



# *A Comparison of Software and Hardware Techniques for x86 Virtualization*

**Keith Adams and Ole Agesen**

**VMware**

**ASPLOS 2006**

**Presenter: Daiping Liu**



# Roadmap

**Overview**

**Classical Virtualization**

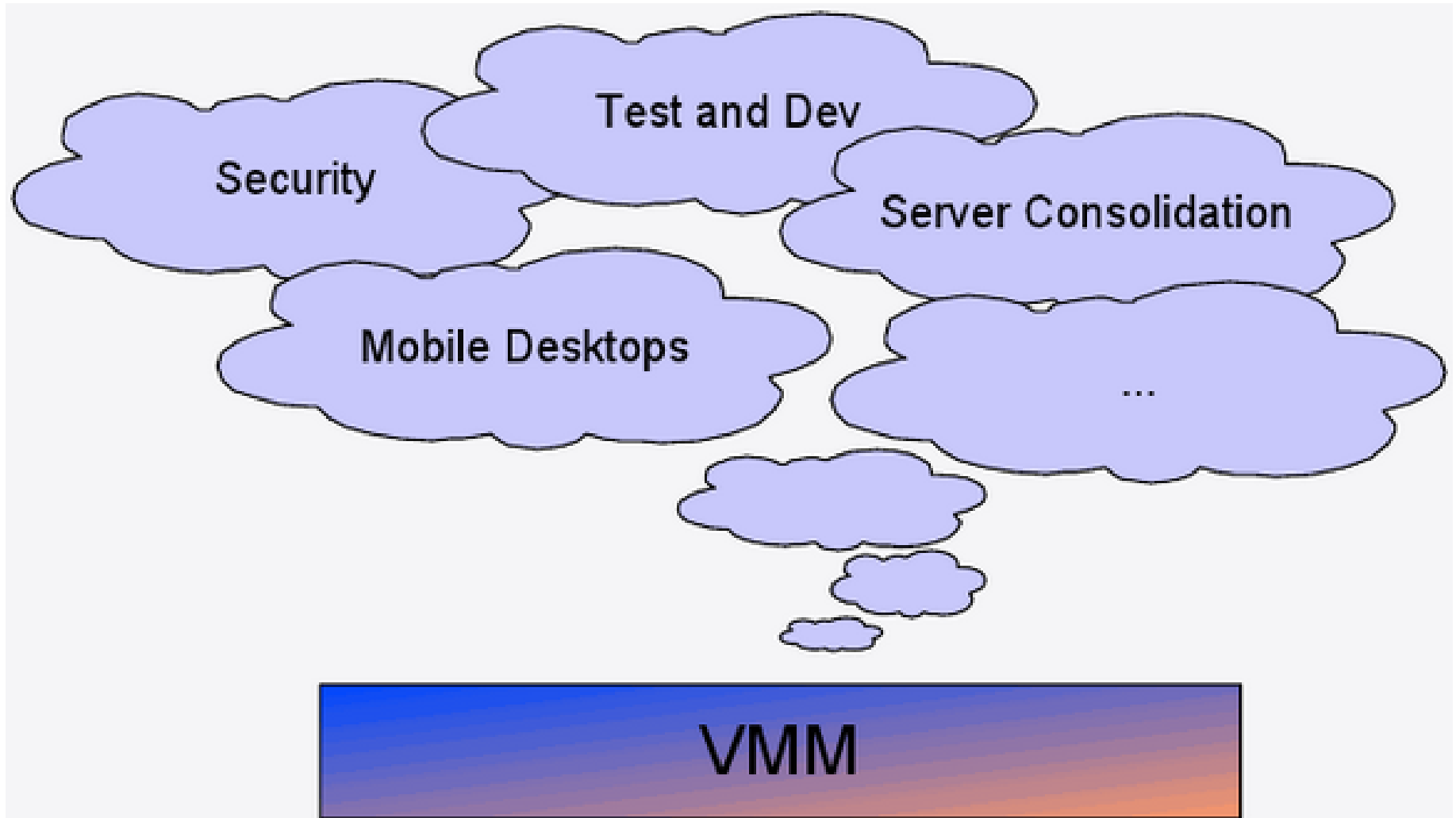
**Software Virtualization**

**Hardware Virtualization**

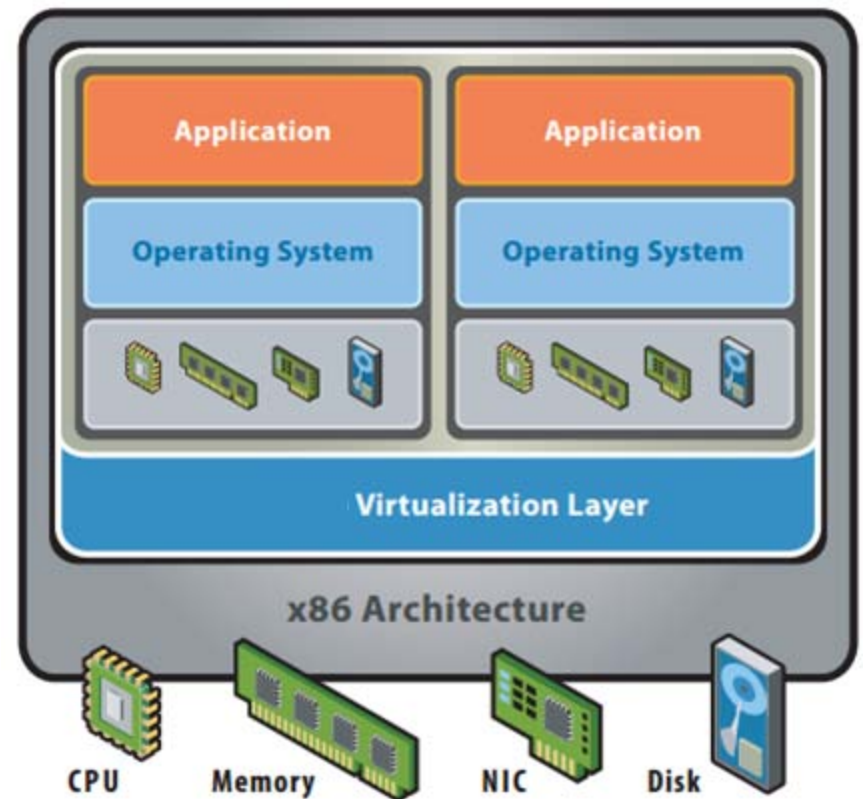
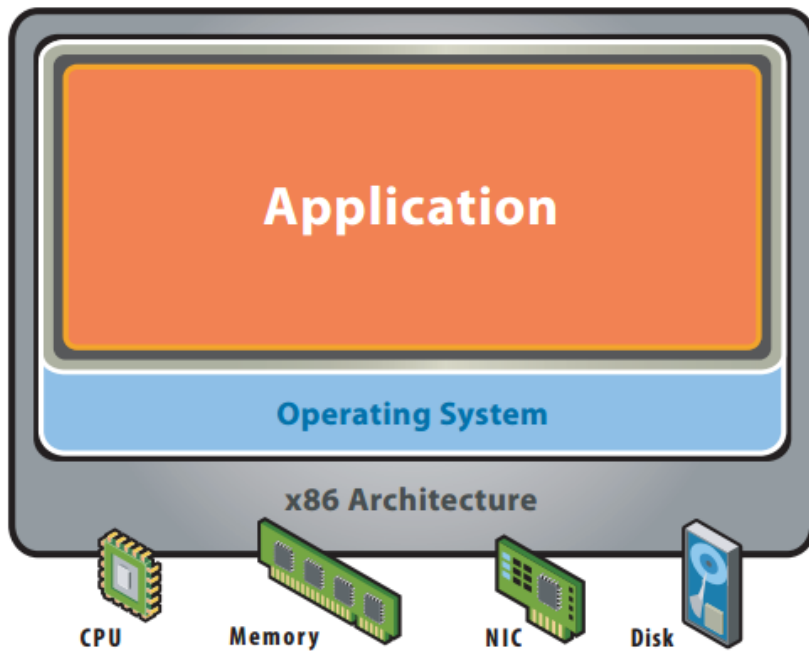
**Evaluation**

**Conclusion**

# Virtualization is Everywhere



# Virtualization in a Nutshell





# Roadmap

**Overview**

**Classical Virtualization**

**Software Virtualization**

**Hardware Virtualization**

**Evaluation**

**Conclusion**



# Popek & Goldberg Theorem

*An effective VMM may be constructed if the set of sensitive instructions for that computer is a subset of privileged instructions.*

