Bare-Metal Performance for x86 Virtualization

Muli Ben-Yehuda

Technion & IBM Research

Background: x86 machine virtualization

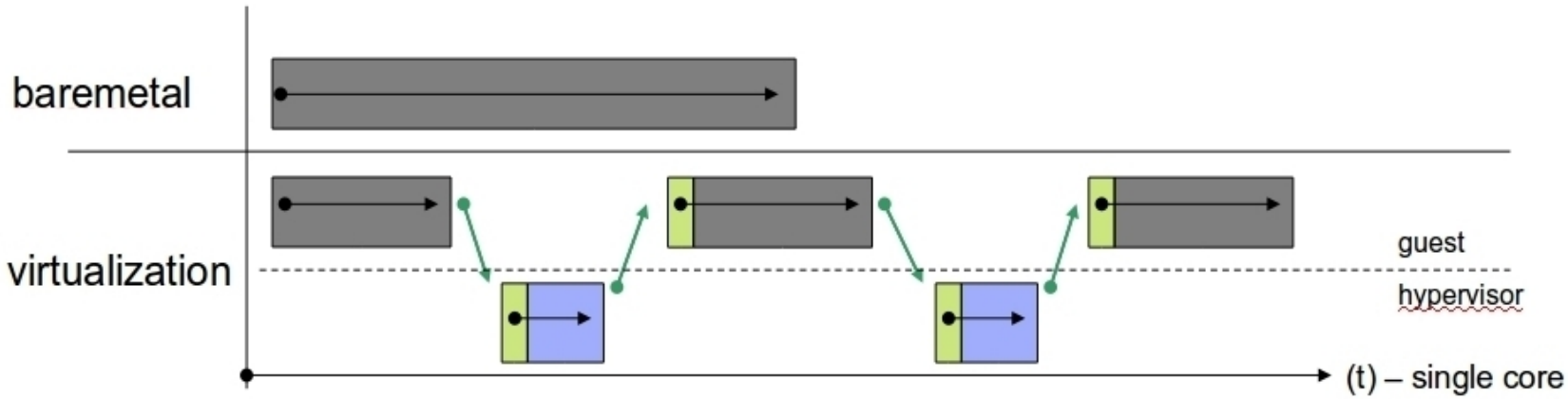
- Running multiple different unmodified operating systems
- Each in an isolated virtual machine
- Simultaneously
- On the x86 architecture
- Many uses: live migration, record & replay, testing, security, ...
- Foundation of IaaS cloud computing
- Used nearly everywhere

The problem is performance

- Machine virtualization can reduce performance by orders of magnitude [Adams06,Santos08,Ram09,Ben-Yehuda10,Amit11,...]
- Overhead limits use of virtualization in many scenarios
- We would like to make it possible to use virtualization everywhere
- Including I/O intensive workloads and nested workloads
- Where does the overhead come from?

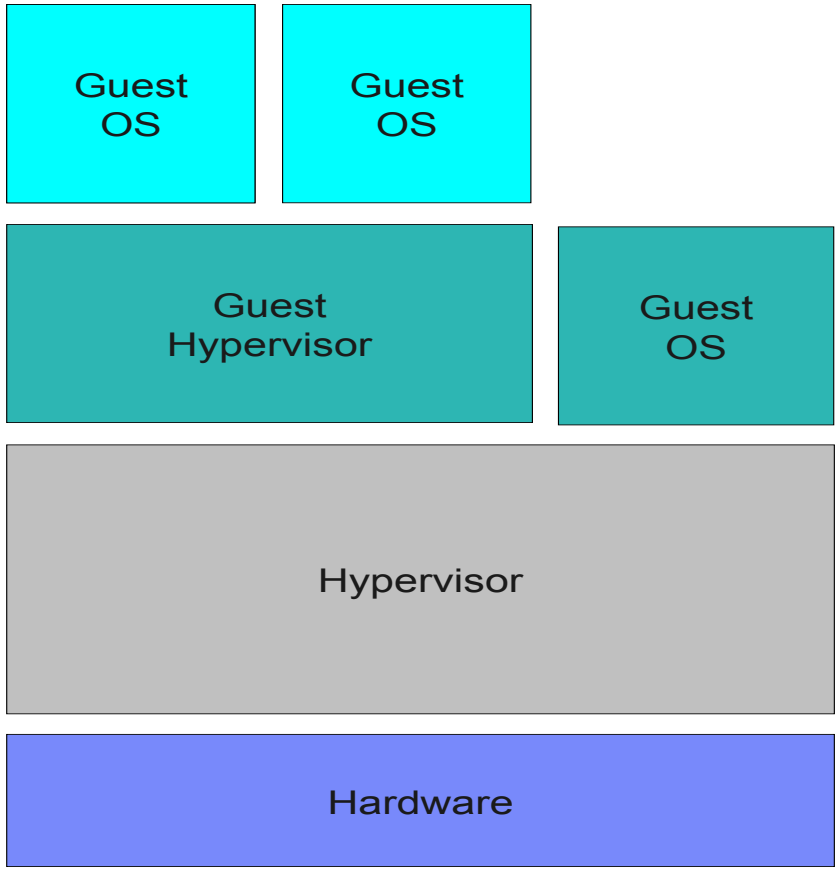
The origin of overhead

- Popek and Goldberg's virtualization model [Popek74]: Trap and emulate
- Privileged instructions trap to the hypervisor
- Hypervisor emulates their behavior
- Traps cause an exit
- I/O intensive workloads cause many exits
- Nested workloads cause many exits



What is nested x86 virtualization?

- Running multiple unmodified hypervisors
- With their associated unmodified VM's
- Simultaneously
- On the x86 architecture
- Which does not support nesting in hardware...
- ...but does support a single level of virtualization



Why?

- Operating systems are already hypervisors (Windows 7 with XP mode, Linux/KVM)
- Security: attack via or defend against hypervisor-level rootkits such as Blue Pill
- To be able to run other hypervisors in clouds
- Co-design of x86 hardware and system software
- Testing, demonstrating, debugging, live migration of hypervisors

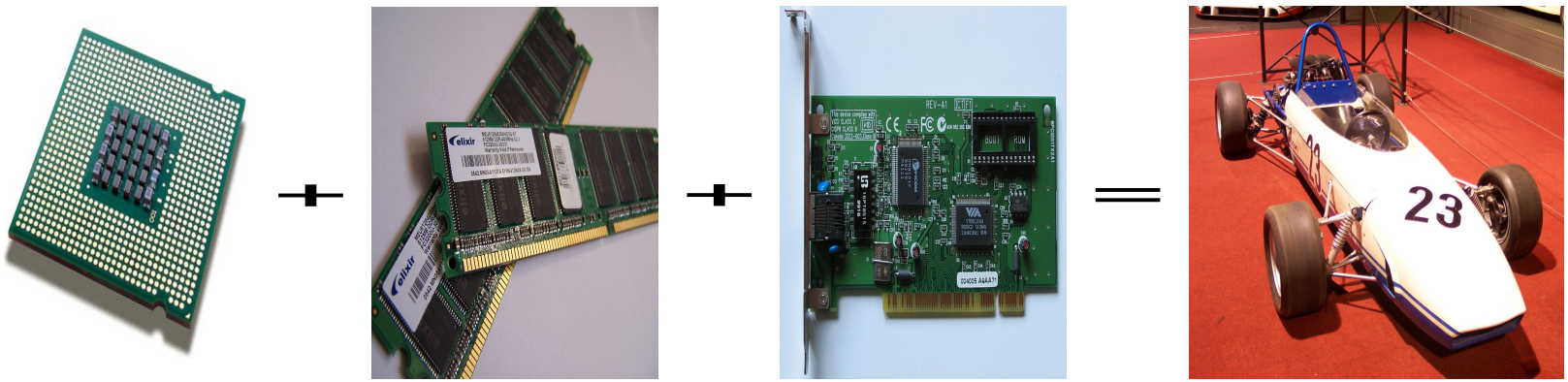
What is the Turtles project?

