A COMPARISON OF SOFTWARE AND HARDWARE TECHNIQUES FOR X86 VIRTUALIZATION

by Keith Adams, Ole Ageson
Presented by Michael Wallner

May 10th, Software Systems Seminar 2007

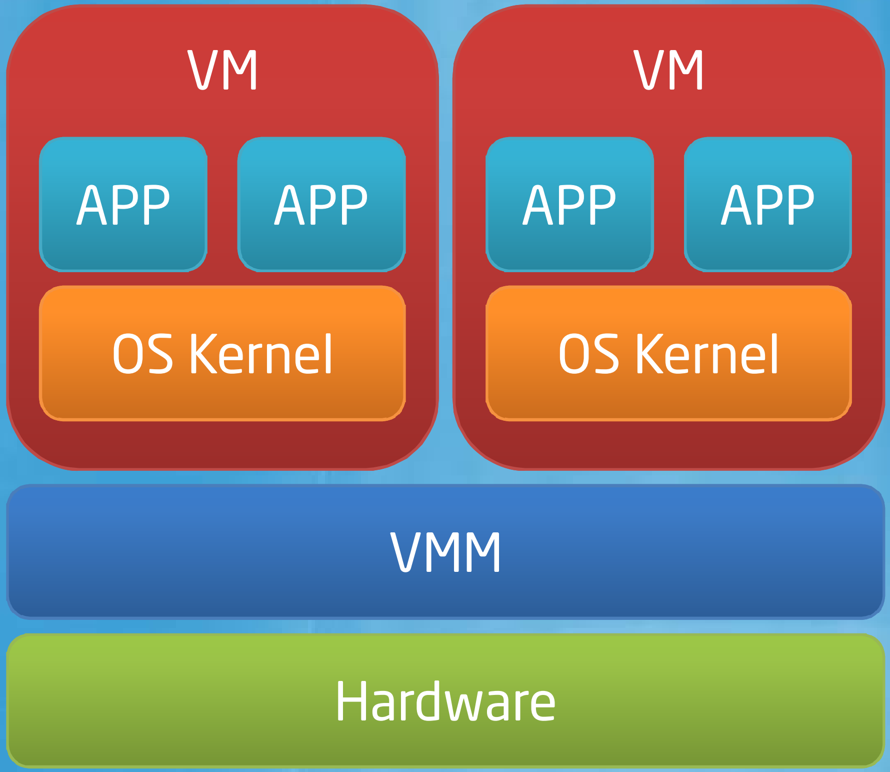
Department of Computer Sciences, University of Salzburg

Content

- Introduction
- Virtualization
  - Classical
  - Software
  - Hardware
- Comparison
- Opportunities
- Conclusion

Introduction and Terminology

- Virtualization
- Virtual Machine
  - Guest
- Virtual Machine Monitor
  - Host
- Motivation
  - Resource utilization
  - Development
  - ...