**Fidelity:** Behave identically in guest OS and host OS

**Safety:** VMM manages all hardware resources

**Performance:** No VMM intervention for dominant number of instructions

***trap-and-emulate virtualization***

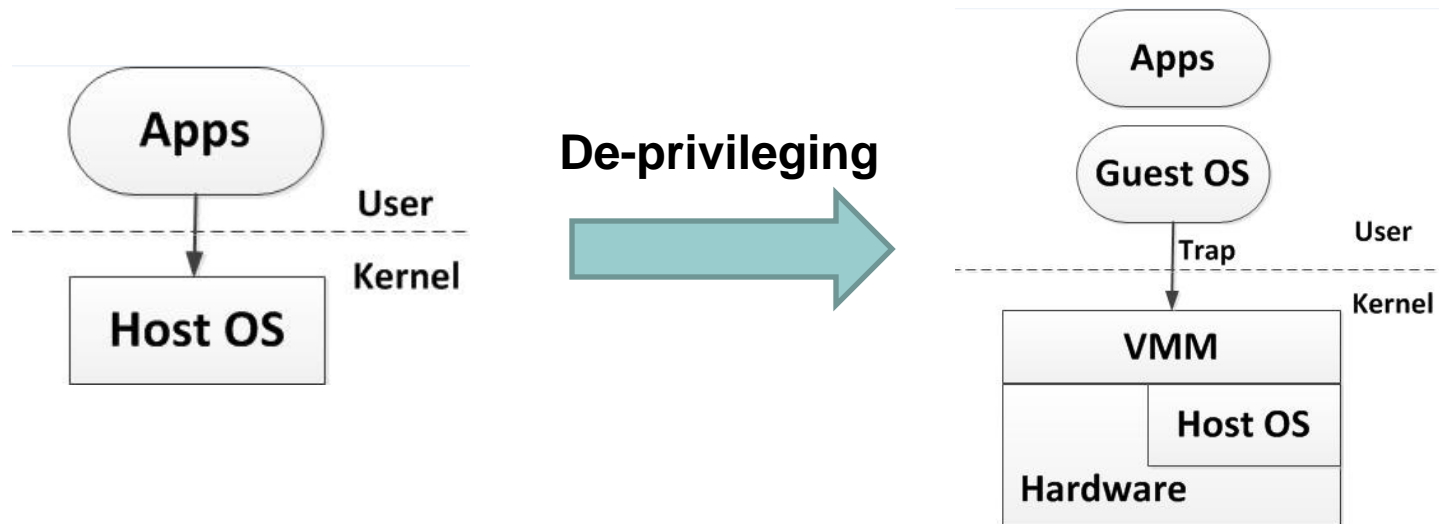
# De-privileging

**Privileged:** Trap-in-user-mode; No-trap-in-kernel-mode

**Control sensitive:** Change the config of resources

**Behavior sensitive:** Depend on the config of resources

$$\{\text{Sensitive Instructions}\} \subset \{\text{Privileged Instructions}\}$$





# Shadow Structures

VMM provides an **execution environment** for guest OS and maintains **shadow structures** for privileged data.

**On-CPU privileged state:** A set of registers

- Maintain an image of registers
- Emulate as operations trap

**Off-CPU privileged data:** page tables, MMIO devices

- Access without trapping instructions (coherency lost)
- Memory traces





# Memory Traces

Use hardware page protection (**tracing**) to trap accesses to in-memory primary structures.

- Guest PTEs with shadow PTEs are write-protected
- MMIO devices must be read/write protected

- 1) **Decode** the faulting guest instruction;
- 2) **Emulate** its effect in the primary structure;
- 3) **Propagate** the change to the shadow structure.

***Shadow Page Tables***



# Refinements

**Exploit VMM/Guest OS interface:** Make guest OS provide higher-level information to VMM

- Improved performance
- Violate Fidelity

**Exploit hardware/VMM interface:** Make hardware understand VMM

- Interpretive execution: Encode guest privileged state in hardware-defined format and execute in interpretive execution mode
- Reduced the frequency of traps



# Roadmap

**Overview**

**Classical Virtualization**

**Software Virtualization**

**Hardware Virtualization**

**Evaluation**

**Conclusion**



# x86 obstacles

**Visibility of privileged state:** Guest OS can observe that it's de-privileged by reading CPL of %cs.

**Lack of traps for privileged instructions:** Violate Popek-Goldberg's Theorem

- popf may change ZF and IF in privileged code; In de-privileged code, hardware simply suppresses attempts to modify IF without generating traps.

***Binary Translation***



# Simple Binary Translation

**Interpreter** separates virtual state (VCPU) from physical state (CPU) through maintaining the complete state of the machine (registers / memory)

- **Overcomes semantic obstacles:** prevent leakage of privileged state from physical CPU into guest; emulate non-trapping instructions correctly
- **Performance bottleneck:** Fetch-decode-execute cycle of the interpreter may generate hundreds of physical instructions per guest instruction.



# Interpreter Example

```
while (!halt && !interrupt) {
    inst = code[PC];
    opcode = extract(inst,31,6);
    switch(opcode) {
        case LoadWordAndZero: LoadWordAndZero(inst);
        case ALU: ALU(inst);
        case Branch: Branch(inst);
        . . .}
}
```