- Efficient nested virtualization for VMX based on KVM
- Runs multiple guest hypervisors, including VMware, Windows

"The Turtles Project: Design and Implementation of Nested Virtualization", Ben-Yehuda, Day, Dubitzky, Factor, Hare'El, Gordon, Liguori, Wasserman and Yassour, OSDI '10

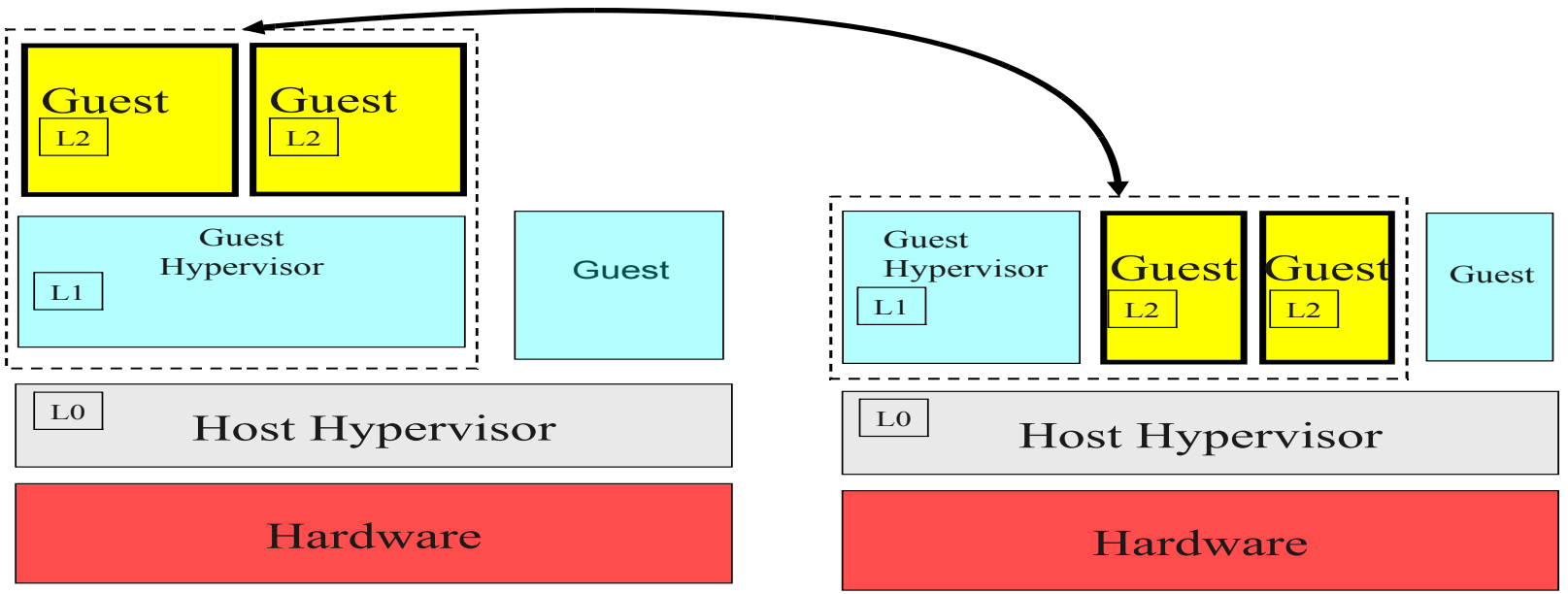
What is the Turtles project? (cont')

- Nested VMX virtualization for nested CPU virtualization
- Multi-dimensional paging for nested MMU virtualization
- Multi-level device assignment for nested I/O virtualization
- Micro-optimizations to make it go fast



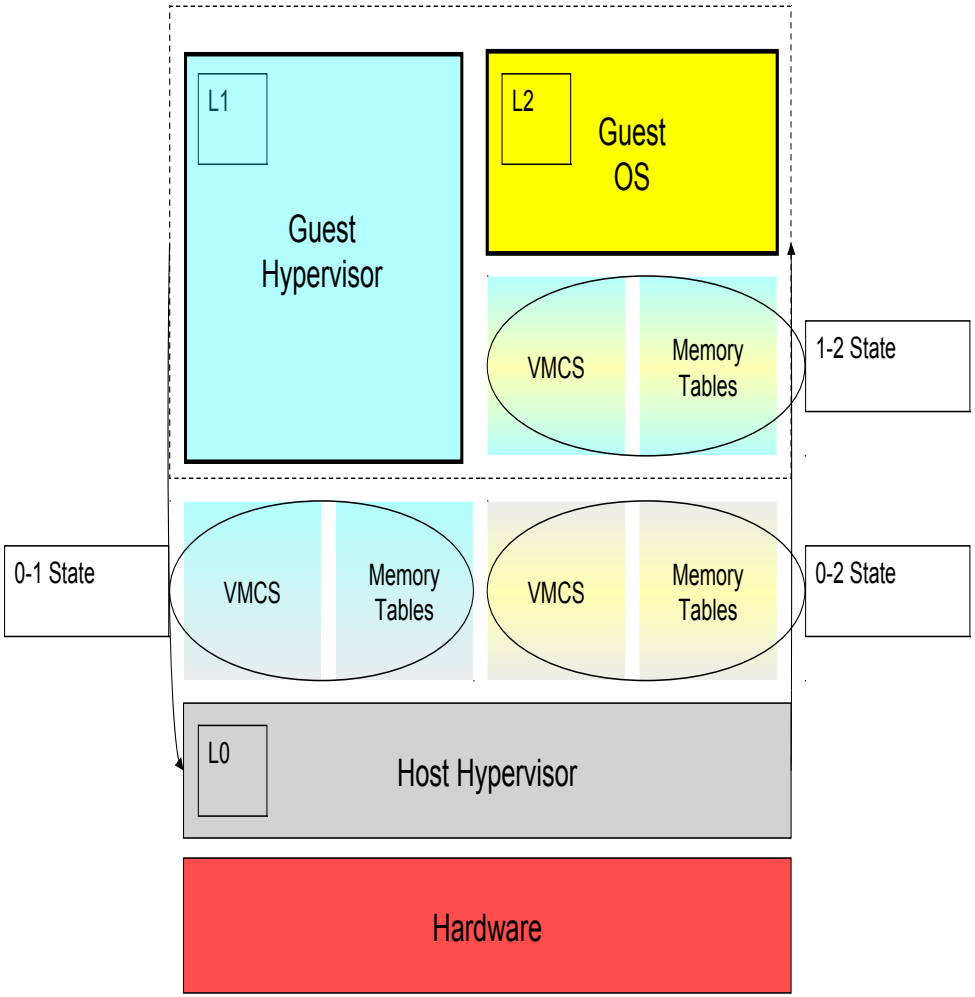
Theory of nested CPU virtualization

- Trap and emulate[PopekGoldberg74] $\Rightarrow$ it's all about the traps
- Single-level (x86) vs. multi-level (e.g., z/VM)
- Single level $\Rightarrow$ one hypervisor, many guests
- Turtles approach: L_0 multiplexes the hardware between L_1 and L_2, running both as guests of L_0—without either being aware of it
- (Scheme generalized for n levels; Our focus is n=2)