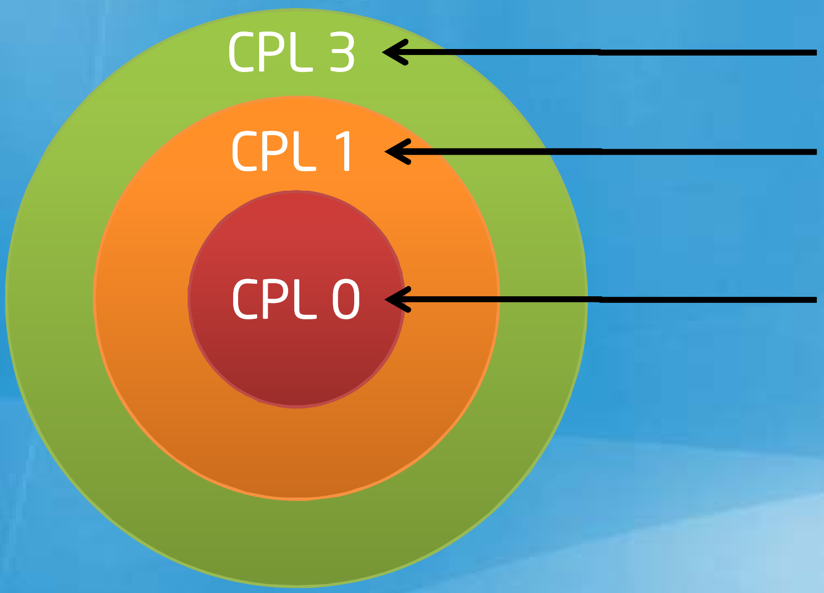
CLASSICAL VIRTUALIZATION

Classical Virtualization

- Three essential characteristics (Popek and Goldberg)
  - Fidelity – runs any software
  - Performance – fairly fast
  - Safety – VMM manages hardware
- Trap-and-Emulate
  - Only real solution until recently

De-Privileging

- Read/Write privileged state Instruction
- Direct Execution but reduced privileged level
- VMM intercepts traps and emulates



CPL 3 Applications

CPL 1 Operating System

CPL 0 Virtual Machine Monitor

Shadow Structures

- Virtual state differs from physical state
- VMM provides basic Execution Environment
- Shadow Structures
- On-CPU privileged state
  - Maintained as Image
- Off-CPU privileged data
  - Resides in Memory

Memory Traces

- Use of hardware page protection mechanisms for coherency of shadow structures
- Protection for memory-mapped devices
- Handling a trace fault:
  - Decode guest instruction
  - Emulate its effect in the primary structure
  - Apply change to the shadow structure

Tracing Example

- Use of Shadow Page Tables to run guest
- Vmware manages SPTs as cache
- True Page Fault
  - Violation of the protection policy
  - Forwarded to guest
- Hidden Page Fault
  - Missing Page in SPT
  - No guest-visible effect

Refinements

- Flexibility in VMM/guest OS Interface
  - Modify guest OS
  - Performance Gains
  - Extended Features
- Flexibilty in VMM/hardware Interface
  - Hardware Execution mode for guest OS
- “Paravirtualization”

SOFTWARE VIRTUALIZATION

x86 Obstacles

- Visibility of privileged state
- Lack of Traps at user-level
- Example: Unprivileged popf
    - Privileged level: ALU & system flags
    - De-privileged level: ALU changes
    - No trap in de-privileged level

Simple Binary Translation – Interpreter

- Use of an interpreter
- Prevent leakage of privileged state
- Correct implementation of non-trapping instructions
- Separation of virtual state from physical state
- Fails Performance Criteria