*Instruction function list*

```
LoadWordAndZero(inst){
    RT = extract(inst,25,5);
    RA = extract(inst,20,5);
    displacement = extract(inst,15,16);
    if (RA == 0) source = 0;
    else source = regs[RA];
    address = source + displacement;
    regs[RT] = (data[address]<< 32)>> 32;
    PC = PC + 4;
}
```

```
ALU(inst){
    RT = extract(inst,25,5);
    RA = extract(inst,20,5);
    RB = extract(inst, 15,5);
    source1 = regs[RA];
    source2 = regs[RB];
    extended_opcode = extract(inst,10,10);
    switch(extended_opcode) {
        case Add: Add(inst);
        case AddCarrying: AddCarrying(inst);
        case AddExtended: AddExtended(inst);
        . . .}
    PC = PC + 4;
}
```



# Dynamic Binary Translation

**Binary:** Input is x86 binary code

**Dynamic:** Translation happens at runtime

**Subsetting:** Output is a safe subset (mostly user-mode instructions) of the full x86 instructions set

**On demand:** Code is translated only when it is about to execute

**System level:** Make no assumptions about the guest code.

**Adaptive:** Translated code is adjusted in response to guest behavior changes to improve overall efficiency

# An example

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

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






# Translation Unit

**Translation unit:** 12 instructions or a terminating instruction

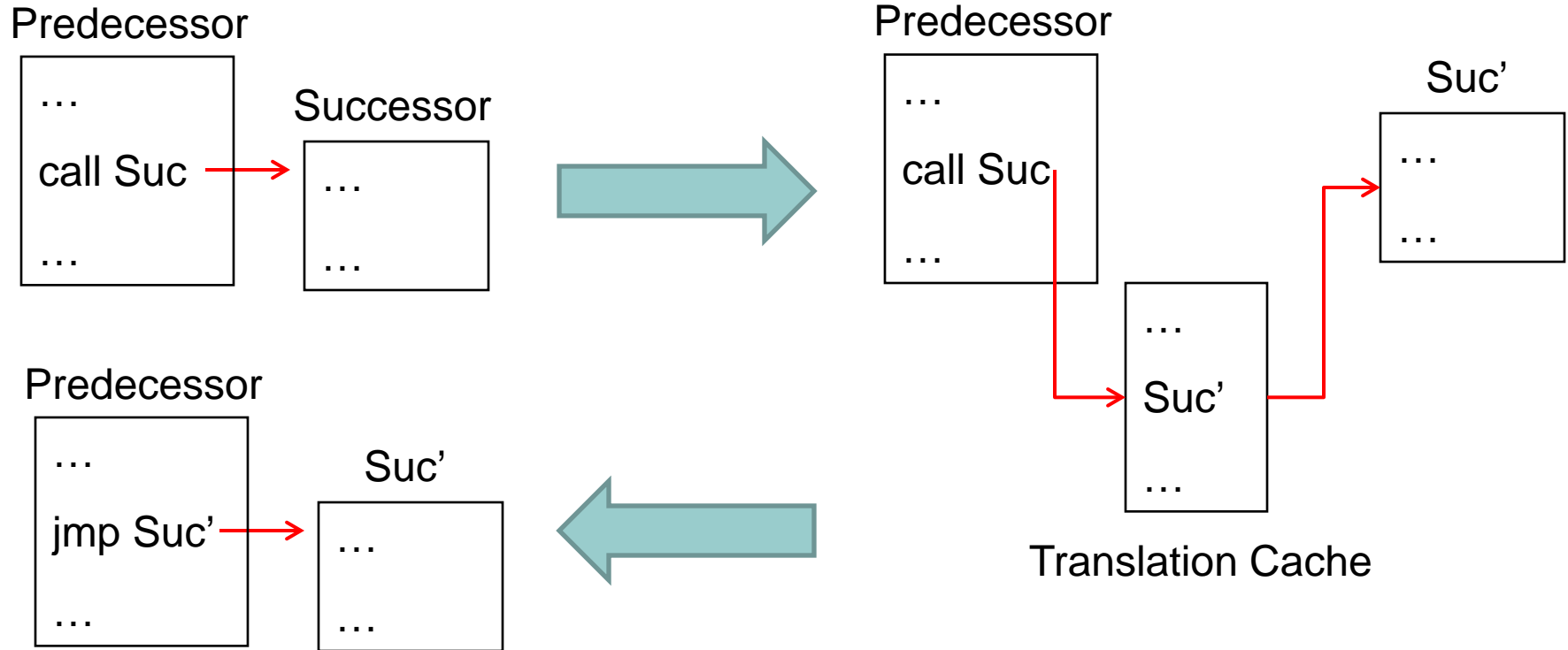
**TU → Compiled code fragment (CCF)**

<code>isPrime:  mov %ecx, %edi</code>		<code>isPrime':  mov %ecx, %edi    ; IDENT</code>
<code>          mov %esi, \$2</code>		<code>          mov %esi, \$2</code>
<code>          cmp %esi, %ecx</code>		<code>          cmp %esi, %ecx</code>
<code>          jge prime</code>		<code>          jge [takenAddr] ; JCC</code>
		<code>          jmp [fallthrAddr]</code>

