A Comparison of Software and Hardware Techniques for x86 virtualization

Keith Adams, Ole Agesen

Presented by Ramya Naidu

Abstract

Until 2006, x86 architecture did not permit classical trap-and-emulate virtualization. Instead VMMs for x86 like VMware used Binary Translation of the guest kernel code.

2006 - Intel and AMD introduced architectural extensions to support classical virtualization on x86.

The paper compares software VMM techniques with the new VMM with hardware support. And discovers that hardware techniques suffer from low performance than software VMM.

Outline

- Classical Virtualization
- Software Virtualization
- Hardware Virtualization
- Qualitative Comparison
- Experiments
- Software and Hardware Opportunities
- Conclusion

Classical Virtualization

- Essential characteristics for a VMM, *established by Popek and Goldberg 's 1974 paper*
  - Fidelity - Software on the VMM executes identically to its execution on hardware
  - Performance – Software performs on VMM as it would on hardware
    - To ensure performance, majority of guest instructions should be executed without VMM intervention
  - Safety – VMM manages all hardware resources
    - VMM should intervene when instructions that interact with hardware are executed *(sensitive instructions)*

Classical VMM

- *Classical VMM* – particular VMM implementation style, uses trap-and-emulate
- *Classically Virtualizable Architecture* – all instructions that read or write privileged state can be made to trap when executed in unprivileged context
- Important ideas from classical VMM implementations
  - ➢ De-Privileging
  - ➢ Maintain Shadow Structures
  - ➢ Use of Memory Traces

Classical VMM*: De-privileging*

- Executes guest operating systems directly but at lesser privilege level, *user-level*
- Instructions that read or write privileged state – trap
- VMM intercepts the trap and emulates the trapping instruction against the virtual machine state

Classical VMM: *Primary and Shadow 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 guests primary structures
- These structures are kept coherent using – *memory tracing*

Classical VMM: *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*

Classical VMM : Memory Tracing example

- Guest PTEs for which shadow PTEs are constructed may be write-protected
- Guest access to these cause traps *(tracing faults)*
- VMM decodes the faulting guest instruction, emulates its effect on primary structure and propagates the change to shadow structures

Refinements to Classical VMM

- Traps are expensive

- Exploit flexibility VMM/guest OS interface
  - Modify guest OS to provide higher-level information to VMM
  - Approach relaxes Popek and Goldberg's *fidility* requirement
  - But gains in performance

- Exploit flexibility in VMM/hardware interface
  - IBMs System 370 architecture

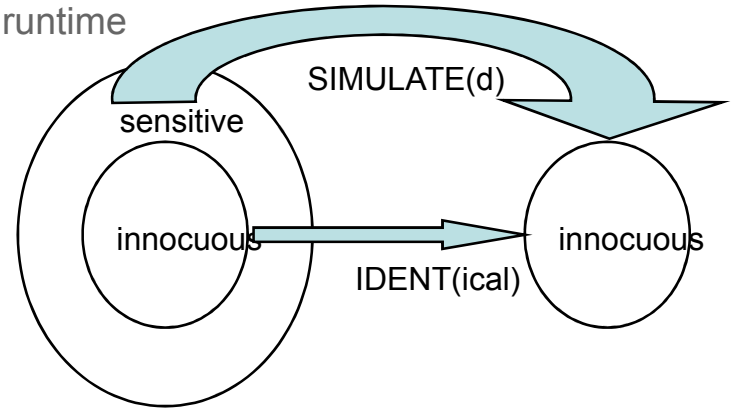
x86 obstacles to classical virtualization

- x86 architecture is not classically virtualizable

- Lack of traps when privilege instructions run at user-level.
  - Example – popf (pop Stack into flags register), loads word from stack into flags register
  - When executed in user-level, *popf* should trap but instead the flag register is overwritten except for the interrupt-enable flag IF bit

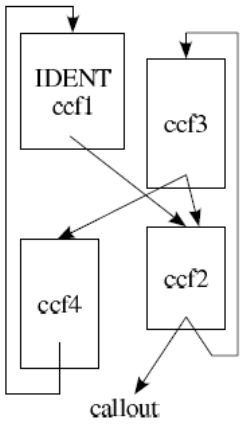
Software VMM: To overcome obstacles on x86