Simple Binary Translation

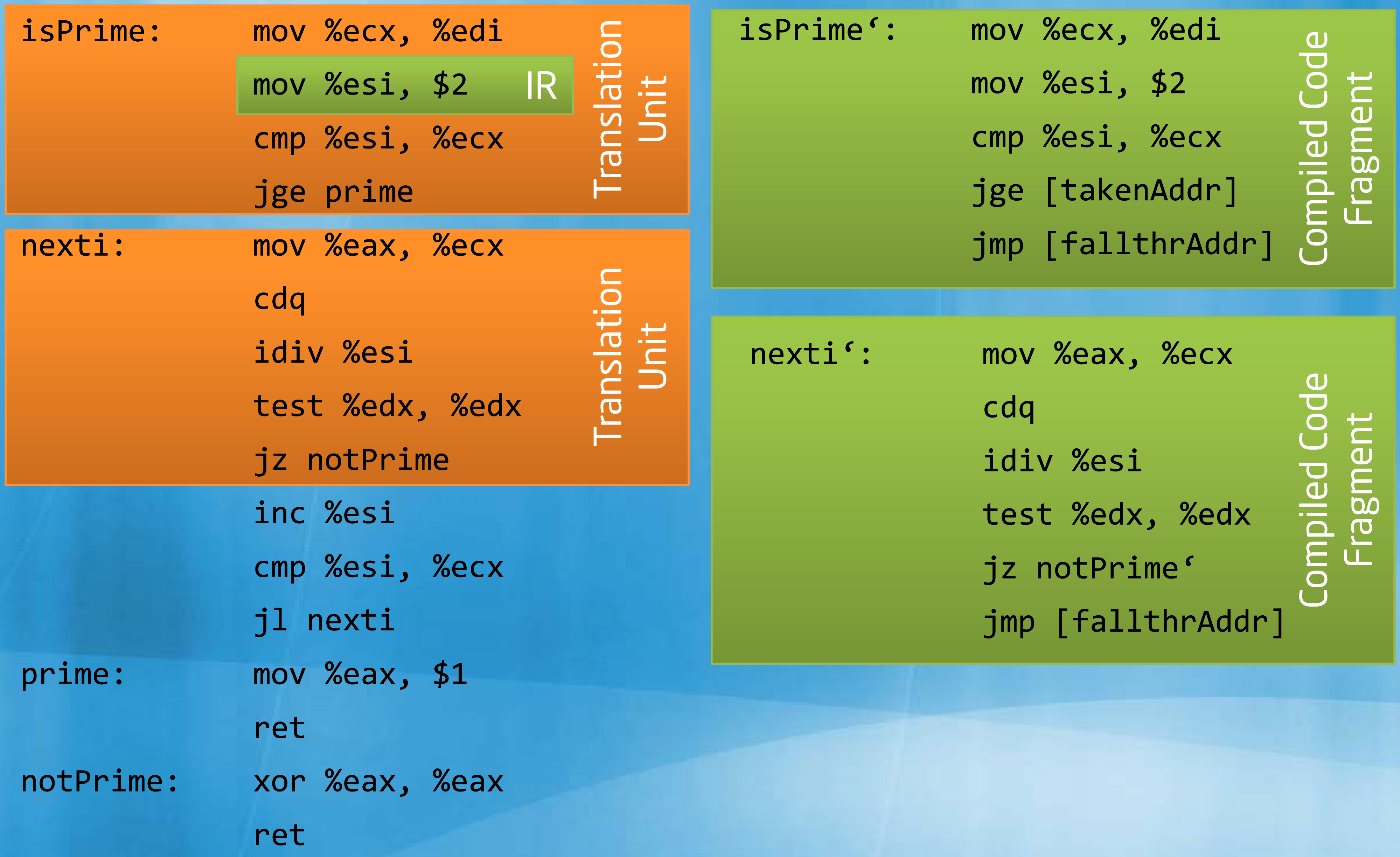
- Binary Translation combines Interpreter with Performance
- VMware's Translator offers this properties:
  - Binary
  - Dynamic
  - On Demand
  - System Level
  - Sub-Setting
  - Adaptive

Simple Binary Translation – Example

- Simple prime validation
- Invoke `isPrime(49)`

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

Simple Binary Translation – Example



```
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
```

```
isPrime‘:     mov %ecx, %edi
              mov %esi, $2
              cmp %esi, %ecx
              jge [takenAddr]
              jmp [fallthrAddr]
```

```
nexti‘:       mov %eax, %ecx
              cdq
              idiv %esi
              test %edx, %edx
              jz notPrime‘
              jmp [fallthrAddr]
```