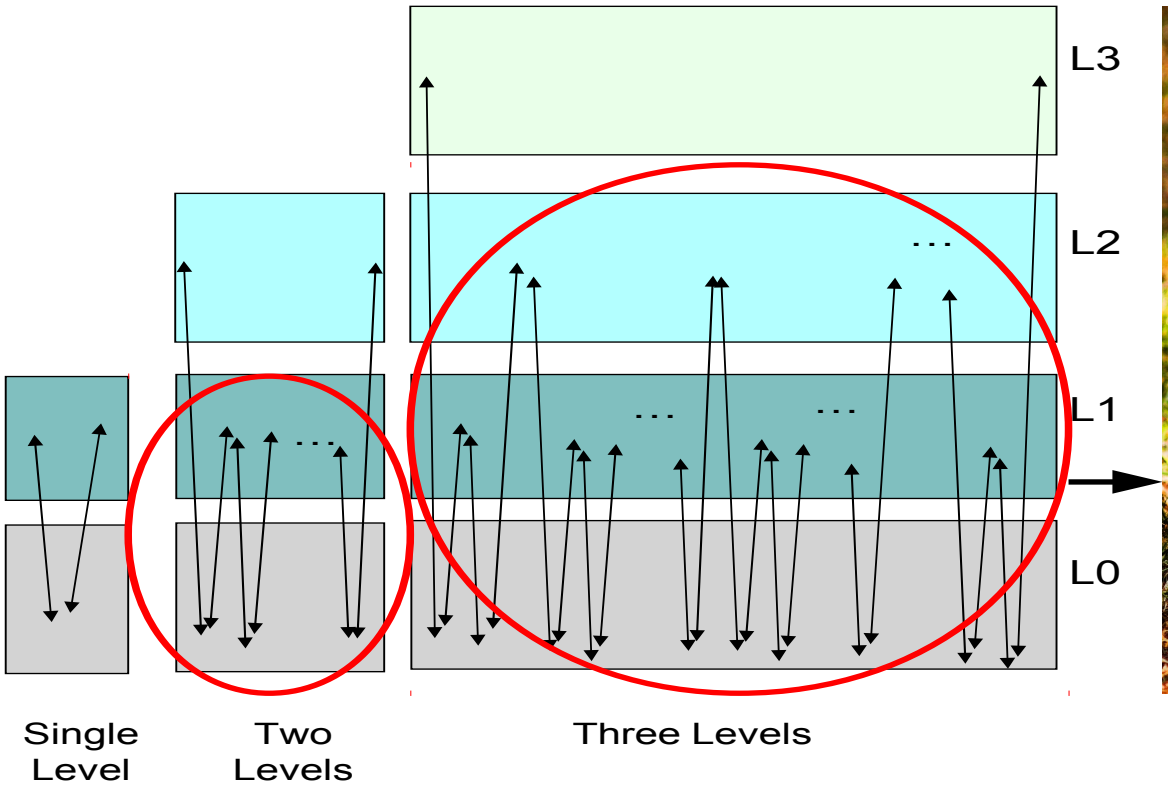
Nested VMX virtualization: flow

- L_0 runs L_1 with $VMCS_{0\to1}$
- L_1 prepares $VMCS_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges VMCS's: $VMCS_{0\to1}$ merged with $VMCS_{1\to2}$ is $VMCS_{0\to2}$
- L_0 launches L_2
- L_2 causes a trap
- L_0 handles trap itself or forwards it to L_1
- ...
- eventually, L_0 resumes L_2
- repeat



Exit multiplication makes angry turtle angry

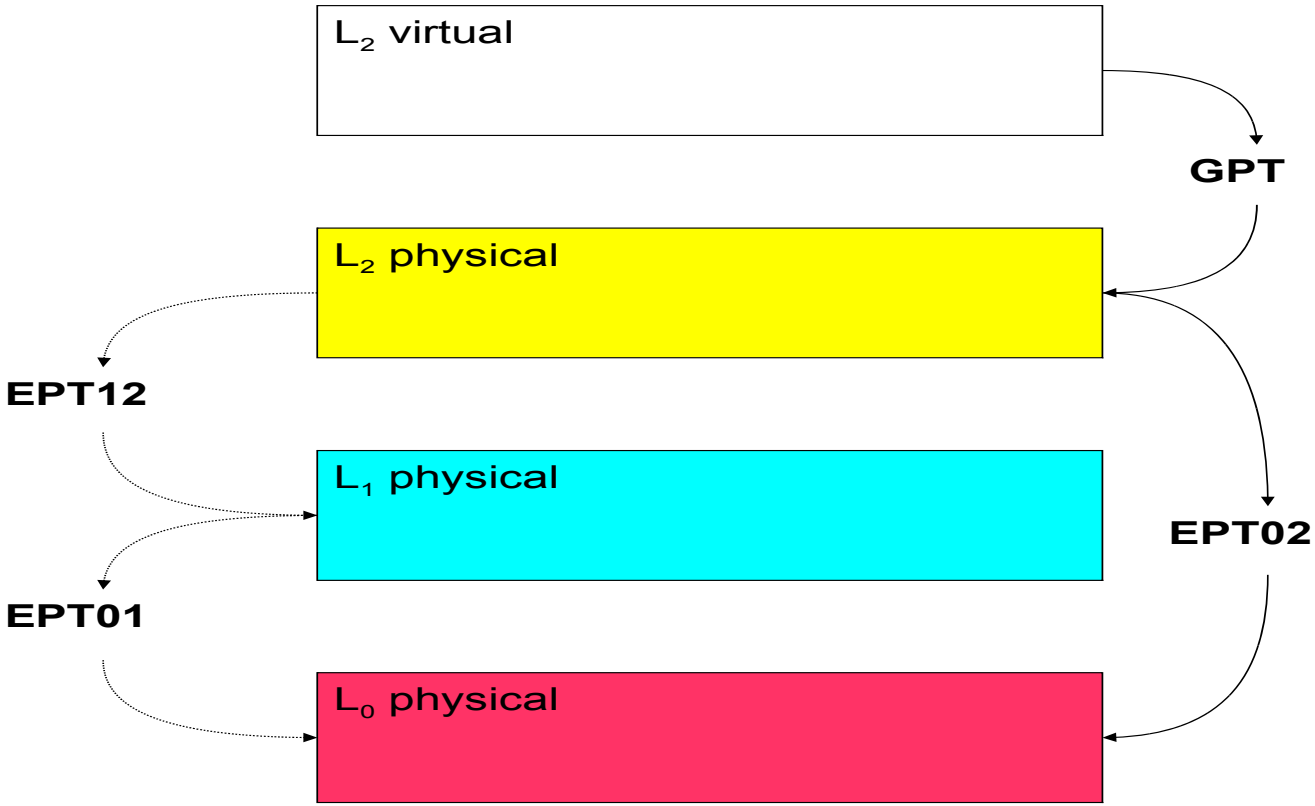
- To handle a single L_2 exit, L_1 does many things: read and write the VMCS, disable interrupts, ...
- Those operations can trap, leading to exit multiplication
- Exit multiplication: a single L_2 exit can cause 40-50 L_1 exits!
- Optimize: make a single exit fast and reduce frequency of exits



