A Comparison of Software and Hardware Techniques for x86 Virtualization

Keith Adams,Ole Agesen

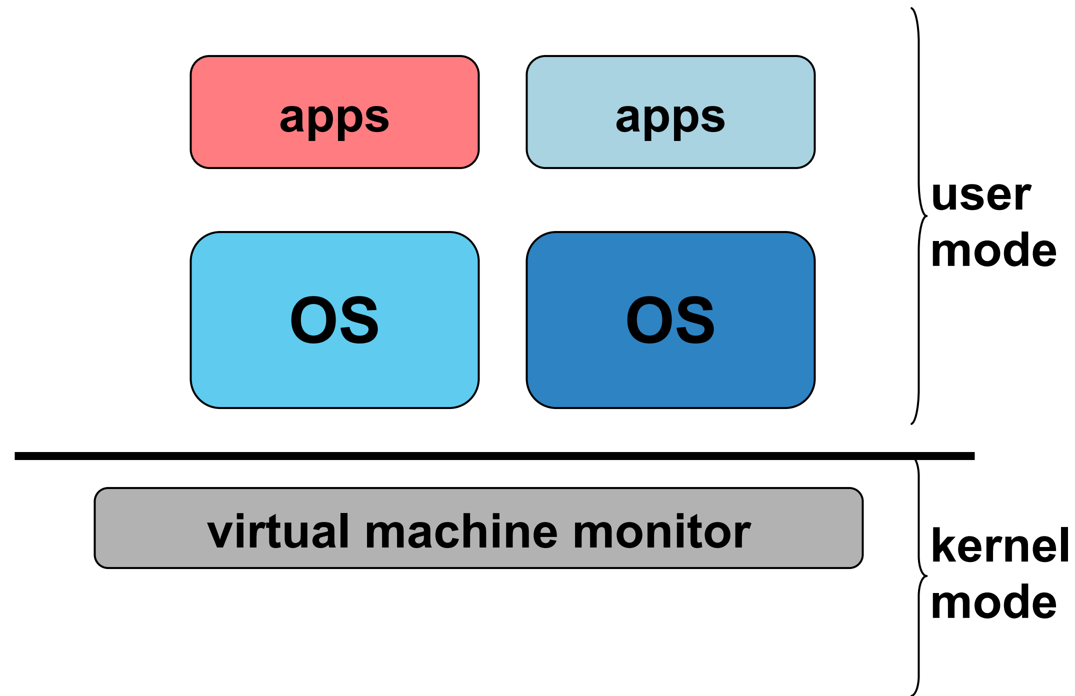
Presenter: Jovan Stojkovic

Classical Virtualization

- Popek and Goldberg’c criteria:
  - 1. Fidelity
  - 2. Performance
  - 3. Safety
- Trap-and-emulate technique
- Classically virtualizable architecture
- The most important ideas from classical VMM implementations
  - De-privileging
  - Primary and shadow structures
  - Memory traces

De-privileging

- A classical VMM executes guest operating systems directly, but at a reduced privilege level.



Shadow and Primary Structures

- Privileged state of each guest differs from that of the underlying hardware
- Basic function of VMM is to meet guest’s expectation
- To accomplish this VMM derives shadow structures from guest-level primary structures
  - Primary structures reflect the state of guest
  - VMM-level shadow structures are copies of guest’s primary structures
- These structures are kept coherent using – memory tracing

Memory Tracing

- On-CPU privileged state – handled trivially
  - Includes - Page table pointer register, processor status register etc
  - Guest access to these registers coincide with trapping instructions
  - On trap VMM refers to the corresponding shadow of the guest register structure in the instruction emulation
- Off-CPU privileged data
  - Guest access to these do not coincide with trapping instructions
  - Example : Guest PTEs are considered privileged data – dependencies on this are not accompanied by traps
  - They can be modified by any store in guest instruction stream
  - VMM cannot maintain coherency of shadow structures
- VMMs use hardware page protection mechanisms to trap access to in memory primary structures – memory tracing

Refinements to classical virtualization

- Traps are expensive (~3000 cycles)
- Many traps unavoidable
  - E.g., page faults
- Important enhancements
  - “Paravirtualization” to reduce traps (e.g., Xen)
  - Hardware VM modes (e.g., IBM s370)

Can x86 trap-and-emulate?