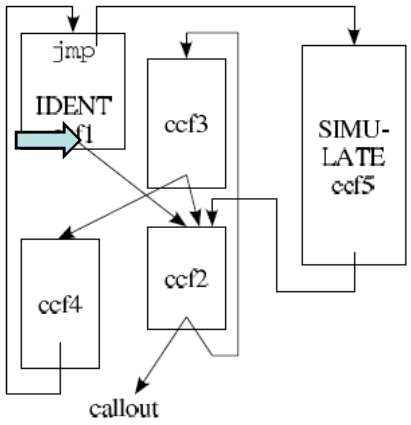
- Execute guest on Interpreter
  - ➢But fetch-decode-execute cycle of interpreter reduces performance
- Binary Translation of the guest
  - VMM can switch between BT mode and direct execution mode
  - Properties
    - ➢On demand – Code is translated only when it is about to execute
    - ➢Dynamic - Translation happens at runtime
    - ➢Subsetting – the translators input is full x86 instruction set, including all the privileged instructions; output is a safe subset of user-mode instructions
      - – Privilege Instructions: in-TC sequences are used
      - – For complex operations like context switch – callout to runtime



Software VMM: Adaptive Binary Translation

- Adaptive Translation – used to reduce more traps
  - ➢Privileged instruction traps – eliminated by simple BT
  - ➢Non-privileged instructions *(eg: 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

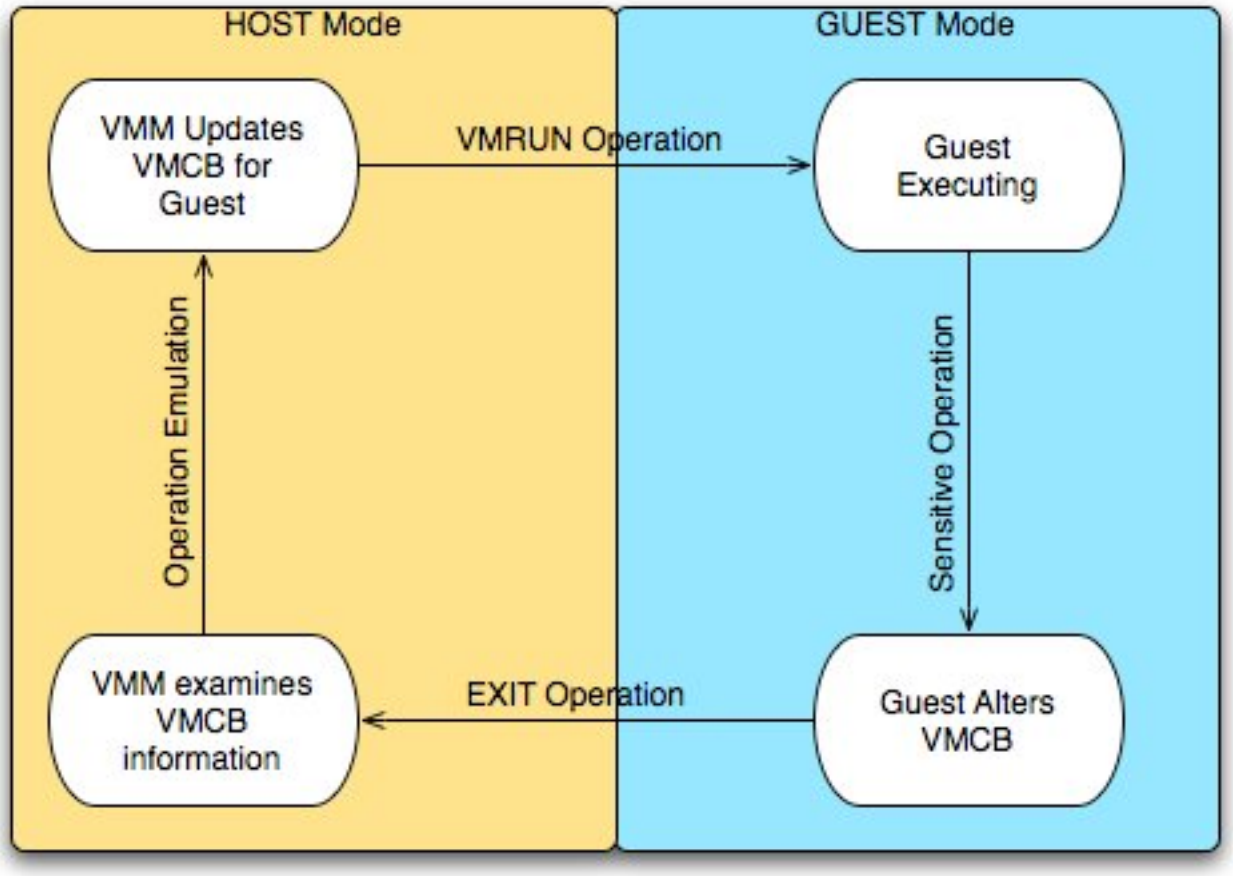




Hardware Virtualization

- Architectural changes to permit classical virtualization on x86
- Virtual Machine Control Block: in-memory data structure
  - contains the state of guest virtual CPU
- New less privileged execution mode – *guest mode*
  - guest mode : supports direct execution of guest code and privilege code
- New Instructions
  - vmrun and vmexit