Introduction to x86 MMU virtualization

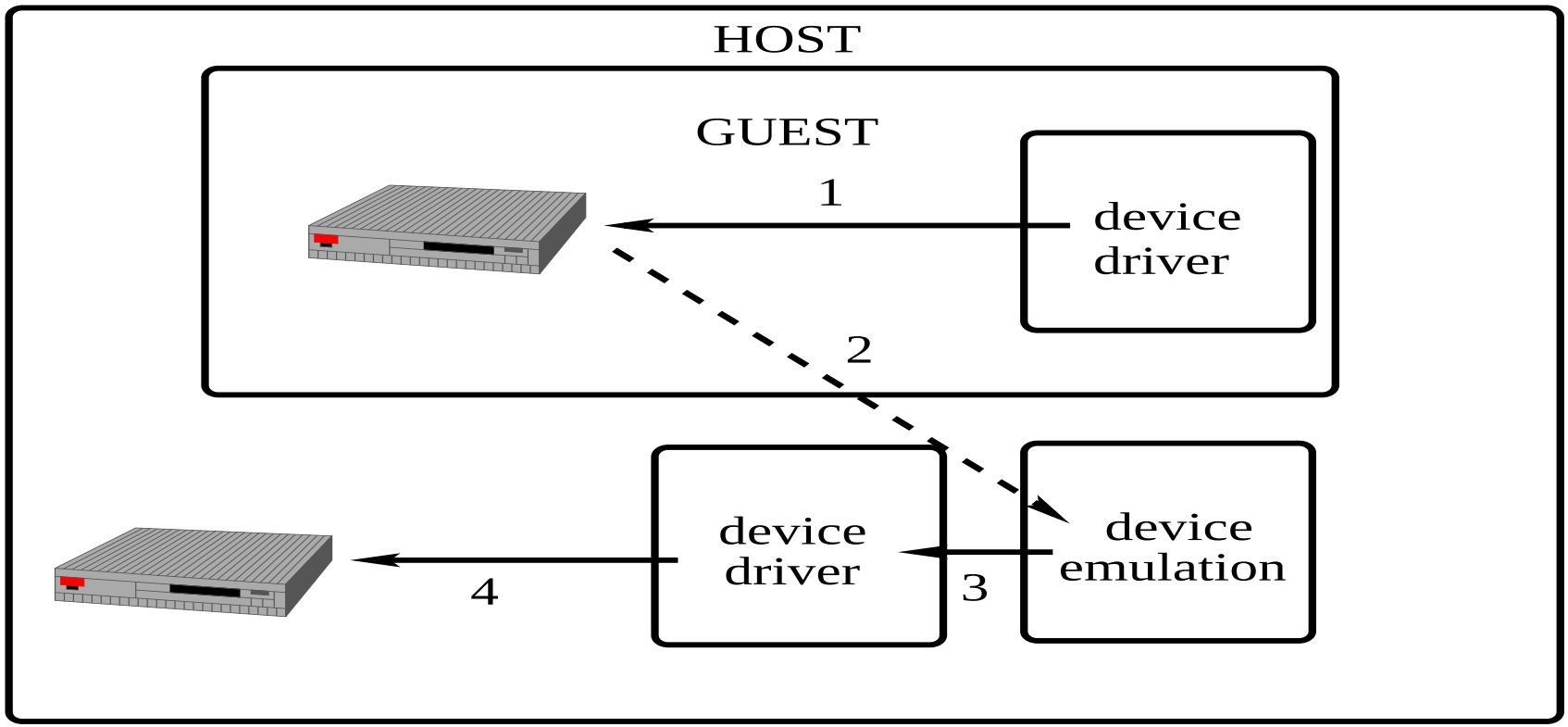
- x86 does page table walks in hardware
- MMU has one currently active hardware page table
- Bare metal $\Rightarrow$ only needs one logical translation, (virtual $\rightarrow$ physical)
- Virtualization $\Rightarrow$ needs two logical translations
  1. Guest page table: (guest virt $\rightarrow$ guest phys)
  2. Host page table: (guest phys $\rightarrow$ host phys)
- . . . but MMU only knows to walk a single table!

Solution: multi-dimensional paging



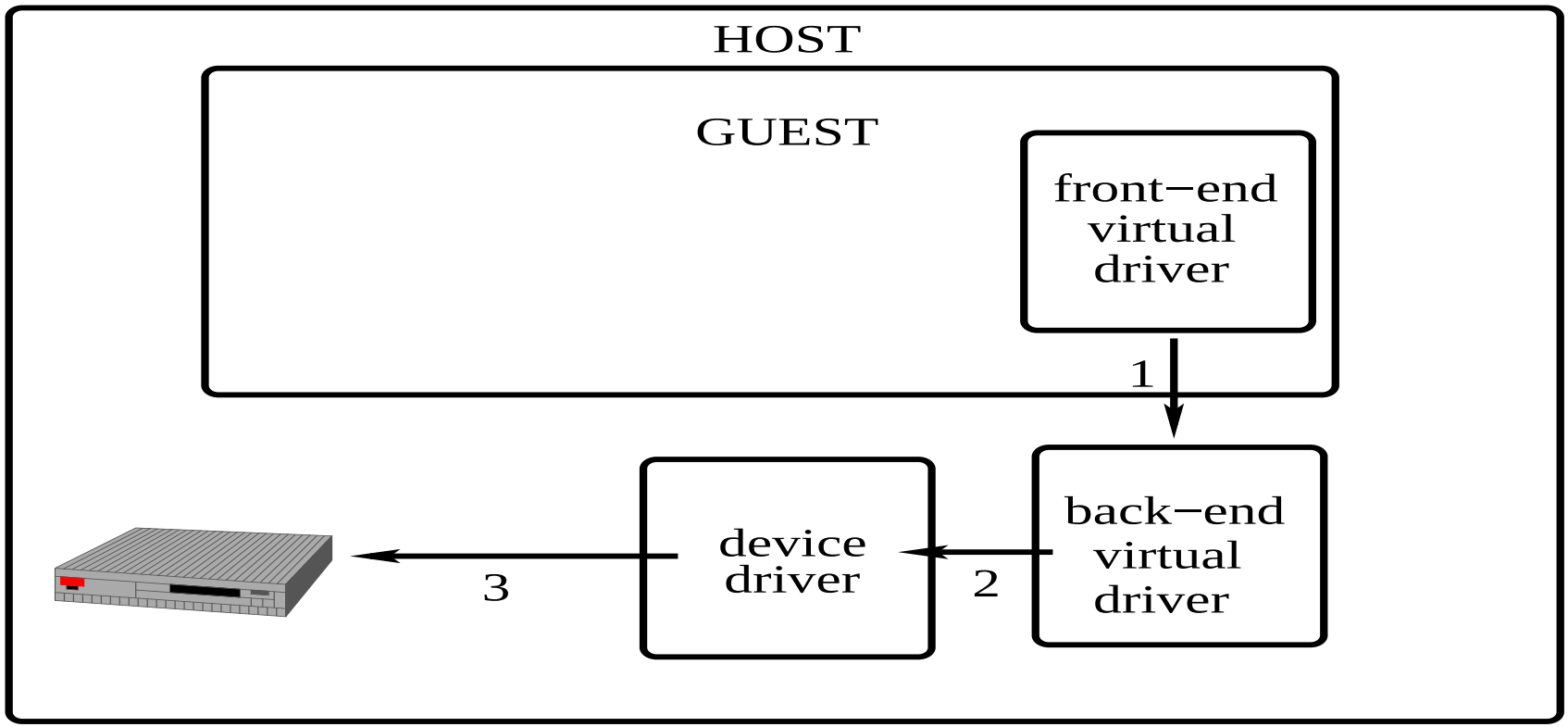
- EPT table rarely changes; guest page table changes a lot
- Compress three logical translations $\Rightarrow$ two in hardware
- L_0 emulates EPT for L_1
- L_0 uses $EPT_{0\rightarrow1}$ and $EPT_{1\rightarrow2}$ to construct $EPT_{0\rightarrow2}$
- End result: a lot less exits!