- No
  - Visibility of privileged state - current privilege level (CPL) is stored in the low two bits of %cs
  - Lack of traps when privileged instructions run at user-level
- Classic Example: *popf* instruction
  - Same instruction behaves differently depending on execution mode
  - User Mode: changes ALU flags
  - Kernel Mode: changes ALU **and** system flags
  - **Does not generate a trap in user mode**

Software VMM

- The guest executes on an interpreter instead of directly on a physical CPU
  - The interpreter separates virtual state (the VCPU) from physical state (the CPU)
  - Performance issues
- Binary translation can combine the semantic precision of interpretation with high performance

Properties	
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	The translator makes no assumptions about the guest code
Subsetting	Input is full x86 instruction set, output safe subset
Adaptive	Translated code is adjusted to improve overall efficiency

Simple Binary Translation

- Translator classifies the bytes as prefixes, opcodes or operands to produce intermediate representation (IR) objects and accumulates them into a translation unit (TU)
- Each IR object represents one guest instruction

```
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
```

Simple Binary Translation

- Most instructions can be translated IDENT
- Exceptions:
  - PC-relative addressing
  - Direct control flow
  - Indirect control flow (jmp, call, ret)
  - Privileged instructions
- Switching guest execution between BT mode and direct execution as the guest switches between kernel- and user-mode → limit BT overheads to kernel code

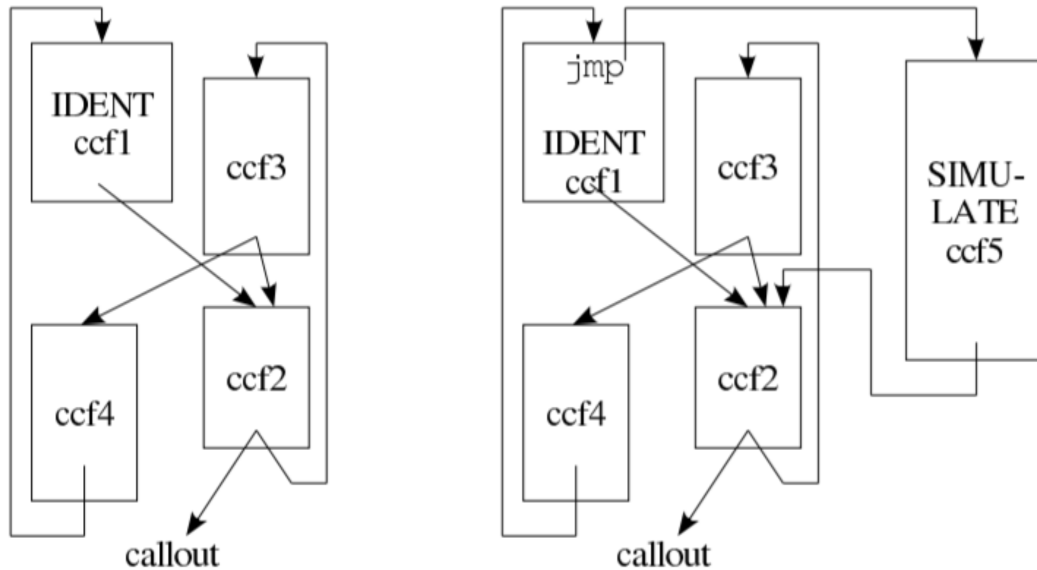
Simple Binary Translation - Example

```
isPrime:   mov %ecx, %edi
           mov %esi, $2
           cmp %esi, %ecx
           jge prime
```

```
isPrime':   mov %ecx, %edi     ; IDENT
            mov %esi, $2
            cmp %esi, %ecx
            jge [takenAddr]    ; JCC
            jmp [fallthrAddr]
```