Hardware Virtualization



[2]

Hardware Virtualization

- Where possible, privileged instructions that affect state within the virtual CPU as represented within the VMCB, rather than unconditionally trapping
  - Example: popf - with x86 extension to support classical virtualization
  - VMCB includes a hardware-maintained shadow of the guest flags register
  - When running in guest mode – instructions operating on flags, operate on the shadow – removing the need for exits

- Hardware provides means of throttling some exit types, however use of traces and hidden page faults directly impact the exit rate

Qualitative Comparison – Hardware and Software VMMs

- Trap elimination – Software fares better
  - Hardware VMM : replaces traps with exits
  - Software VMM : adaptive BT can replace most traps with *faster* callouts
- Emulation Speed – Software faster
  - Hardware VMM : must fetch VMCB and decode trapping instruction before emulating it
  - Software VMM : callouts jump to precoded emulation routine
- Code Density – Hardware wins
  - Since there is no translation
- System Calls – runs without VMM intervention

Experiments

- Software VMM – VMware Player 1.0.1
- Hardware VMM – VMware implemented experimental hardware assisted VMM
- 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

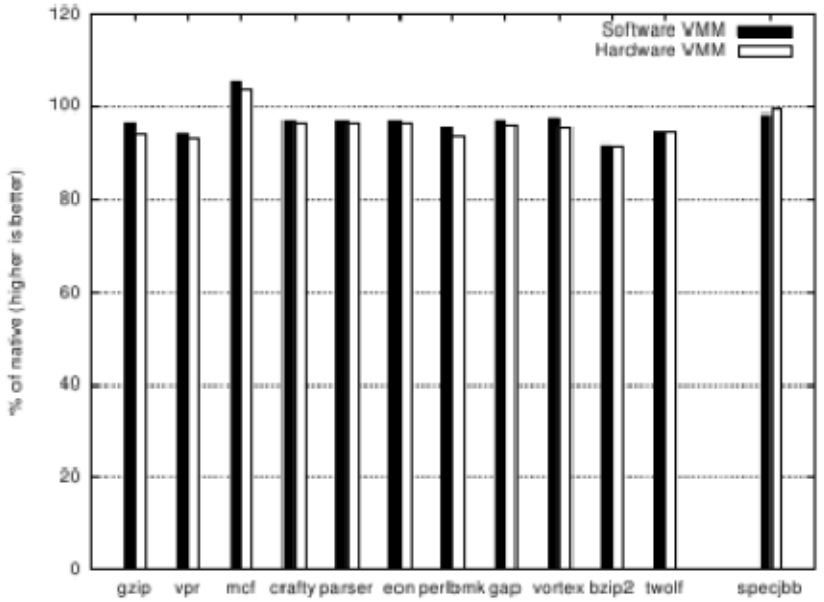


Figure 2. SPECint 2000 and SPECjbb 2005.

Experiments: Server Workload

- Benchmarks:
  - ➢Apache ab benchmarking tool – on Linux installation of Apache http server and on Windows installation
- Observations:
  - ➢ Both VMMs perform poorly
  - ➢VMware Player uses hosted I/O model in which all network packets pass through the host OS's I/O stack
  - ➢Performance on Windows and Linux differ
  - ➢On Windows – Hardware VMM fares better
  - ➢On Linux – Software VMM fares better
- Reason: Apache Configuration
  - ➢Apache defaults to single address space on Windows – and on Linux many address spaces