I/O virtualization via device emulation



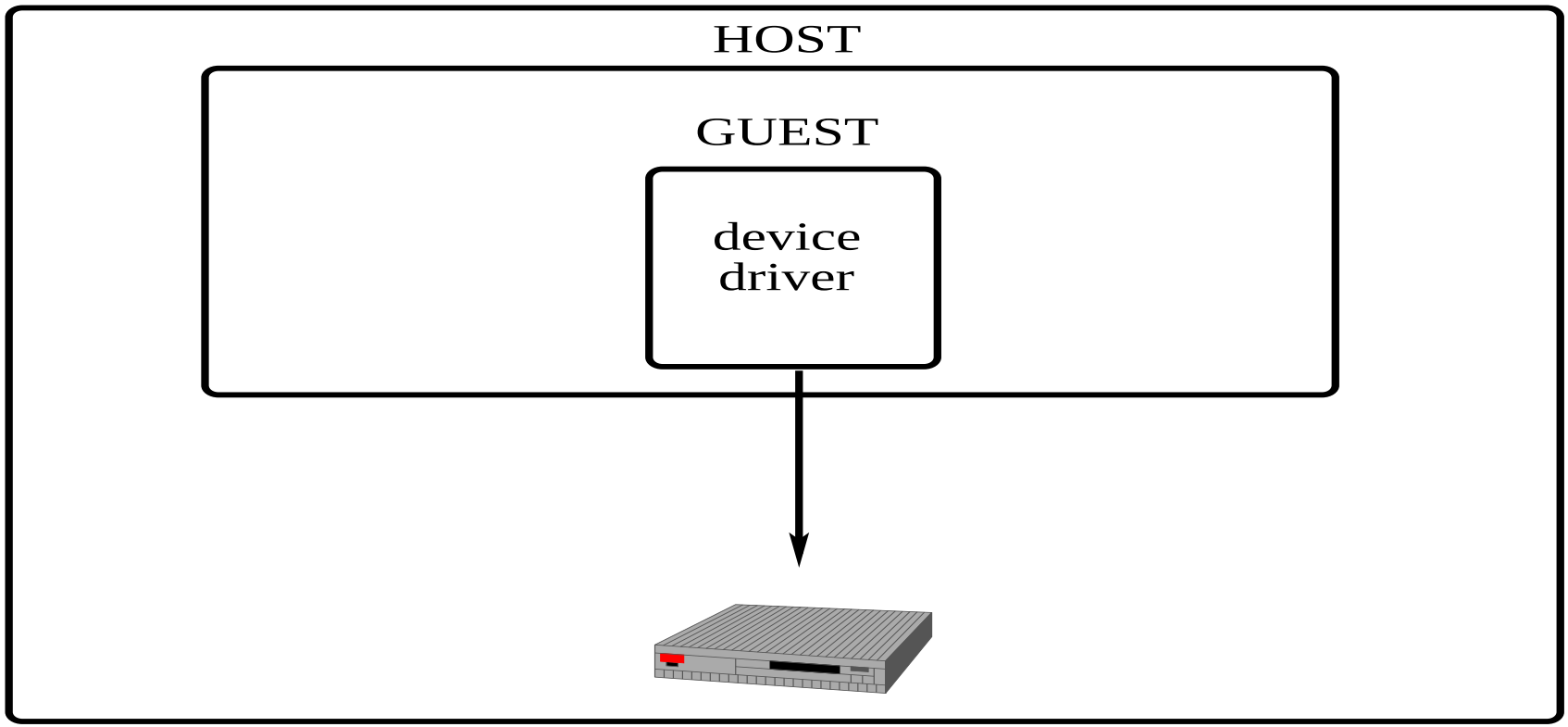
- Emulation is usually the default [Sugerman01]
- Works for unmodified guests out of the box
- Very low performance, due to many exits on the I/O path

I/O virtualization via paravirtualized devices



- Hypervisor aware drivers and “devices” [Barham03,Russell08]
- Requires new guest drivers
- Requires hypervisor involvement on the I/O path

I/O virtualization via device assignment



- Bypass the hypervisor on I/O path [Levasseur04,Ben-Yehuda06]
- SR-IOV devices provide sharing in hardware
- Better performance than paravirtual—but far from native

Comparing I/O virtualization methods

IOV method	throughput (Mb/s)	CPU utilization
bare-metal	950	20%
device assignment	950	25%
paravirtual	950	50%
emulation	250	100%

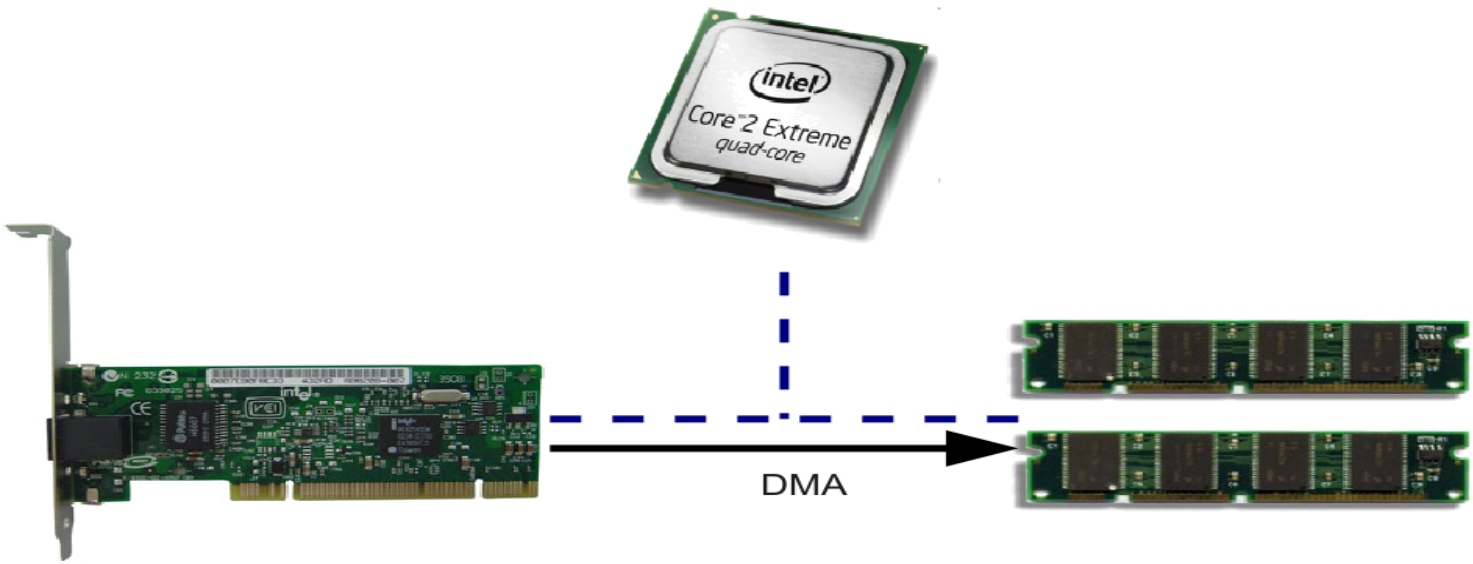
- `netperf` TCP_STREAM sender on 1Gb/s Ethernet (16K msgs)
- Device assignment best performing option

What does it mean, to do I/O?

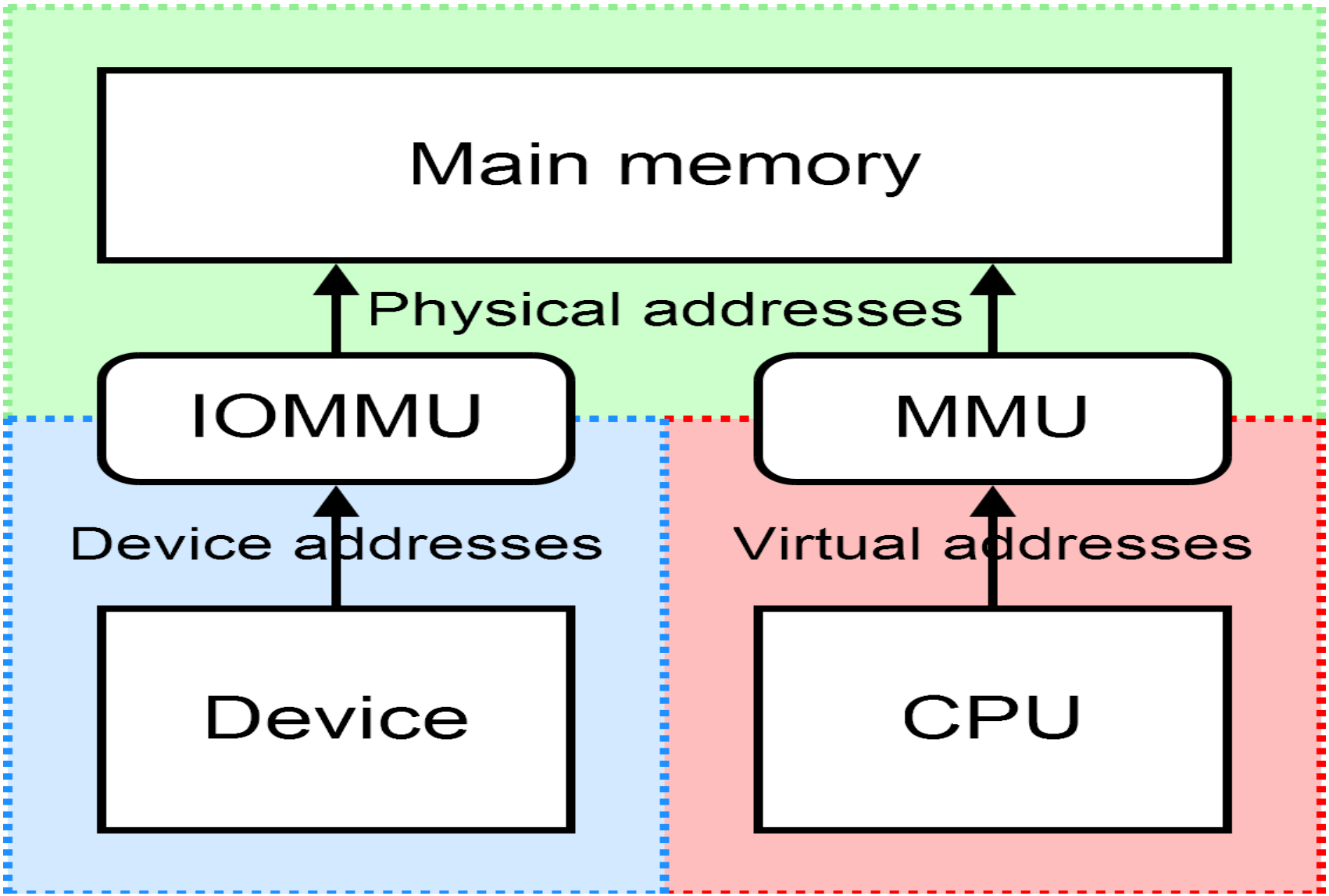
- Programmed I/O (in/out instructions)
- Memory-mapped I/O (loads and stores)
- Direct memory access (DMA)
- Interrupts