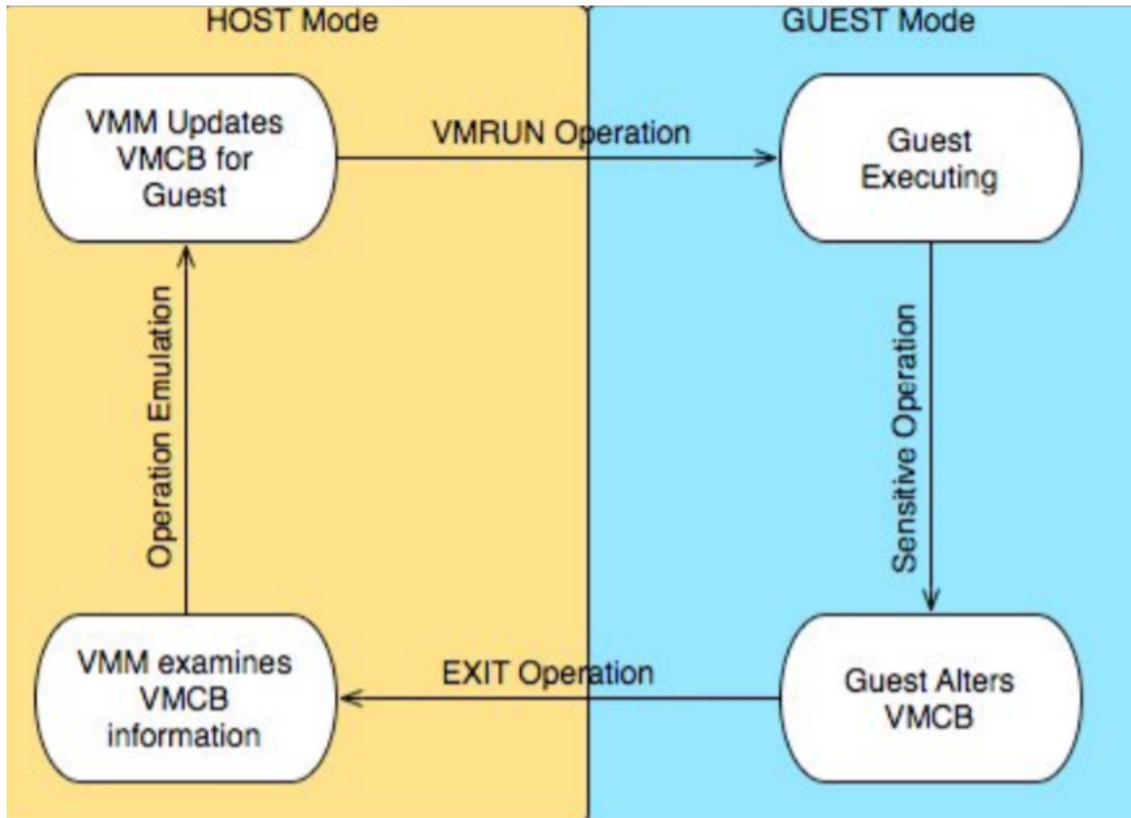
- Each translator invocation consumes one TU and produces one compiled code fragment (CCF)

Adaptive Binary Translations



- Privileged instruction traps – eliminated by simple BT
- Non-privileged instructions (e.g. load, store) accessing sensitive data such as page tables
- Strategy : *innocent until proven guilty*
  - Identify the CCF that traps frequently
  - IDENT translation type is adapted to SIMULATE translation type
  - Patch CCF5 with a jump in CCF1
  - This avoids trap in CCF1

Hardware Virtualization



- x86 architecture extensions
  - virtual machine control block (VMCB)
  - guest vs host mode
  - vmrun and exit instructions
  - diagnostic fields in the VMCB aid the VMM in handling the exit

