Virtualizing CPU

Yiying Zhang
Logistics

• Project topics are out

• Form project group by this Wed => Fri

• Office hours announced and start this week

• Attendance and paper summary start today

Acknowledgment: slides adapted from Columbia E6998 Lecture 2 by Scott Devine
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

• Can handle all types of instructions, but super slow
Recap: Virtualization Approach 2: Direct Execution with Trap-and-Emulate
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)
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))
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))
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))

Goldberg (1974) two classes of instructions

- **privileged instructions**: those that trap when in user mode

- **sensitive instructions**: those that modify or depends on hardware configs
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!

Goldberg (1974) two classes of instructions

- *privileged instructions*: those that trap when in user mode

- *sensitive instructions*: those that modify or depends on hardware configs
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!

Goldberg (1974) two classes of instructions

– *privileged instructions*: those that trap when in user mode

– *sensitive instructions*: those that modify or depends on hardware configs
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!
- When guest OS executes a *privileged instruction*, will *trap* into VMM

Goldberg (1974) two classes of instructions

- *privileged instructions*: those that trap when in user mode
- *sensitive instructions*: those that modify or depends on hardware configs
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!

• When guest OS executes a privileged instruction, will trap into VMM

• When guest applications generates a software interrupt, will trap into VMM

Goldberg (1974) two classes of instructions

- privileged instructions: those that trap when in user mode
- sensitive instructions: those that modify or depends on hardware configs
Trap-and-Emulate

• Goal: hand off sensitive operations to the VMM

• Reality: privileged operations trap to VMM

• VMM emulates the effect of privileged operations on virtual hardware provided to the guest OS
  • VMM controls how the VM interacts with physical hardware
  • VMM fools the guest OS into thinking that it runs at the highest privilege level

• Performance implications
  • Almost no overhead for non-privileged instructions
  • Large overhead for privileged instructions
Trap-and-Emulate

- Guest OS + Applications
- Virtual Machine Monitor
  - Page Fault
  - Undef Instr
  - vIRQ
- MMU Emulation
- CPU Emulation
- I/O Emulation

Unprivileged

Privileged
open:
  push    dword mode
  push    dword flags
  push    dword path
  mov     eax, 5
  push    eax
  int     80h

<table>
<thead>
<tr>
<th>Process</th>
<th>Hardware</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Execute instructions (add, load, etc.)</td>
<td>3. Switch to kernel mode;</td>
<td>4. In kernel mode;</td>
</tr>
<tr>
<td>2. System call:</td>
<td>Jump to trap handler</td>
<td>Handle system call;</td>
</tr>
<tr>
<td>Trap to OS</td>
<td></td>
<td>Return from trap</td>
</tr>
<tr>
<td>6. Resume execution (@PC after trap)</td>
<td>5. Switch to user mode;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return to user code</td>
<td></td>
</tr>
</tbody>
</table>

Figure B.1: Executing a System Call
## System Calls with Virtualization

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
<th>VMM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. System call:</strong></td>
<td>Trap to OS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. OS trap handler: Decode trap and execute appropriate syscall routine; When</td>
<td>2. Process trapped: Call OS trap handler (at reduced</td>
</tr>
<tr>
<td></td>
<td>done: return from trap</td>
<td>privilege)</td>
</tr>
<tr>
<td></td>
<td>3. Resume execution (@PC after trap)</td>
<td></td>
</tr>
<tr>
<td>**Figure B.2: System Call Flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Without Virtualization</td>
<td></td>
</tr>
<tr>
<td><strong>1. System call:</strong></td>
<td>Trap to OS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Process trapped: Call OS trap handler (at reduced privilege)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. OS trap handler: Decode trap and execute syscall; When done: issue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return-from-trap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. OS tried return from trap: Do real return from trap</td>
<td></td>
</tr>
<tr>
<td>**5. Resume execution ( @PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>after trap)</td>
<td></td>
</tr>
<tr>
<td>**Figure B.3: System Call Flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with Virtualization</td>
<td></td>
</tr>
</tbody>
</table>
x86 Difficulties

Popek and Goldberg’s Theorem (1974)

– A machine can be virtualized (using trap-and-emulate) if every sensitive instruction is privileged.
x86 Difficulties

Popek and Goldberg’s Theorem (1974)
– A machine can be virtualized (using trap-and-emulate) if every sensitive instruction is privileged.