Direct memory access (DMA)

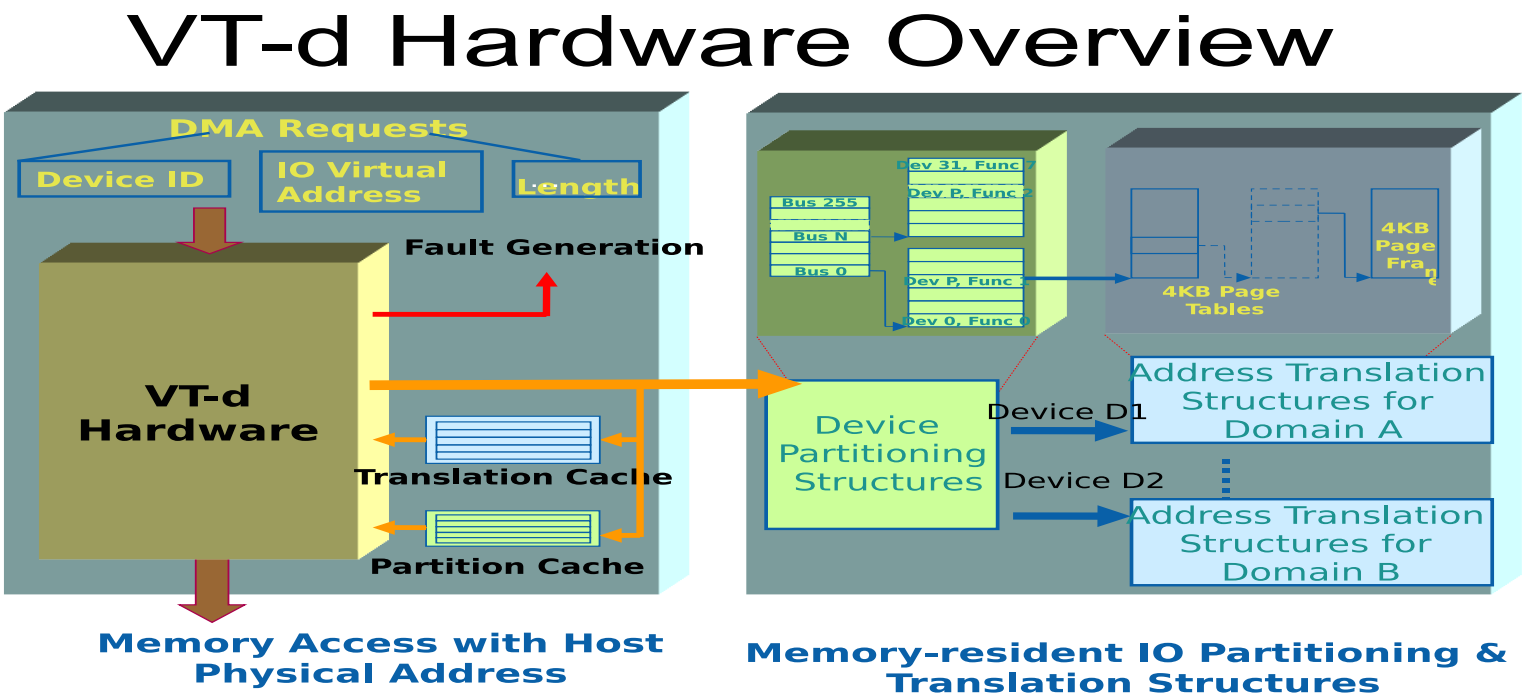
- All modern devices access memory directly
- On bare-metal:
  - A trusted driver gives its device an address
  - Device reads or writes that address
- Protection problem: guest drivers are not trusted
- Translation problem: guest memory $\neq$ host memory
- Direct access: the guest bypasses the host
- What to do?



IOMMU


Main memory
Physical addresses
IOMMU
MMU
Device addresses
Virtual addresses
Device
CPU

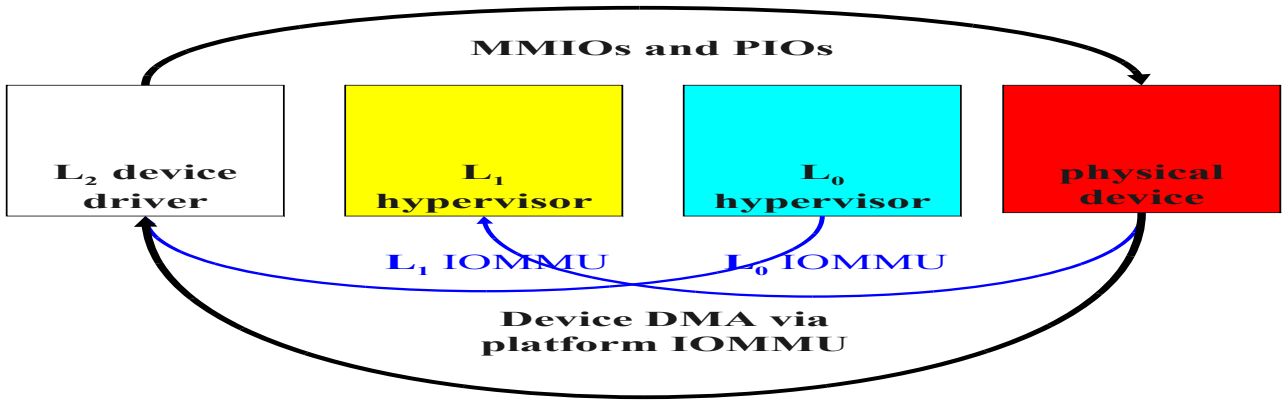
The IOMMU mapping memory/performance tradeoff



- When does the host map and unmap translation entries?
- Direct mapping up-front on virtual machine creation: all memory is pinned, no intra-guest protection
- During run-time: high cost in performance
- We want: direct mapping performance, intra-guest protection, minimal pinning

