The Evolution of an x86 Virtual Machine Monitor

Haibo Chen
Institute of Parallel and Distributed Systems (IPADS)
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn/haibo_chen
Why Virtualization

Computing capacity increases year by year

  Moore’s Law
  Lower cost

Low utilization of computers

  Most are below 20%

Maintenance of machines is costly

  ~5X to 10X cost of HW/SW
Virtualization Benefits

Increased resource utilization
  Server consolidation
Mobility
Enhanced Security
Trusted Computing
Test and Deployment
Utilization: Server Consolidation
Mobility: Load Balance

Server 1
CPU Utilization = 90%

Server 2
CPU Utilization = 50%

VM Guest 1 | VM Guest 3 | VM Guest 4 | VM Guest 5 | **VM Guest 2** | VM Guest 6 | VM Guest 7
Testing and Deployment

Development VM → QA VM

→ Production VM
→ Production VM
→ Production VM
→ Production VM
System Virtualization

Allows multiple operating systems to run simultaneously on the same computer
Terminology

Virtual Machine Monitor (VMM)
- create isolated execution environments
- manage the real resources of the computer system, exporting them to virtual machines

Virtual machine (VM)
- a self-contained operating environment
- behave like a separate computer
Categories of Virtualization

Based on the Platform they are built upon

Type I – Host platform is the hardware
VMware’s ESX Server

Type II – Host platform is the host operating system
VMWare Workstation, Virtual PC, Virtual Server

<table>
<thead>
<tr>
<th>guest application</th>
<th>guest application</th>
<th>guest application</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest operating system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual-machine monitor (VMM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>host hardware</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>guest application</th>
<th>guest application</th>
<th>guest application</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest operating system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual-machine monitor (VMM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>host operating system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>host hardware</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type I VMM

Type II VMM
Virtualization Approaches

Traditional trap-and-emulate
- Guest code runs with reduced privilege
- Cause trap when accessing privilege state
- VMM emulates the instruction using interpreter
- Guest resumes execution

Not suitable for x86
- Some instructions just silently fail
  - e.g., popf, load a set of flags from stack into %eflags
Virtualization Approaches

Run guest code in a full interpreter

Interpret guest code instruction by instruction
    Introduce great overhead

Available, but rather too slow
Virtualization Approaches

Binary Translation (BT)
VMware’s approach when there’s no hw support

Translate guest kernel code into plain one
Guest user code runs directly

Flexible as full interpretation, but with much lower overhead
Binary Translation Properties

Binary
Input is binary x86 code, not source code

Dynamic
Happens at runtime, interleaved with guest execution

On Demand
Code is translated when it’s about to execute

System Level
Input is the full x86 ISA

Subsetting
Output is a subset (mostly user-mode instructions)
Binary Translation

Each translator invocation
  Consume one input Translation Unit
  Produce one output Compiled Code Fragment
Store output in Translation Cache
  For future reuse to amortize translation costs
### Binary Translation Example

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push %ebx</code></td>
<td>callee saved</td>
</tr>
<tr>
<td><code>mov %eax,%edx</code></td>
<td>%edx = %eax = lock</td>
</tr>
<tr>
<td><code>cli</code></td>
<td>disable interrupts</td>
</tr>
<tr>
<td><code>mov $1,%ecx</code></td>
<td>%ecx = 1</td>
</tr>
<tr>
<td><code>xor %ebx,%ebx</code></td>
<td>%ebx = 0</td>
</tr>
<tr>
<td><code>jmp doTest</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push %ebx</code></td>
<td>callee saved</td>
</tr>
<tr>
<td><code>mov %eax,%edx</code></td>
<td>%edx = %eax = lock</td>
</tr>
<tr>
<td><code>and $0x{a0d},%gs:vcpu.flags</code></td>
<td>disable interrupts</td>
</tr>
<tr>
<td><code>mov $1,%ecx</code></td>
<td>%ecx = 1</td>
</tr>
<tr>
<td><code>xor %ebx,%ebx</code></td>
<td>%ebx = 0</td>
</tr>
<tr>
<td><code>jmp [translator]</code></td>
<td></td>
</tr>
</tbody>
</table>
Virtual Memory
MMU Virtualization

Two Purpose

Maintain isolation between guest and VMM or other guests

Ensure memory visible to guest corresponds to its own guest page table
Three Levels of Addresses
Shadow Page Table

x86 cpu has only one page table

VMM maintains a shadow page table that merges these two mappings

  %cr3 points to shadow page table

Get adapted on changes to virtual memory layout
Shadow Page Table
Trace Mechanism

Why need trace

Implicit TLB flush semantic when switching %cr3

How to trace

Mark shadow page table entries pointing to guest page table as read-only

Subsequent guest page table modification would cause a page fault
Eager Validate

Why “eager validate”

Reduce hidden page faults (mappings are valid in guest page table, but not present in shadow page table)

Combine trace and revalidation into one step

Drop and reload entries immediately when guest page table is modified

Via instruction emulation
3-way Performance Trade-off in Shadow Page Tables

Trace costs
VMM must intercept Guest writes to guest page tables
Propagate change into shadow page table (invalidate)

Page fault costs
VMM must intercept page faults
Validate shadow (hidden page fault), or forward fault to guest (true fault)

Context switch costs
VMM must intercept CR3 writes
Activate new set of shadow page tables
Protect VMM Address Space

Segment-truncation protecting the upper 4 MB containing the VMM address-space
Hardware Virtualization

Key feature: root vs. guest CPU mode
  VMM executes in root mode
  Guest (OS and apps) execute in guest mode
  Hardware-defined VMCS/VMCB holds guest state

VMM and Guest run as “co-routines”
  VM enter
  Guest runs
  A while later: VM exit
  VMM runs

Very much like classical trap-and-emulate
  Guest-invisible deprivileging
Hardware Virtualization

VMX Non-Root Operation

VM Exit

VMX Root Operation

vmlaunch/vmresume
How VMM Controls Guest Execution

Hardware-defined structure
- Intel: VMCS (virtual machine control structure)
- AMD: VMCB (virtual machine control block)

VMCS/VMCB contains
- Guest state

Control bits that define conditions for exit
- Exit on IN, OUT, CPUID, ...
- Exit on write to control register %cr3
- Exit on page fault, pending interrupt, ...

VMM uses control bits to “confine” and observe guest
Analysis of HV-based VMM

VMM only intervenes to handle exits

Same performance as classical trap-and-emulate exit frequency * (average exit cost + average handling cost)

VMCB/VMCS can avoid simple exits but many remain
  Page table updates
  Context switches
  In/out
  Interrupts
Nested Page Table

TLB

<table>
<thead>
<tr>
<th>VA</th>
<th>MA</th>
</tr>
</thead>
</table>

TLB fill hardware

Guest PT ptr

Nested PT ptr

VA → PA mapping

guest

VMM

PA → MA mapping
Analysis of NPT

Dynamic
MMU composes VA→PA and PA→MA mappings on the fly at TLB fill time

Benefits
Significant reduction in exit frequency
No shadow page table memory overhead
Better scalability to high vcpu count

Costs
More expensive TLB misses
$O(n^2)$ cost for page table walk ($n =$ depth of the page table tree)
Conclusions

The evolution of x86 virtualization
Traditional trap-and-emulate is not suitable for x86

VMware handled this using Binary Translation
Convert guest kernel code into a x86 subset
MMU virtualization: Shadow page table

Hardware virtualization
Root mode and non-root mode
Nested page table