Simple Binary Translation – Example

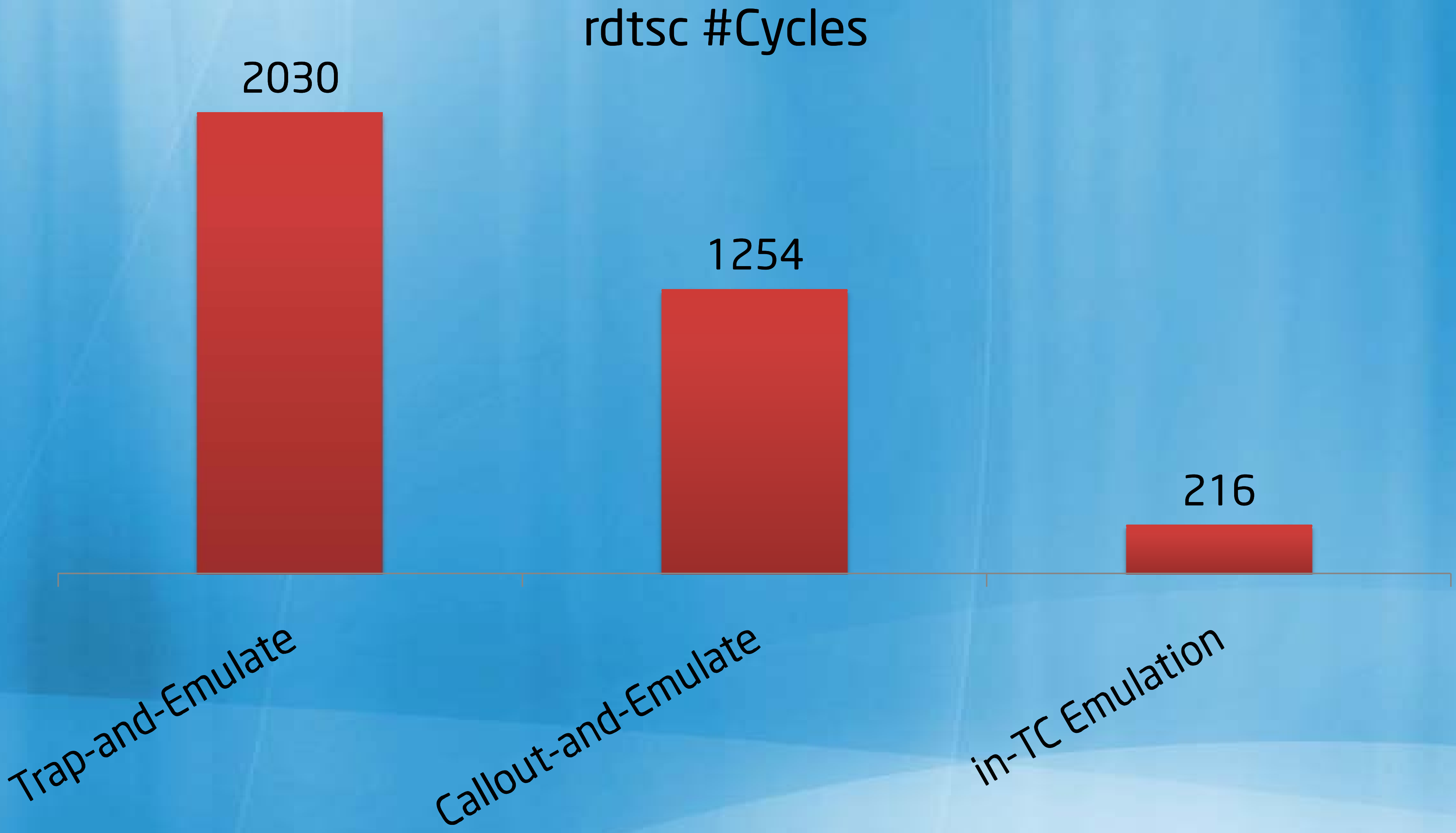
```
isPrime:      mov %ecx, %edi
              mov %esi, $2
              cmp %esi, %ecx
              jge prime
nexti:        mov %eax, %ecx
              cdq
              idiv %esi
              test %edx, %edx
              jz notPrime
              inc %esi
              cmp %esi, %ecx
        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