**Translator-invoking Continuations:** translation does not preserve original code layout

**Chaining optimization:** allow one CCF to jump directly to another OR fall through directly

# Chaining



**No need to locate the generated block for Successor**

**Indirect jump?** jmp/call %eax, jmp/call \*0xdeadbeef

# TU to CCF

## TUs

```
isPrime:  mov    %ecx, %edi
          mov    %esi, $2
          cmp    %esi, %ecx
          jge    prime
nexti:    mov    %eax, %ecx
          cdq
          idiv   %esi
          test   %edx, %edx
          jz     notPrime
          inc    %esi
          cmp    %esi, %ecx
          jl     nexti
prime:    mov    %eax, $1
          ret
notPrime: xor    %eax, %eax
          ret
```

## CCFs

```
isPrime': *mov    %ecx, %edi    ; IDENT
          mov    %esi, $2
          cmp    %esi, %ecx
          jge    [takenAddr]  ; JCC
          ; fall-thru into next CCF
nexti':   *mov    %eax, %ecx    ; IDENT
          cdq
          idiv   %esi
          test   %edx, %edx
          jz     notPrime'     ; JCC
          ; fall-thru into next CCF
          *inc    %esi         ; IDENT
          cmp    %esi, %ecx
          jl     nexti'        ; JCC
          jmp    [fallthrAddr3]
notPrime': *xor    %eax, %eax    ; IDENT
          pop    %r11           ; RET
          mov    %gs:0xff39eb8(%rip), %rcx ; spill %rcx
          movzx  %ecx, %r11b
          jmp    %gs:0xfc7dde0(8*%rcx)
```



# Memory Access & Virtual Registers

**Segmentation (%gs):** Efficient and safe memory access in guest OS

- VMM mapped in high part of the guest's address space
- All segment registers but %gs hold truncated segments
- Translator inserts %gs prefix to gain access to VMM
- Non-IDENT translation if guest inst uses %gs prefix

```
notPrime': *xor    %eax, %eax    ; IDENT
            pop     %r11         ; RET
            mov     %gs:0xff39eb8(%rip), %rcx ; spill %rcx
            movzx   %ecx, %r11b
            jmp     %gs:0xfc7dde0(8*%rcx)
```



# Non-IDENT Translations

**Most instructions can be translated IDENT, except:**

- **PC-relative address:** small code expansion and slowdown
- **Direct control flow:** insignificant slowdown
- **Indirect control flow:** dynamic lookup
- **Privileged instructions:** it depends
  - May be faster than native: 60-cycle cli v.s. vcpu.flags.IF:=0
  - Or measurable overhead due to the callout and the emulation



# Adaptive Binary Translation

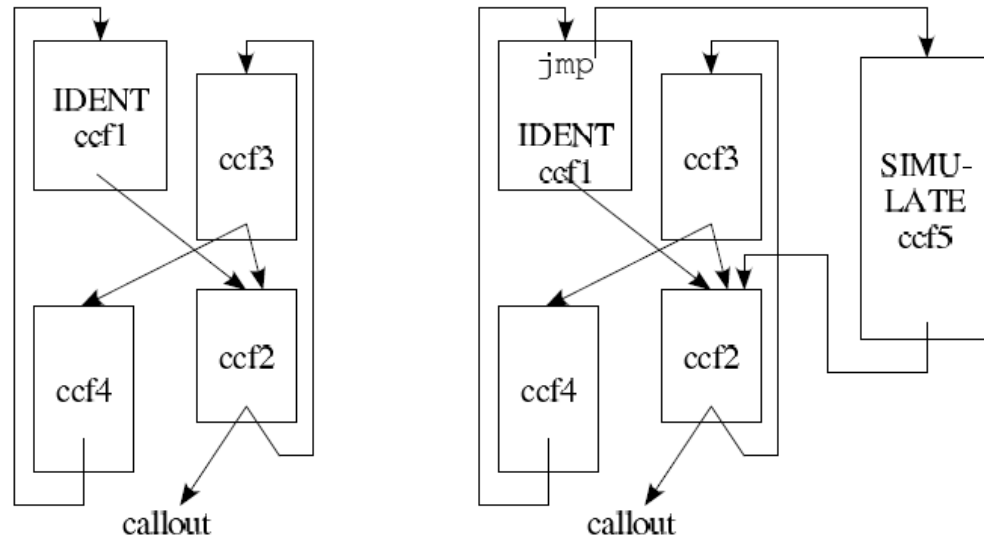
BT eliminates traps from *privileged instructions* and outperforms classical VMMs

rdtsc on Pentium 4 CPU	
Trap-and-emulate	2030 cycles
Callout-and-emulate	1254 cycles
In-TC emulation	216 cycles

**Innocent until proven guilty** for non-privileged instructions accessing sensitive data: 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