• Not all sensitive instructions are privileged with x86 for many years, i.e., non-virtualizable processor

• These instructions do not trap and behave differently in kernel and user mode

• Example: \textit{popf}
  • Pops 16 bits from top of the stack to the \texttt{eflags} register
  • Bit 9 of \texttt{eflags} masks interrupts (i.e., enables/disables interrupts)
  • \textit{popf} is not privileged. What happens if guest OS (ring 1) runs \textit{popf} to \texttt{eflags}?
  • In Ring 0, \textit{popf} can set bit 9, but CPU silently ignores \textit{popf}’s setting of system flags (bit 9) when running in Ring 1
  • What should happen is a trap so that VMM can emulates interrupts (change which interrupts to forward to guest OS)
Trap-and-Emulate

• Pros and Cons?
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)
Basic Idea of Binary Translation

• Based on input guest binary, compile (translate) instructions in a cache
• and run them directly

• Challenges:
  • Protection of the cache
  • Correctness of direct memory addressing
  • Handling relative memory addressing (e.g., jumps)
  • Handling sensitive instructions
VMware’s Dynamic Binary Translation

- **Binary**: Input is binary x86 code
- **Dynamic**: Translation happens at runtime
- **On demand**: Code is translated only when it is about to execute
- **System level**: Rules set by x86 ISA, not higher-level ABIs
- **Subsetting**: Output a safe subset of input full x86 instruction set
- **Adaptive**: Translated code is adjusted according to guest behavior changes
Translation Unit
Translation Unit

- TU: 12 instructions or a “terminating” instruction (a basic code block) implemented by the VMM
Translation Unit

- TU: 12 instructions or a “terminating” instruction (a basic code block)
- Why TU as the unit not individual instruction?

implemented by the VMM
Translation Unit

• TU: 12 instructions or a “terminating” instruction (a basic code block)

• Why TU as the unit not individual instruction?

• TU -> Compiled Code Fragment (CCF)

• CCF stored in translation cache (TC)

• At the end of each CCF, call into translator to decide and translate the next TU (more optimization soon)
  • If the destination code is already in TC, then directly jumps to it
  • Otherwise, compiles the next CCF into TC

implemented by the VMM
Architecture of VMware’s Binary Translation
Basic Blocks

Guest Code

<table>
<thead>
<tr>
<th>vPC</th>
<th>mov ebx, eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>cli</td>
<td></td>
</tr>
<tr>
<td>and ebx, ~0xffff</td>
<td></td>
</tr>
<tr>
<td>mov ebx, cr3</td>
<td></td>
</tr>
<tr>
<td>sti</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

- **Straight-line code**
- **Control flow**
- **Basic Block**
IDENT/Non-IDENT Translation

• Most instructions can be translated IDENT, except for

• PC-relative address
• Direct control flow
• Indirect control flow
• Sensitive instructions
  • If already traps, then can be handled when it traps (more optimization soon to be discussed)
  • Otherwise, replace it with a call to the emulation function
Binary Translation

Guest Code

- `mov ebx, eax`
- `cli`
- `and ebx, ~0xffff`
- `mov ebx, cr3`
- `sti`
- `ret`

Translation Cache

- `mov ebx, eax`
- `call HANDLE_CLI`
- `and ebx, ~0xffff`
- `mov [CO_ARG], ebx`
- `call HANDLE_CR3`
- `call HANDLE_STI`
- `jmp HANDLE_RET`
Basic Binary Translator

```c
void BT_Run(void)
{
    CPUState.PC = _start;
    BT_Continue();
}

void BT_Continue(void)
{
    void *tcpc;
    tcpc = BTFindBB(CPUState.PC);
    if (!tcpc)
    {
        tcpc = BTTranslate(CPUState.PC);
    }
    RestoreRegsAndJump(tcpc);
}

void *BTTranslate(uint32 pc) {
    void *start = TCTop;
    uint32 TCPC = pc;

    while (1) {
        inst = Fetch(TCPC);
        TCPC += 4;

        if (IsPrivileged(inst)) {
            EmitCallout();
        } else if (IsControlFlow(inst)) {
            EmitEndBB();
            break;
        } else {
            /* ident translation */
            EmitInst(inst);
        }
    }
    return start;
}
```

Example of a CPU emulation function (store interrupt flag):

```c
void BT_CalloutSTI(BTSavedRegs regs)
{
    CPUState.PC = BTFindPC(regs.tcpc);
    CPUState.GPR[] = regs.GPR[];
    CPU_STI();
    CPUState.PC += 4;
    if (CPUState.IRQ && CPUState.IE) {
        CPUVector();
        BT_Continue();
        /* NOT_REACHED */
    }
    return;
}
```
Binary Translation