Multi-level device assignment

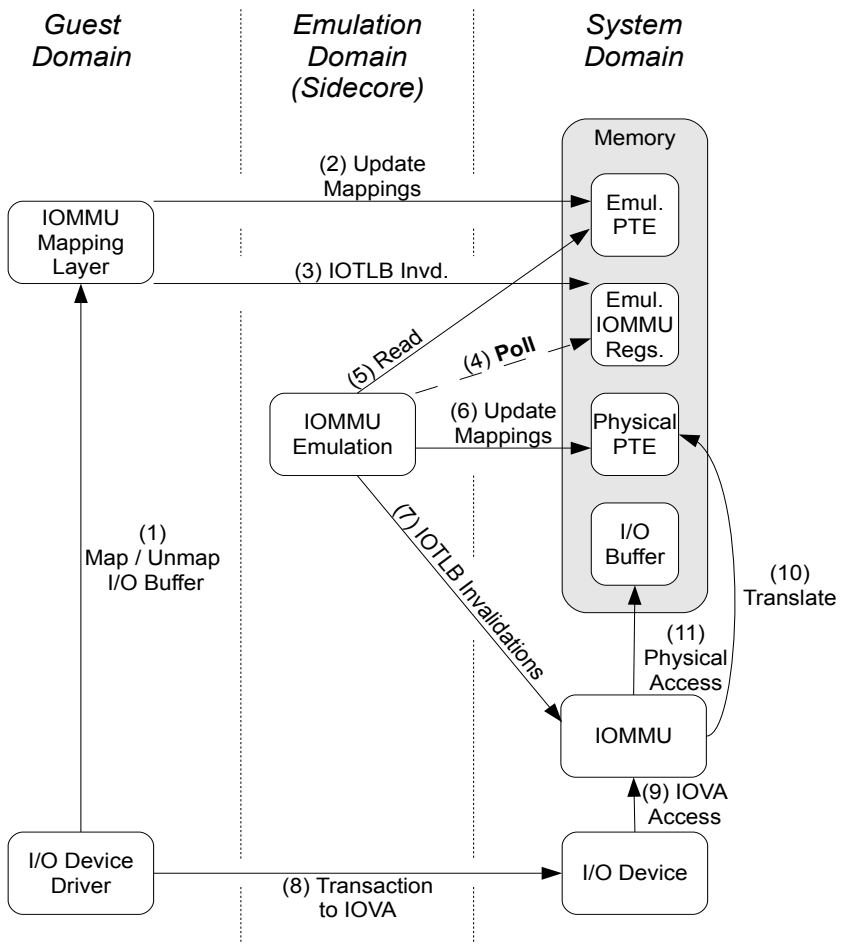
- With nested 3x3 options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- Multi-level device assignment means giving an L_2 guest direct access to L_0's devices, safely bypassing both L_0 and L_1



- Requires that L_0 emulate an IOMMU efficiently
- L_0 compresses multiple IOMMU translations onto the single hardware IOMMU page table
- L_2 programs the device directly
- Device DMA's into L_2 memory space directly

vIOMMU: efficient IOMMU emulation

- Emulate an IOMMU so that we know when to map and unmap: enable memory-overcommitment, intra-guest protection
- Use a sidecore [Kumar07] for efficient emulation: avoid costly exits by running emulation on another core in parallel
- Optimistic teardown: relax protection to increase performance by caching translation entries
- vIOMMU provides high performance with intra-guest protection and minimal pinning



"vIOMMU: Efficient IOMMU Emulation", Amit, Ben-Yehuda, Schuster, Tsafrir,

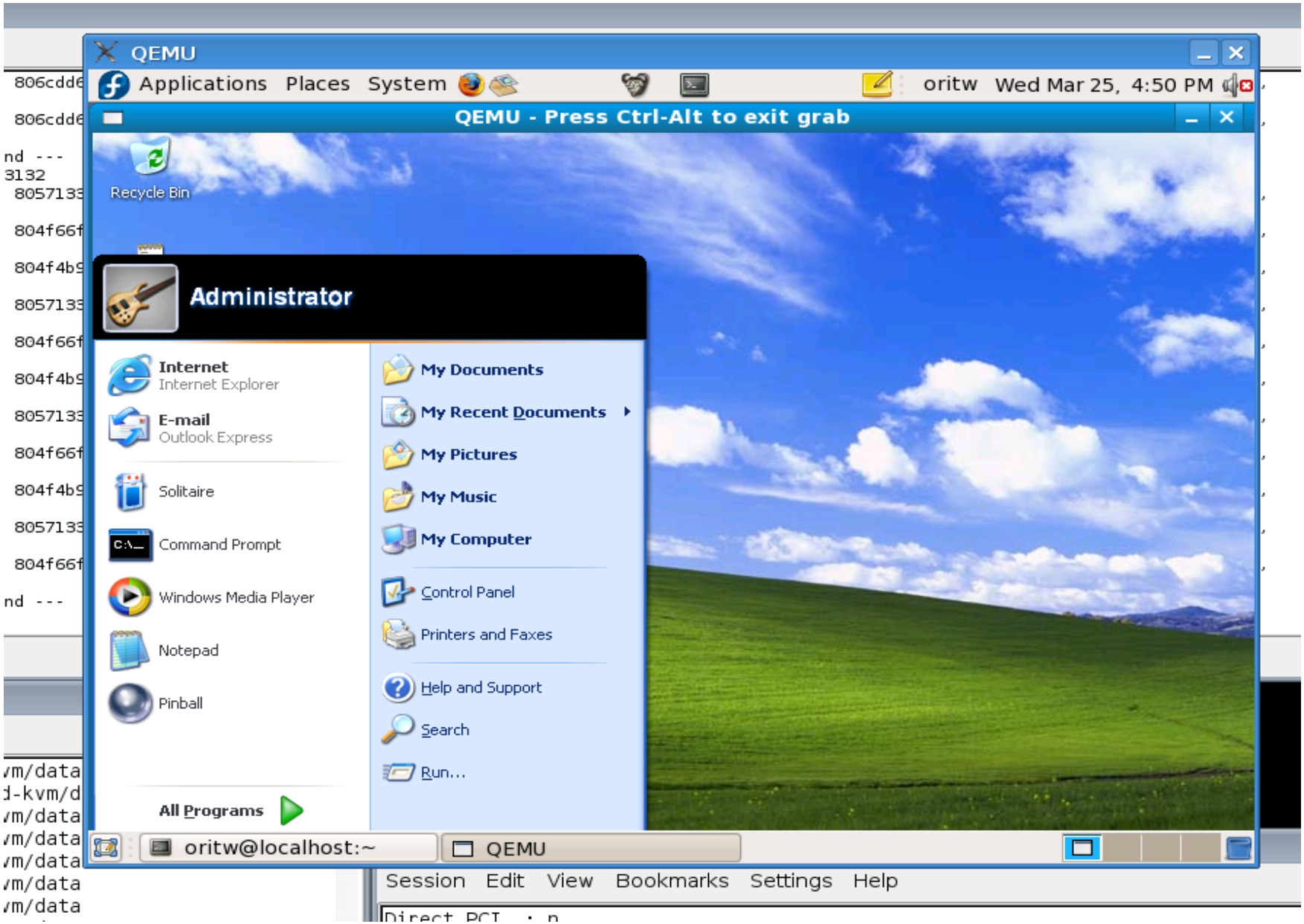
Micro-optimizations

- Goal: reduce world switch overheads
- Reduce cost of single exit by focus on VMCS merges:
  - Keep VMCS fields in processor encoding
  - Partial updates instead of whole-sale copying
  - Copy multiple fields at once
  - Some optimizations not safe according to spec
- Reduce frequency of exits—focus on `vmread` and `vmwrite`
  - Avoid the exit multiplier effect
  - Loads/stores vs. architected trapping instructions
  - Binary patching?