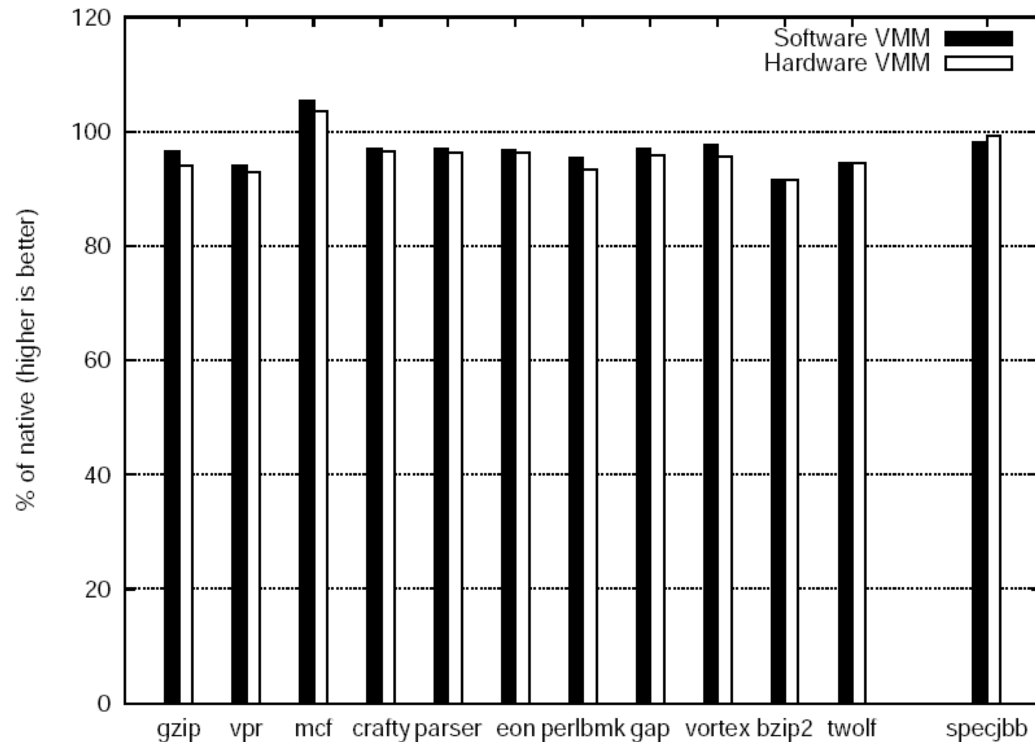
Qualitative comparison

- BT wins:
  - Trap elimination
  - Emulation speed
  - Callout avoidance
- HW wins:
  - Code density
  - Precise exceptions
  - System calls

Experiments

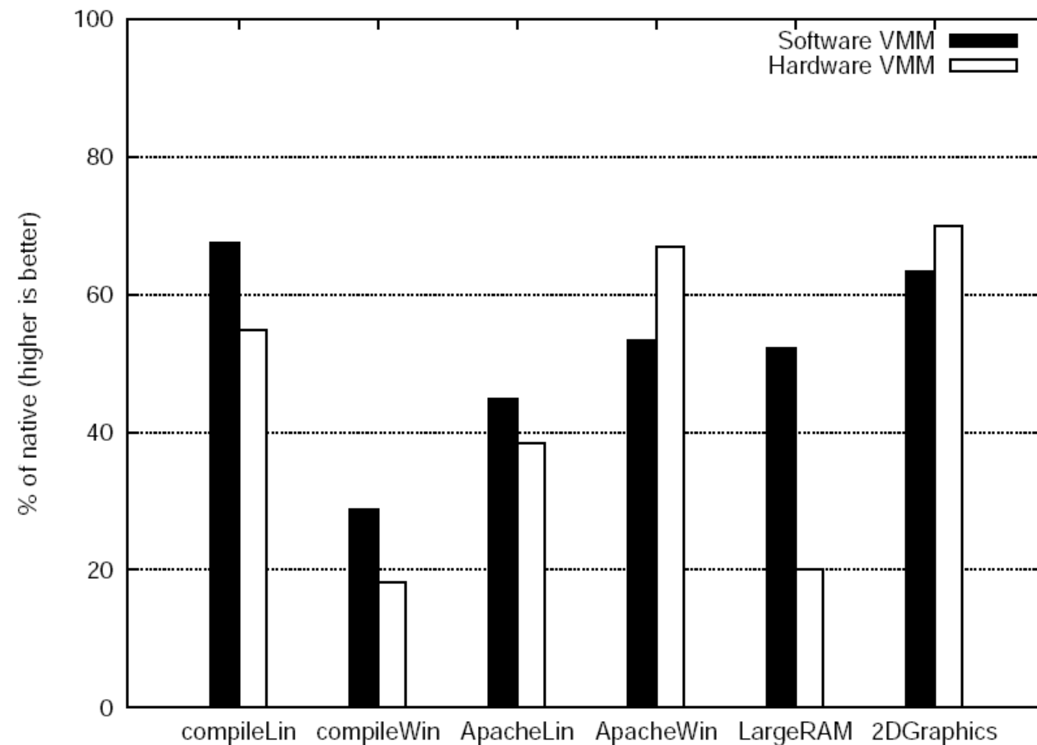
- Software VMM – VMware Player 1.0.1
- Hardware VMM – VMware implemented experimental hardware assisted VMM
- Host – HP workstation, VT-enabled Virtual Machines 18
- Host – HP workstation, VT-enabled
  - 3.8 GHz Intel Pentium
- All experiments are run natively, on software VMM and on Hardware-assisted VMM