# Adaptive Binary Translation



Adaption takes **constant time** due to a forwarding jump

After adaption, we avoid taking a trap in ccf1 and instead execute a faster callout in ccf5

Remove forwarding jump from ccf1 if the offending instruction becomes innocent again



# Roadmap

**Overview**

**Classical Virtualization**

**Software Virtualization**

**Hardware Virtualization**

**Evaluation**

**Conclusion**





# Intel VT-x

Intel VT-x defines processor-level support for VMMs on x86, in the form of **Virtual-Machine eXtensions** (VMX) operation

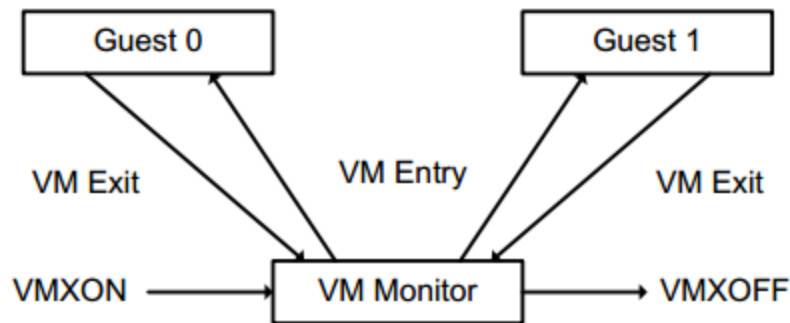
- VMX **root** operation for VMM
- VMX **non-root** operation for guest software
- VMX **transitions**: root (VM exits) <-> non-root (VM entries)

VMX enables guest software to run at the privilege level for which it was originally designed, even CPL 0.

# Virtual-Machine Control Structure

*VMX non-root operation* and *VMX transitions* are controlled by a data structure **Virtual-Machine Control Structure (VMCS)**

- In-memory data structure pointed by **VMCS Pointer**
- VMX exit information, CPU state and control data
- Each VMCS for each virtual processor





# Example: Process Creation

Guest OS creates a process using `fork()`

- `fork()` invoked: CPL transition happens w/o VMM intervention
- COW in guest OS for both parent and child address space
- Guest OS schedules the child process: exit and VMM constructs a new shadow page table
- Hidden page fault exit: the child process touches memory that is not yet mapped in shadow page table
- True page fault exit: paging protection constraints



# Performance

**Reducing the frequency of exits is the most important optimization for classical VMMs**

**Privileged instructions affect state within the virtual CPU as represented within the VMCS rather than unconditionally trapping**



# Roadmap

**Overview**

**Classical Virtualization**

**Software Virtualization**

**Hardware Virtualization**

**Evaluation**

**Conclusion**



# Qualitative Comparison

**BT wins in areas:**

- **Trap elimination**
- **Emulation speed**
- **Callout avoidance**

**Hardware VMM wins in areas:**

- **Code density**
- **Precise exceptions**
- **System calls**



# Quantitative Comparison

**Vmware Player 1.0.1:** software and hardware-assisted VMMs

**HP xw4300 workstation:** VT-enabled 3.8 GHz Intel P 4 672

## Guest OS

- RedHat Enterprise Linux 3
- Windows 2003 Enterprise x64 Edition

## Test cases:

- **Macrobenchmarks:** SPECint 2000, SPECjbb 2005, Apache, compile, 2D Graphics, and PassMark
- **Microbenchmarks:** Forkwait
- **Nanobenchmarks:** syscall, in/out, callret, pgfault, ...