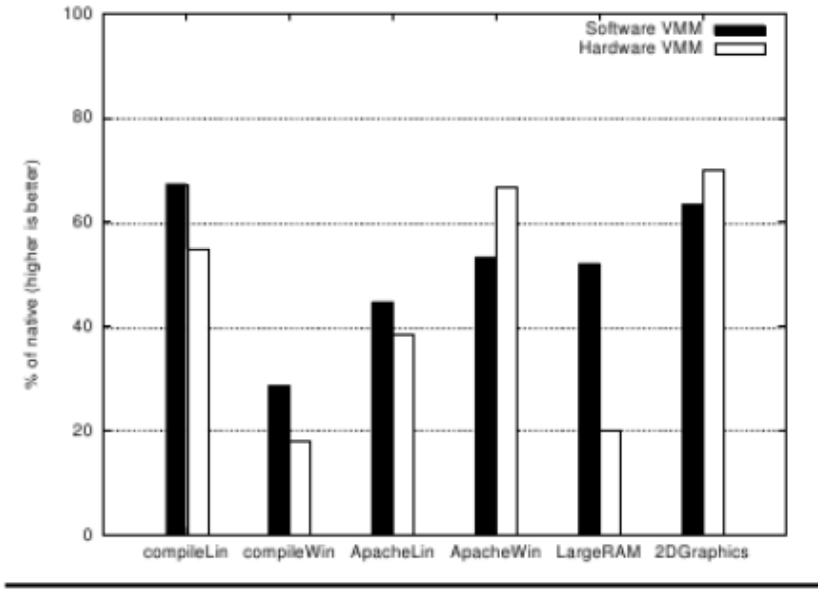


Figure 3. Macrobenchmarks.

Experiment: Desktop-Oriented Workload

- Benchmark
  - ➢PassMark on Windows XP Professional
  - ➢The suite of microbenchmarks test various aspects of workstation performance
- Observations:
  - ➢Both VMMs encounter similar overhead in most cases
  - ➢Paper presents result of only the “Large RAM” and “2DGraphics” tests.

  - Large RAM
    - ➢The component tests paging capabilities
    - ➢ Software VMM performs better than Hardware VMM
  - 2D Graphics
    - ➢Involves system calls
    - ➢Hardware VMM fares better - Handles kernel/user transitions better

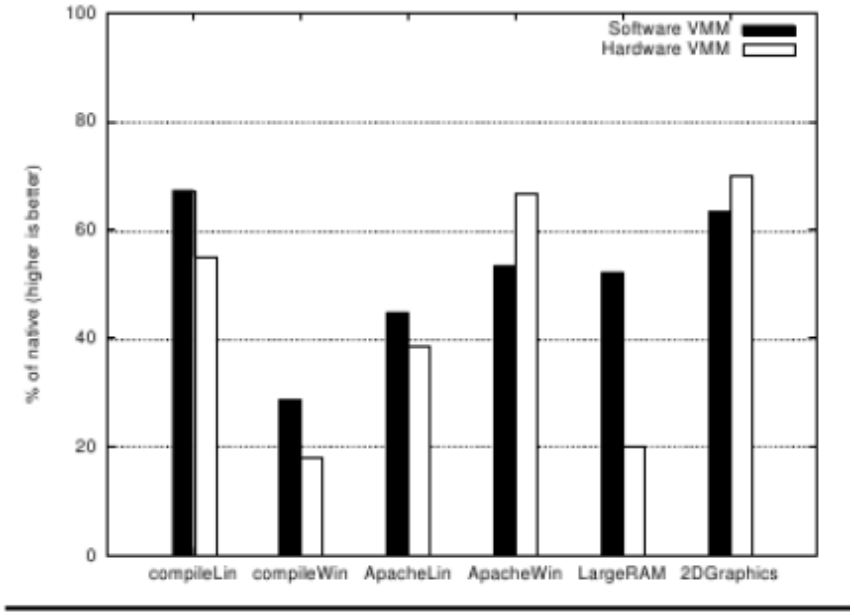


Figure 3. Macrobenchmarks.

Experiments – Less Synthetic Workload

- Compilation times of Linux kernel and Apache (on Cygwin)

- Software VMM beats Hardware VMM
  - ➢ Big compilation jobs -> lots of page faults. Software VMM is better in handling page faults