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

Simple Binary Translation – Exceptions

- PC-relative addressing
  - Translator output on different location
- Direct control flows
  - Code layout changes need reconnection
- Indirect control flows
  - Dynamically computed targets
- Privileged instructions

Comparison


rdtsc #Cycles
2030
1254
216
Trap-and-Emulate
Callout-and-Emulate
in-TC Emulation

HARDWARE VIRTUALIZATION

x86 Architecture Extensions

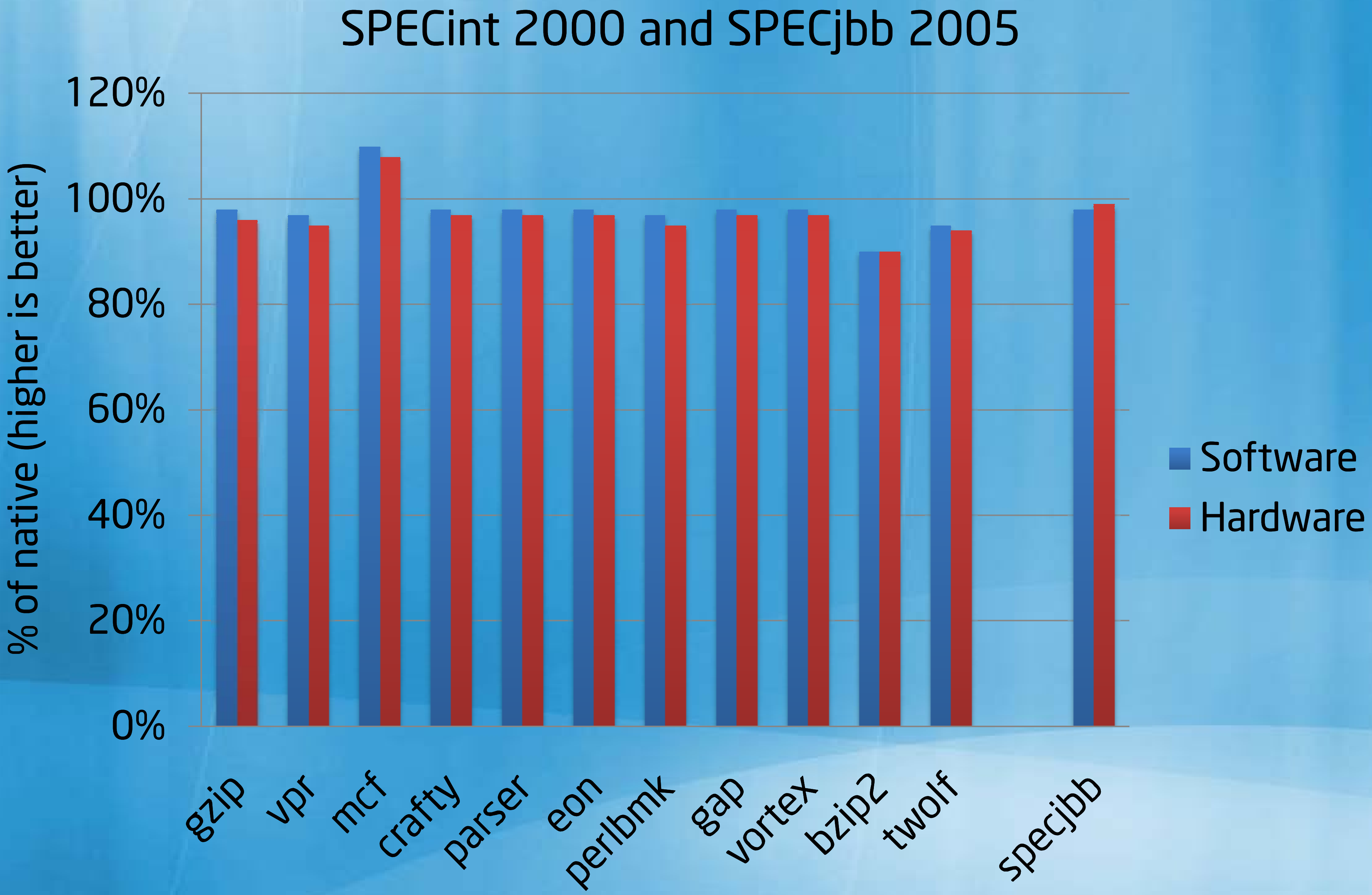
- Allows classical Trap-and-Emulate
- Virtual Machine Control Block
  - Diagnostics Fields
- Guest Mode (VMX) vs. Host Mode
- `vmrun` Command

Qualitative Comparision

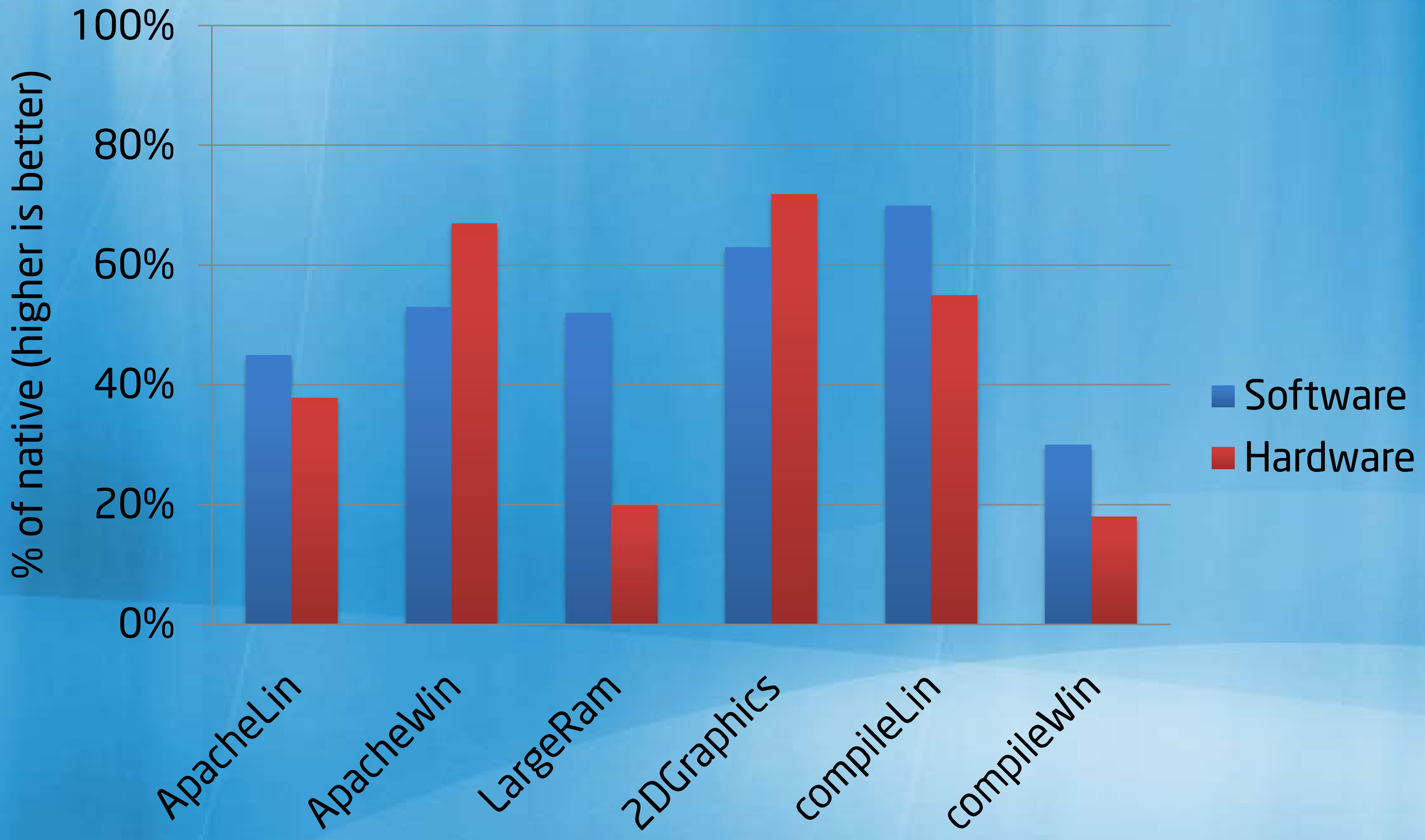
- Binary Translator
  - Trap Elimination
  - Emulation Speed
  - Callout avoidance
- Hardware-assisted VMM
  - Code density
  - Precise exceptions
  - System calls

EXPERIMENTS AND RESULTS

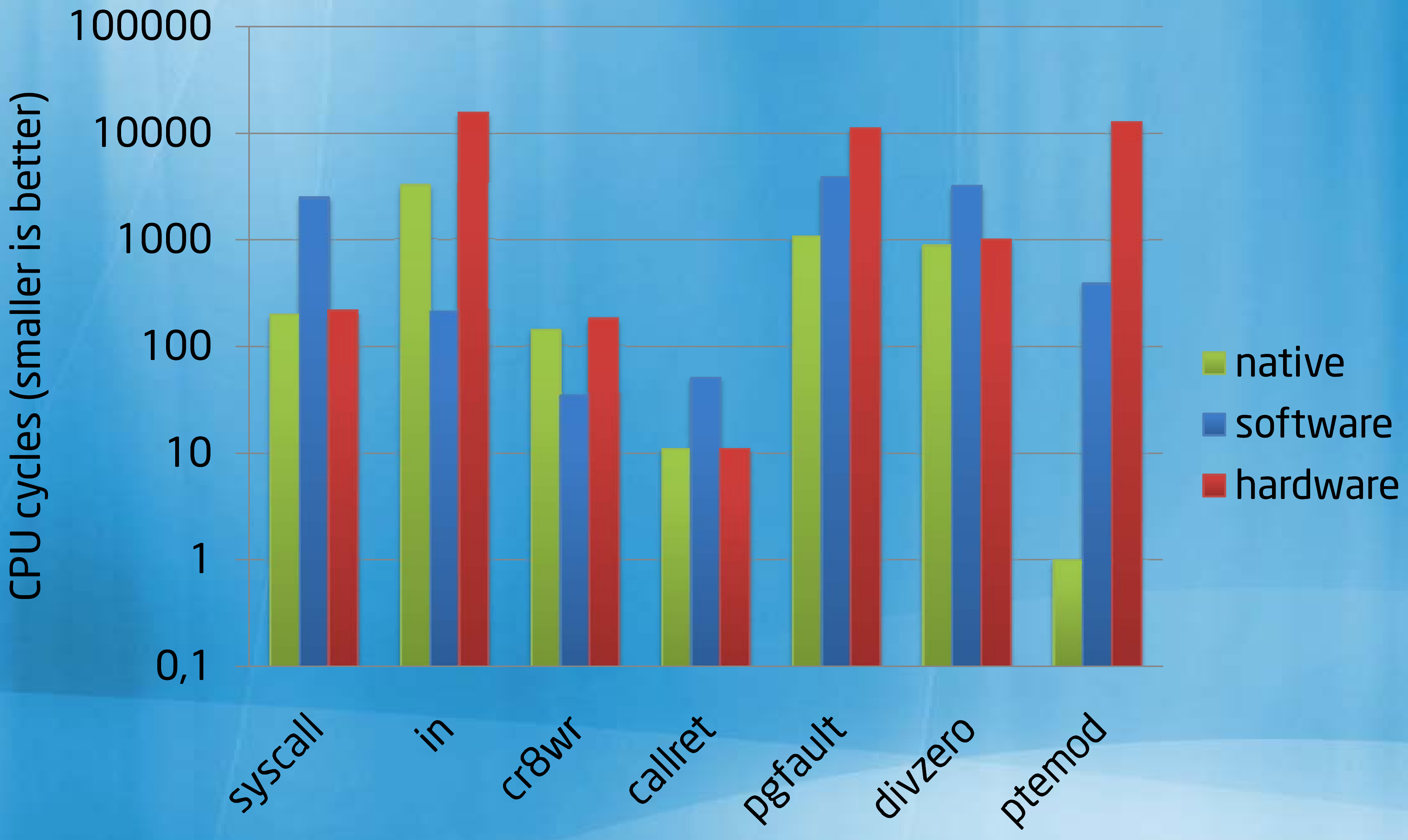
Initial Measuring



Macrobenchmarks



Cost of Operations



Opportunities

- Faster MicroCoreArchitecture Implementations
- Hardware VMM Algorithms
- Hybrid VMM
- Hardware MMU

Conclusions

- First generation of hardware support
  - Permit tran-and-emulate
- No real performance decrease
  - Only at system calls
- New MMU algorithms could help

Michael Wallner

A COMPARISON OF SOFTWARE AND HARDWARE TECHNIQUES FOR X86 VIRTUALIZATION