# MacroBenchmarks

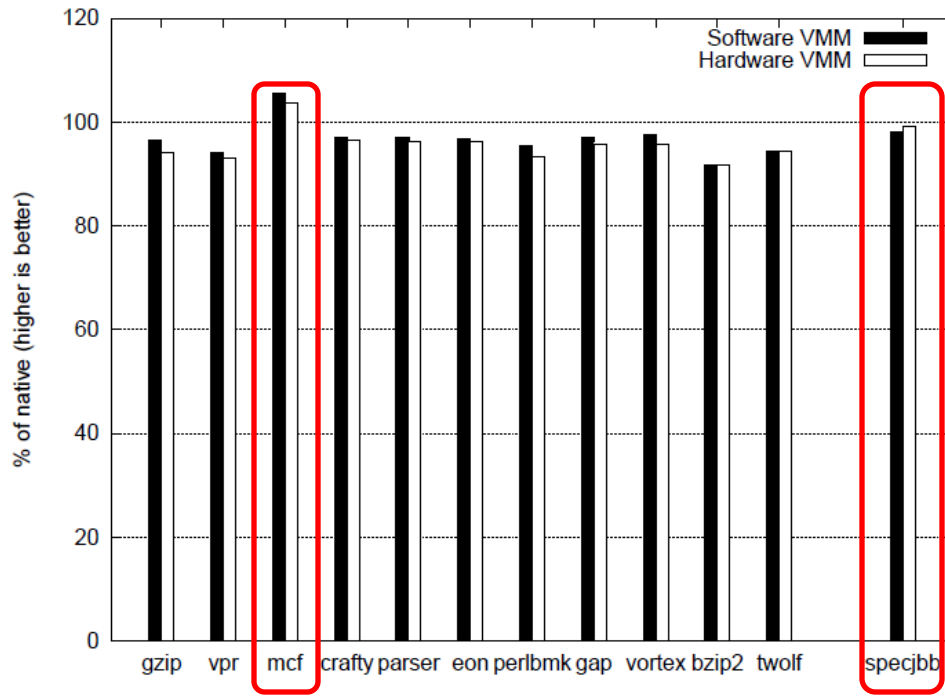


Figure 2. SPECint 2000 and SPECjbb 2005.

- Both VMMs run to score close to native
- mcf runs faster than native on both VMMs
- SPECjbb performs even closer to native



# MacroBenchmarks

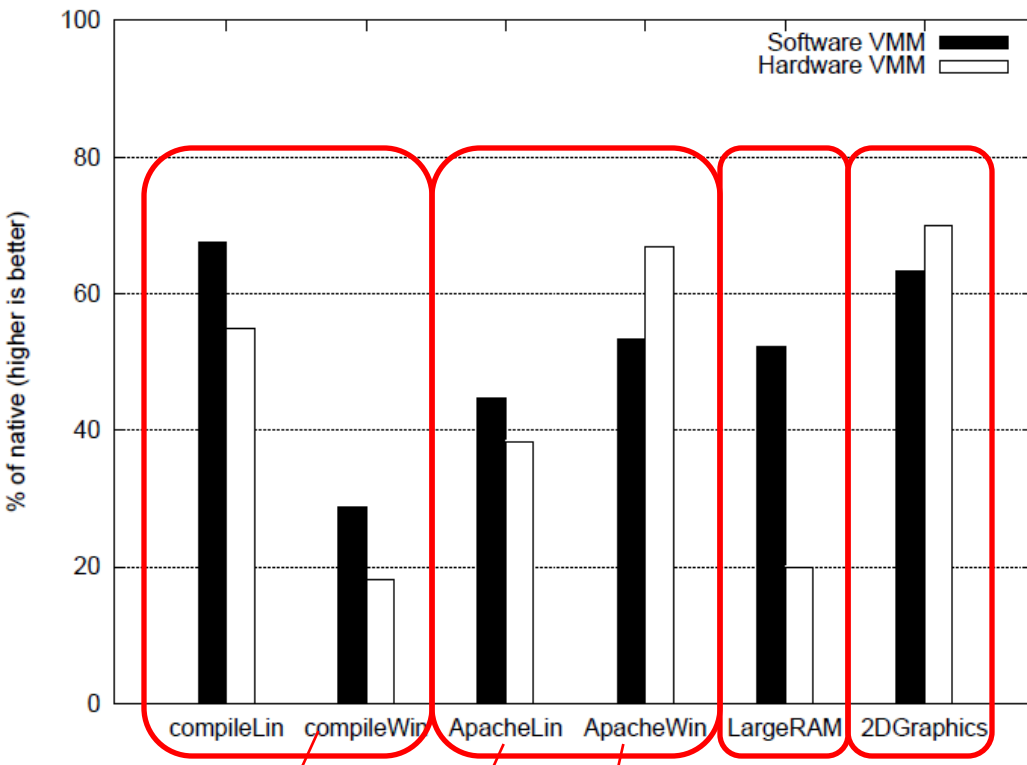


Figure 3. Macrobenchmarks.

Cygwin makes  
it slow

Process  
based

Thread  
based

**Apache:** All four tests  
compare poorly to native

**Compile/PassMark:**  
software VMM outperforms  
hardware VMM

**2DGraphics:** hardware  
VMM outperforms software  
VMM



# MicroBenchmarks

```
int main(int argc, char *argv[]) {  
    for (int i = 0; i < 40000; i++) {  
        int pid = fork();  
        if (pid < 0) return -1;  
        if (pid == 0) return 0;  
        waitpid(pid);  
    }  
    return 0;  
}
```