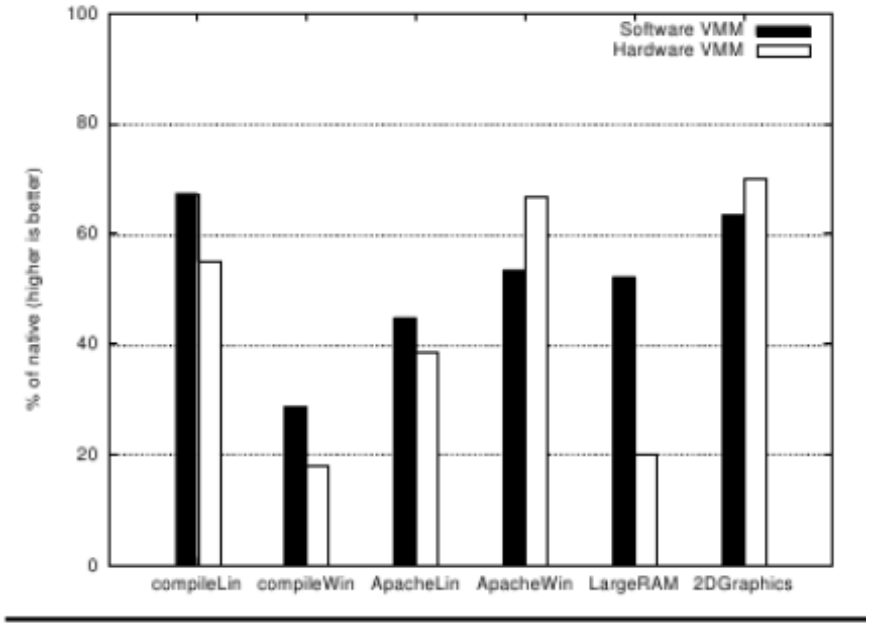


Figure 3. Macrobenchmarks.

Experiments: Forkwait Test

- Test to stress process creation and destruction
  - system calls, context switching, page table modifications, page faults etc
- Results – to create and destroy 40000 processes
  - Host – 0.6 seconds
  - Software VMM – 36.9 seconds
  - Hardware VMM – 106.4 seconds

Experiments: Nanobenchmarks

- Test performance of single virtualization sensitive operation
- Custom guest OS – *FrobOS*
- Syscall -
  - Hardware - No VMM intervention in so near native
  - Software – traps
- in
  - Native – access a off-CPU register
  - Software VMM – translates "in" into a short sequence of instructions that access virtual model of the same
  - Hardware – VMM intervention
- pgfault
  - Both VMMs use software MMU, logically same
  - But path taken different – Software VMM receives control and page faults (in true page fault)
  - In Hardware VMM path taken is longer – exit/run
- ptemod
  - Both use shadowing technique to implement guest paging using traces for coherency
  - PTE writes causes significant overhead compared to native
  - Adaptive BT can reduce overhead but Hardware VMM enters and exits guest mode repeatedly

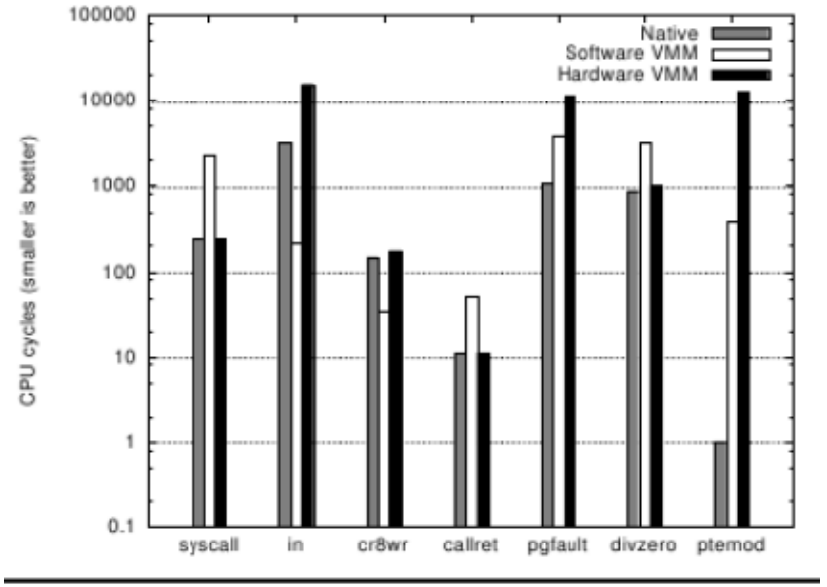


Figure 4. Virtualization nanobenchmarks.

Opportunities

- Microarchitecture
  - ➢ Hardware overheads will shrink over time as implementations mature
  - ➢ Measurements on desktop system using a pre-production version Intel's *core* microarchitecture
- Hardware VMM algorithmic changes
  - ➢ drop trace faults upon guest PTE modification, allowing temporary incoherency with shadow page tables to reduce costs
- Hybrid VMM
- Hardware MMU support
  - ➢ trace faults, context switches and hidden page faults can be handled effictively with hardware assistance in MMU virtualization

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 like context switches, page faults, trace faults, I/O
- New MMU algorithms might narrow the gap in performance

References

[1] " VMware – Hardware Support", http://courses.cs.vt.edu/~cs5204/kafura-fall08/Presentations/VMM-Part2.pdf, retrieved April 8,2009

[2] www.ittc.ku.edu/~niehaus/classes/750-s07/notes/virt-methods-comparison.ppt, retrieved April 8, 2009