effectively maintains two page tables
- Normal page table controlled by guest kernel
- Extended page table (EPT) controlled by VMM
- EPT didn’t exist when VMware published paper

**Diagram:**

- Guest Virtual Address → Guest PT → Guest Physical Address → EPT → Host Physical Address
What’s better HW or SW virt?
What’s better HW or SW virt?

• Software virtualization advantages
  • **Trap emulation**: Most traps can be replaced with callouts
  • **Emulation speed**: BT can generate purpose-built emulation code, hardware traps must decode the instruction, etc.
  • **Callout avoidance**: Sometimes BT can even inline callouts

• Hardware virtualization advantages
  • **Code density**: Translated code requires more instructions and larger opcodes
  • **Precise exceptions**: BT must perform extra work to recover guest state
  • **System calls**: Don’t require VMM intervention
What’s better HW or SW virt?

![Bar chart showing overhead of virtualization tasks for Software VMM (white bars) and Hardware VMM (black bars).](chart.png)

**Figure 5.** Sources of virtualization overhead in an XP boot/halt.
What’s better shadow page table or EPT?
What’s better shadow page table or EPT?

- EPT is faster when page table contents change frequently
  - Fewer traps
- 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, had a tremendous impact
  • Virtualization on PCs was also big, but less significant
• VMware made virtualization possible on an architecture that couldn’t be virtualized (x86) through BT
• Prompted Intel and AMD to change hardware, sometimes faster, sometimes slower than BT
A decade later, what’s changed?

- HW virtualization became much faster
  - Fewer traps, better microcode, more dedicated logic
  - Almost all CPU architectures support HW virt.
  - EPT widely available
- VMMs became commoditized
  - BT technology was hard to build
  - VMMs based on HW virt. 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