Experiments: User-Level Computations



- Benchmarks used:
  - SPECint 2000 benchmark on Red Hat Linux 3
  - SPECjbb 2005 on Windows 2003 • Observations:
- With SPECint benchmarks
  - Near native performance
  - Average slowdown of 4% for software VMM
  - Average slowdown of 5% for hardware VMM
    - Overhead could be due to host background activity, housekeeping kernel threads
- With SPECjbb benchmarks
  - Both very close to native performance – 99% with Hardware VMM and 98% in Software VMM

Experiments: Macrobenchmarks



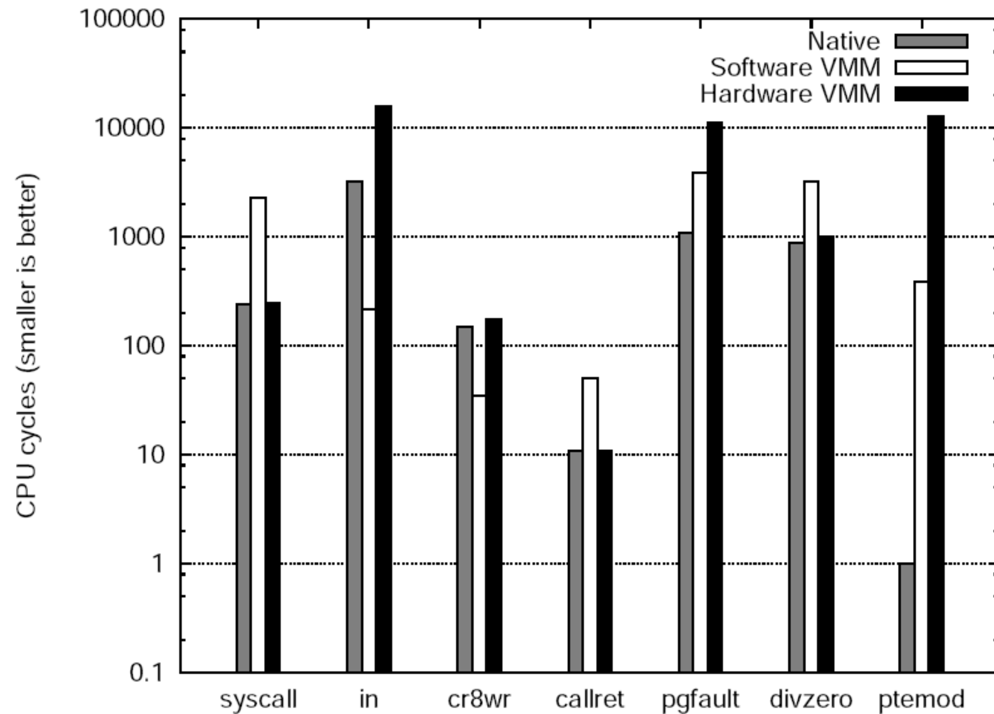
- CompileWin - a larger gap relative to native due to IPC overheads (additional context switches) from the Cygwin UNIX emulation environment
- ApacheLin – process based
- ApacheWin – thread based
- LargeRAM - component exhausts the 1GB of RAM available in both host and guest, leading to paging
- 2DGraphics – system calls

Experiments: Forkwait Test

```
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;
}
```

- Focuses intensely on virtualization-sensitive operations, resulting in low performance relative to native execution
- Test to stress process creation and destruction
  - system calls, context switching, page table modifications, page faults
- Results – to create and destroy 40000 processes
  - Host – 6 seconds
  - Software VMM – 36.9 seconds
  - Hardware VMM – 106.4 seconds

Experiments: Virtualization Nanobenchmarks



- Series of "nanobenchmarks" that each exercise a single virtualization-sensitive operation
- in and cr8wr show that Software VMM can be even faster than Native
- syscall, divzero and callret have similar performance in Hardware VMM and Native
- pgfault and ptemod show where is the biggest overhead of virtualization

Software and Hardware Opportunities

- Many of the difficult cases for the hardware VMM examined surround MMU virtualization
- Faster Microarchitecture implementations
  - Intel Core Duo already much faster than P4
- Hardware VMM algorithms
- Software/Hardware Hybrid VMM
- Hardware MMU
  - Nested paging

Conclusion

- Hardware extensions allow classical virtualization on x86 architecture
- Extensions remove the need for Binary Translation and simplifies VMM design
- Software VMM fares better than Hardware VMM in many cases
  - context switches, page faults, trace faults, I/O
- New MMU algorithms might narrow the gap in performance