- **system calls**
- **context switching**
- **creation of address spaces**
- **modification of traced page table entries**
- **injection of page faults**

forkwait	
Native	6.0s
Software VMM	36.9s
Hardware VMM	106.4s

# NanoBenchmarks

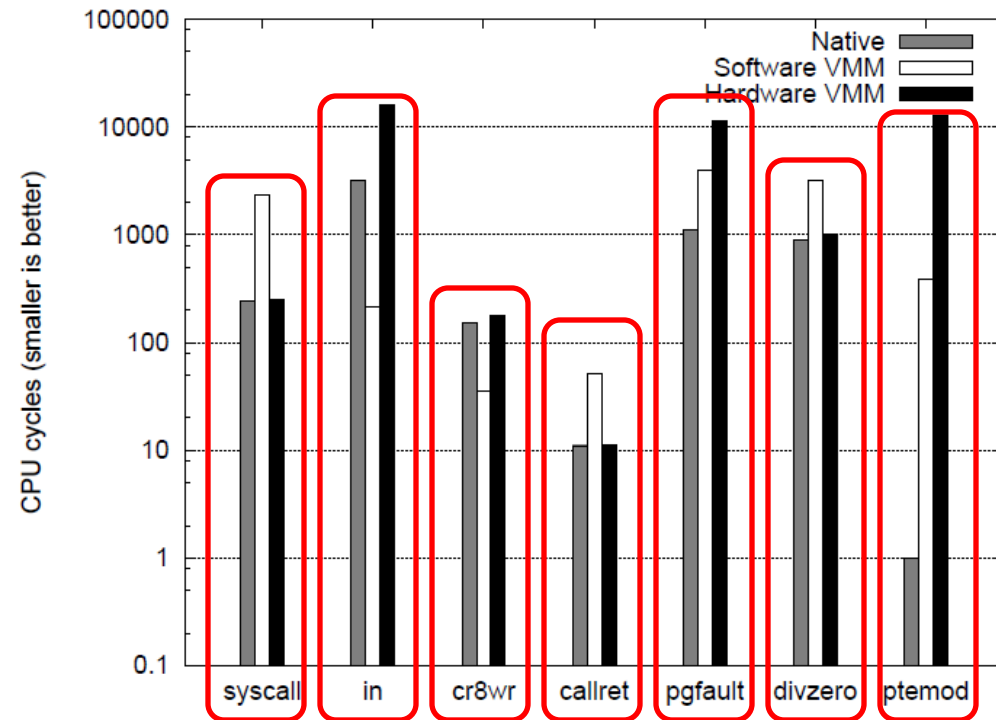


Figure 4. Virtualization nanobenchmarks.

- **syscall, call/ret, divzero:**  
Native = Hardware > Software
- **in, cr8wr:**  
Software > Native > Hardware
- **pgfault, ptemod:**  
Native > Software > Hardware

# Future Opportunities

**Hardware overheads will shrink over time**

**A more potent approach is to eliminate exits entirely**

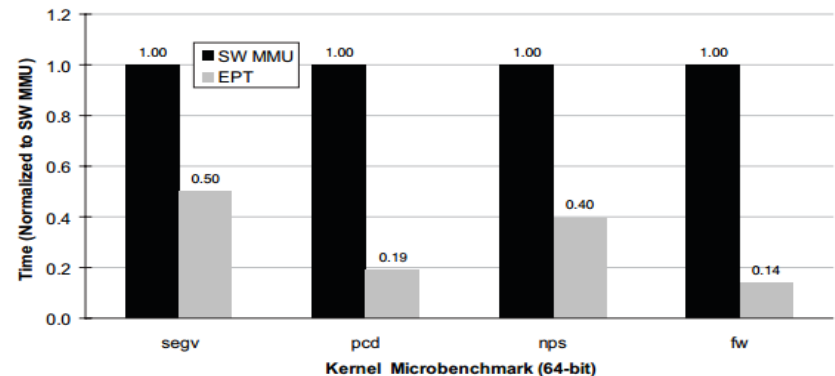
**A hybrid VMM**

**Hardware MMU support**

- Intel's Extended Page Tables (EPT)
- AMD's Nested Paging Tables

	3.8GHz P4 672	2.66GHz Core 2 Duo
VM entry	2409	937
Page fault VM exit	1931	1186
VMCB read	178	52
VMCB write	171	44

**Table 1.** Micro-architectural improvements (cycles).





# Roadmap

**Overview**

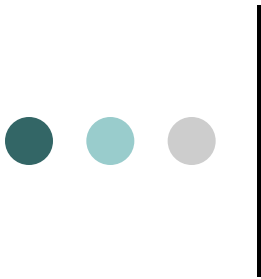
**Classical Virtualization**

**Software Virtualization**

**Hardware Virtualization**

**Evaluation**

**Conclusion**



# Conclusion

- ✓ Both VMMs perform well on compute-bound workloads
- ✓ Software outperforms hardware for workloads that perform I/O, create processes, or switch contexts rapidly
- ✓ Hardware outperforms for workloads with system calls
- ✓ While hardware VMM simplifies VMM design, it rarely improves performance
- ✓ MMU is the bottleneck for hardware VMM



# Software v.s. Hardware VMM

*Thank You!*