Virtualizing CPU

Yiying Zhang
Logistics

• Attendance and paper summary start today
• Course Piazza created
• Lecture recordings available on Canvas -> Media Gallery
• Form project group by this Wed (email TA names, student IDs, email addrs)

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

The success of deep neural networks (DNNs) from Machine Learning (ML) has inspired domain specific architectures (DSAs) for them. Google’s first generation DSA offered 50x improvement over conventional architectures for ML inference in 2015. Google next built the first production DSA supercomputer for the much harder problem of training. Subsequent generations greatly improved performance of both phases. We start with ten lessons learned from such efforts.

The rapid growth of DNNs rightfully raised concerns about their carbon footprint. The second part of the talk identifies the “4Ms” (Model, Machine, Mechanization, Map) that, if optimized, can reduce ML training energy by up to 100x and carbon emissions up to 1000x. By improving the 4Ms, ML held steady at <15% of Google’s total energy use despite it consuming ~75% of its floating point operations. With continuing focus on the 4Ms, we can realize the amazing potential of ML to positively impact many fields in a sustainable way.

Biosketch

David Patterson received BA, MS, and PhD degrees from UCLA. He is a UC Berkeley Pardee professor emeritus, a Google distinguished engineer, the RIOS Laboratory Director, and the RISC-V International Vice-Chair. His most influential Berkeley projects likely were RISC and RAID. He received service awards for his roles as ACM President, Berkeley CS Division Chair, and CRA Chair and awards for his teaching. The most prominent of his seven books is Computer Architecture: A Quantitative Approach. He and his co-author John Hennessy shared the 2017 ACM A.M Turing Award, the 2021 BBVA Foundation Frontiers of Knowledge Award, and the 2022 NAE Charles Stark Draper Prize for Engineering. The Turing Award is often referred to as the “Nobel Prize of Computing” and the Draper Prize is considered a “Nobel Prize of Engineering.”
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

- 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

Guest OS + Applications

Unprivileged

Privileged

Virtual Machine Monitor

Page Fault

Undef Instr

vIRQ

MMU Emulation

CPU Emulation

I/O Emulation
## System Calls with Virtualization

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
<th>VMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. System call: Trap to OS</td>
<td>2. OS trap handler: Decode trap and execute appropriate syscall routine; When done: return from trap</td>
<td>2. Process trapped: Call OS trap handler (at reduced privilege)</td>
</tr>
<tr>
<td>3. Resume execution (@PC after trap)</td>
<td></td>
<td>3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. OS tried return from trap: Do real return from trap</td>
</tr>
<tr>
<td>5. Resume execution (@PC after trap)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure B.2: System Call Flow Without Virtualization**

**Figure B.3: System Call Flow with Virtualization**
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: `popf`
  • Pops 16 bits from top of the stack to the `%eflags` register
  • Bit 9 of `%eflags` masks interrupts (i.e., enables/disables interrupts)
  • `popf` is not privileged. What happens if guest OS (ring 1) runs `popf` to `%eflags`?
  • In Ring 0, `popf` can set bit 9, but CPU silently ignores `popf` when running in Ring 1
  • What should happen is a trap so that VMM can emulate 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
Architecture of VMware’s Binary Translation
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
Guest Code

mov ebx, eax
cli
and ebx, ~0xfff
mov cr3, ebx
sti
ret

Basic Blocks

Straight-line code

Control flow
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

<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>call HANDLE_CLI</td>
</tr>
<tr>
<td>and ebx, ~0xffff</td>
<td>and ebx, ~0xffff</td>
</tr>
<tr>
<td>mov cr3, ebx</td>
<td>mov [CO_ARG], ebx</td>
</tr>
<tr>
<td>sti</td>
<td>call HANDLE_CR3</td>
</tr>
<tr>
<td>ret</td>
<td>call HANDLE_STI</td>
</tr>
</tbody>
</table>

```
mov   ebx, eax
cli
and   ebx, ~0xfff
mov   cr3, ebx
sti
ret
```
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);
}

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

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;
}

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

Guest Code

- `test eax, 1`
- `jeq` (jump if equal)
- `add ebx, 18`
- `mov ecx, [ebx]`
- `mov [ecx], eax`
- `ret`

Translation Cache

- `test eax, 1`
- `jeq`
- `call END_BB`
- `call END_BB`

The diagram illustrates the control flow with `vEPC` (Virtual Execution Point Counter) pointing to the guest code and the translation cache providing the corresponding instructions.
Controlling Control Flow

Guest Code

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

eax == 0
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

eax == 0
```
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

Guest Code

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

Translation Cache

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

eax == 1
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 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

Hypervisor

Guest OS

Guest Applications

Ring 0

Ring 1

Ring 3

Hardware w/ VT-x

Hypervisor

Guest OS

VMX non-root

Host Applications

VMX root

Ring 0

Ring 1

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
  - \texttt{vmcall} instruction
  - EPT page faults (more next lecture)
  - Interrupts
  - Some sensitive instructions (configured in VMCS)
- VM entry
  - \texttt{vmlaunch} instruction: enter with a new VMCS
  - \texttt{vmresume} instruction: enter for the last VMCS
  - Typical vm exit/enter take ~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

![Graph showing CPU cycles for syscall, call/ret, divzero, in, cr8wr, pgfault, ptemod]  

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