<table>
<thead>
<tr>
<th>Guest Code</th>
<th>Translation Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ebx, eax</td>
<td>mov ebx, eax</td>
</tr>
<tr>
<td>cli</td>
<td>mov [CPU_IE], 0</td>
</tr>
<tr>
<td>and ebx, ~0xffff</td>
<td>and ebx, ~0xffff</td>
</tr>
<tr>
<td>mov ebx, cr3</td>
<td>mov [CO_ARG], ebx</td>
</tr>
<tr>
<td>sti</td>
<td>call HANDLE_CR3</td>
</tr>
<tr>
<td>ret</td>
<td>mov [CPU_IE], 1</td>
</tr>
<tr>
<td>test [CPU_IRQ], 1</td>
<td>test [CPU_IRQ], 1</td>
</tr>
<tr>
<td>jne</td>
<td>jne</td>
</tr>
<tr>
<td>call HANDLE_INTS</td>
<td>call HANDLE_INTS</td>
</tr>
<tr>
<td>jmp HANDLE_RET</td>
<td>jmp HANDLE_RET</td>
</tr>
</tbody>
</table>
Binary Translation

<table>
<thead>
<tr>
<th>Guest Code</th>
<th>Translation Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ebx, eax</td>
<td>mov ebx, eax</td>
</tr>
<tr>
<td>cli</td>
<td>mov [CPU_IE], 0</td>
</tr>
<tr>
<td>and ebx, ~0xffff</td>
<td>and ebx, ~0xffff</td>
</tr>
<tr>
<td>mov ebx, cr3</td>
<td>mov [CO_ARG], ebx</td>
</tr>
<tr>
<td>sti</td>
<td>call HANDLE_CR3</td>
</tr>
<tr>
<td>ret</td>
<td>mov [CPU_IE], 1</td>
</tr>
<tr>
<td></td>
<td>test [CPU_IRQ], 1</td>
</tr>
<tr>
<td></td>
<td>jne</td>
</tr>
<tr>
<td></td>
<td>call HANDLE_INTS</td>
</tr>
<tr>
<td></td>
<td>jmp HANDLE_RET</td>
</tr>
</tbody>
</table>

Q: Is the translated code for cli faster or slower than original cli?
Controlling Control Flow
Controlling Control Flow

Guest Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
<td></td>
</tr>
<tr>
<td>jeq</td>
<td></td>
</tr>
<tr>
<td>add ebx, 18</td>
<td></td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
<td></td>
</tr>
<tr>
<td>mov [ecx], eax</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

Translation Cache

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
<td></td>
</tr>
<tr>
<td>jeq</td>
<td></td>
</tr>
<tr>
<td>call END_BB</td>
<td></td>
</tr>
<tr>
<td>add ebx, 18</td>
<td></td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
<td></td>
</tr>
<tr>
<td>mov [ecx], eax</td>
<td></td>
</tr>
<tr>
<td>call HANDLE_RET</td>
<td></td>
</tr>
</tbody>
</table>

eax == 0
Controlling Control Flow

test eax, 1  
jeq  
add ebx, 18  
mov ecx, [ebx]  
mov [ecx], eax  
ret

\[ \text{eax} == 0 \]

test eax, 1  
jeq  
jmp  
call END_BB  
add ebx, 18  
mov ecx, [ebx]  
mov [ecx], eax  
call HANDLE_RET
Controlling Control Flow

Guest Code

- test eax, 1
- jeq
- add ebx, 18
- mov ecx, [ebx]
- mov [ecx], eax
- ret

Translation Cache

- test eax, 1
- jeq
- jmp
- call END_BB
- add ebx, 18
- mov ecx, [ebx]
- mov [ecx], eax
- call HANDLE_RET
- mov [ecx], eax
- call HANDLE_RET

eax == 1
Controlling Control Flow
Adaptive Binary Translation

- Binary translation can outperform classical virtualization by avoiding traps


- What about sensitive instructions that are not privileged?

  - “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

Hardware w/o VT-x

Guest Applications

Guest OS

Hypervisor

Ring 0

Ring 1

Ring 3

Hardware w/ VT-x

Host Applications

VMX root Ring 3

VMX root Ring 0

VMX non-root Ring 0

VMX non-root Ring 3
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

Image source: https://www.anandtech.com/show/2480/9
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

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