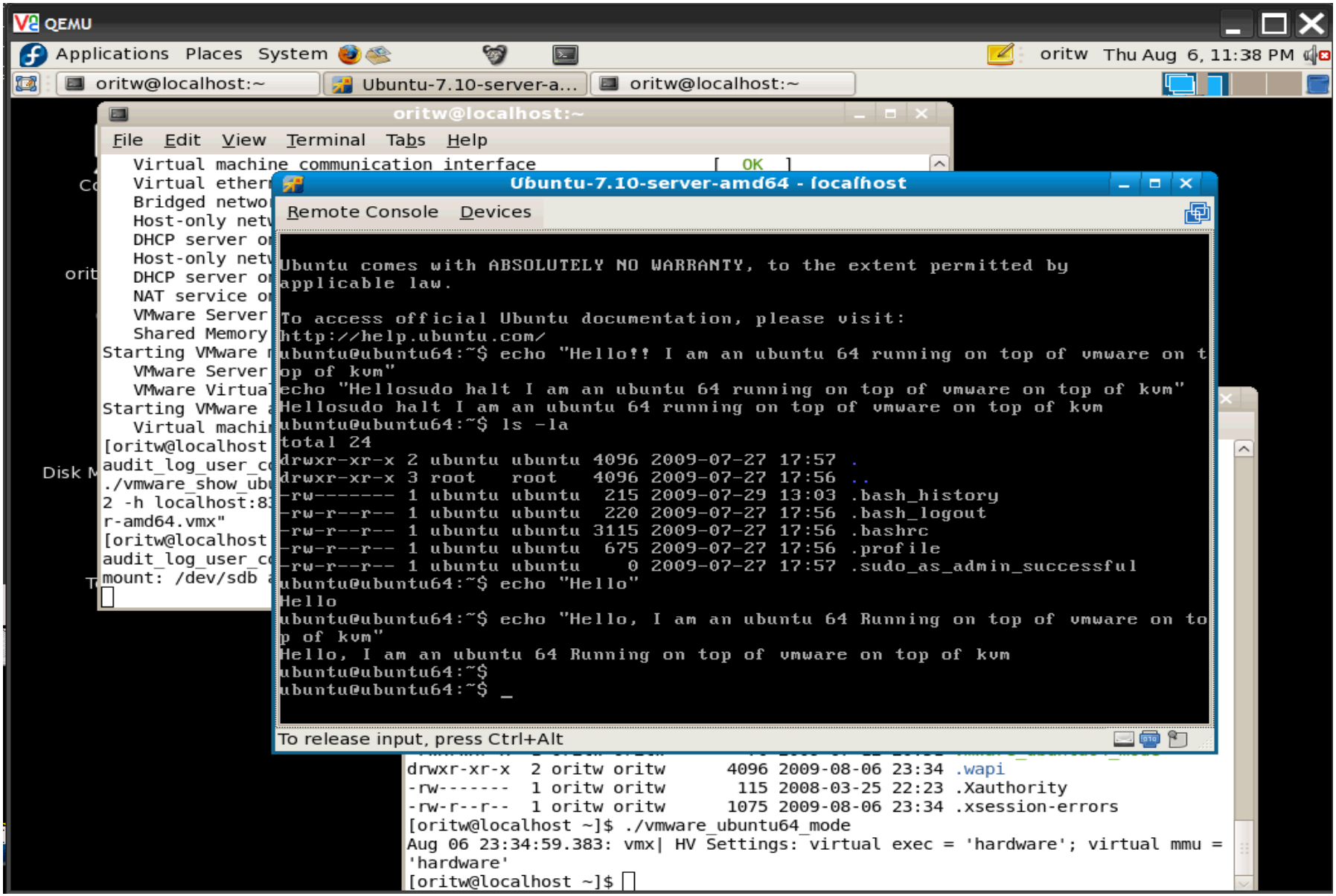
Nested VMX support in KVM

```
Date: Mon, 16 May 2011 22:43:54 +0300
From: Nadav Har'El <nyh (at) il.ibm.com>
To: kvm [at] vger.kernel.org
Cc: gleb [at] redhat.com, avi [at] redhat.com
Subject: [PATCH 0/31] nVMX: Nested VMX, v10

Hi,

This is the tenth iteration of the nested VMX patch set. Improvements in this
version over the previous one include:

 * Fix the code which did not fully maintain a list of all VMCSs loaded on
   each CPU. (Avi, this was the big thing that bothered you in the previous
   version).

 * Add nested-entry-time (L1->L2) verification of control fields of vmcs12 -
   procbased, pinbased, entry, exit and secondary controls - compared to the
   capability MSRs which we advertise to L1.

[many other changes trimmed]

This new set of patches applies to the current KVM trunk (I checked with
6f1bd0daae731ff07f4755b4f56730a6e4a3c1cb).
If you wish, you can also check out an already-patched version of KVM from
branch "nvmx10" of the repository:
         git://github.com/nyh/kvm-nested-vmx.git
```

Windows XP on KVM L_1 on KVM L_0



Linux on VMware L_1 on KVM L_0



Experimental Setup

- Running Linux, Windows, KVM, VMware, SMP, ...
- Macro workloads:
  - kernbench
  - SPECjbb
  - netperf
- Multi-dimensional paging?
- Multi-level device assignment?
- KVM as L_1 vs. VMware as L_1?
- See paper for full experimental details and more benchmarks and analysis

Macro: SPECjbb and kernbench

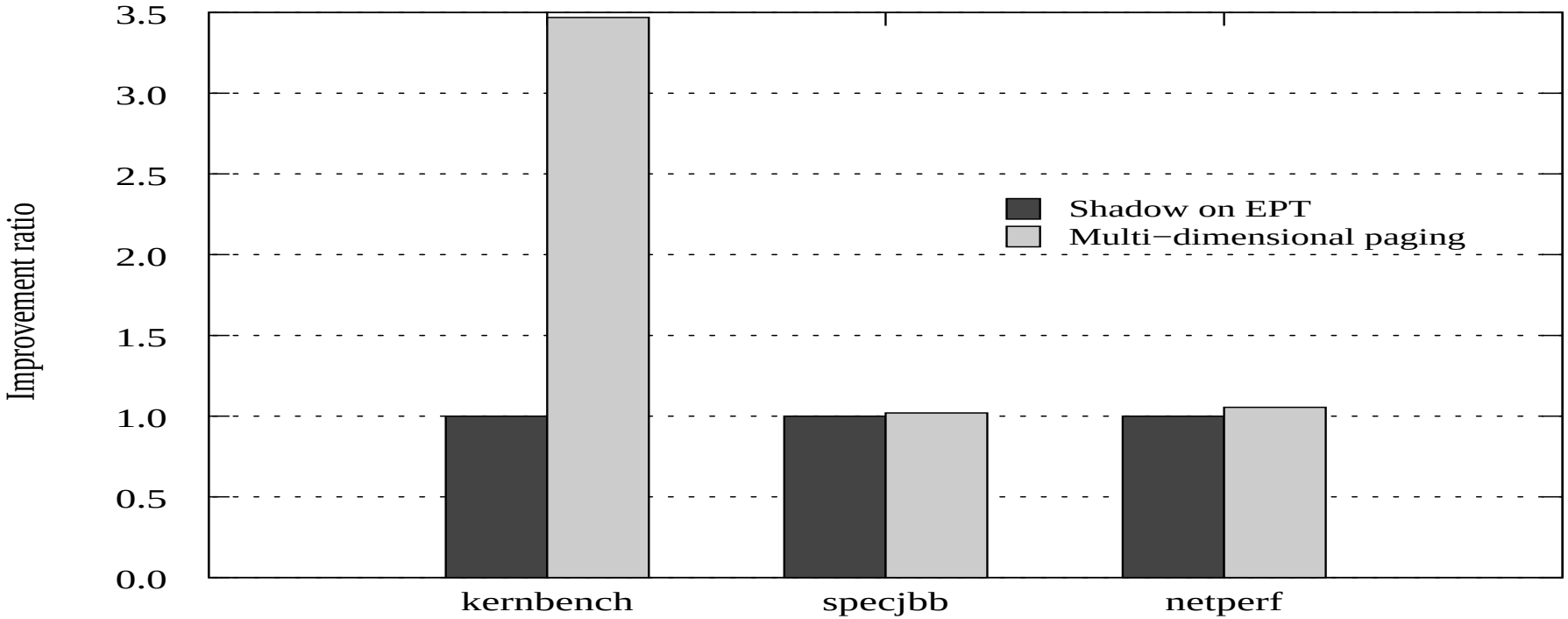
kernbench				
	Host	Guest	Nested	Nested$_{DRW}$
Run time	324.3	355	406.3	391.5
% overhead vs. host	-	9.5	25.3	20.7
% overhead vs. guest	-	-	14.5	10.3

SPECjbb				
	Host	Guest	Nested	Nested$_{DRW}$
Score	90493	83599	77065	78347
% degradation vs. host	-	7.6	14.8	13.4
% degradation vs. guest	-	-	7.8	6.3

Table: kernbench and SPECjbb results

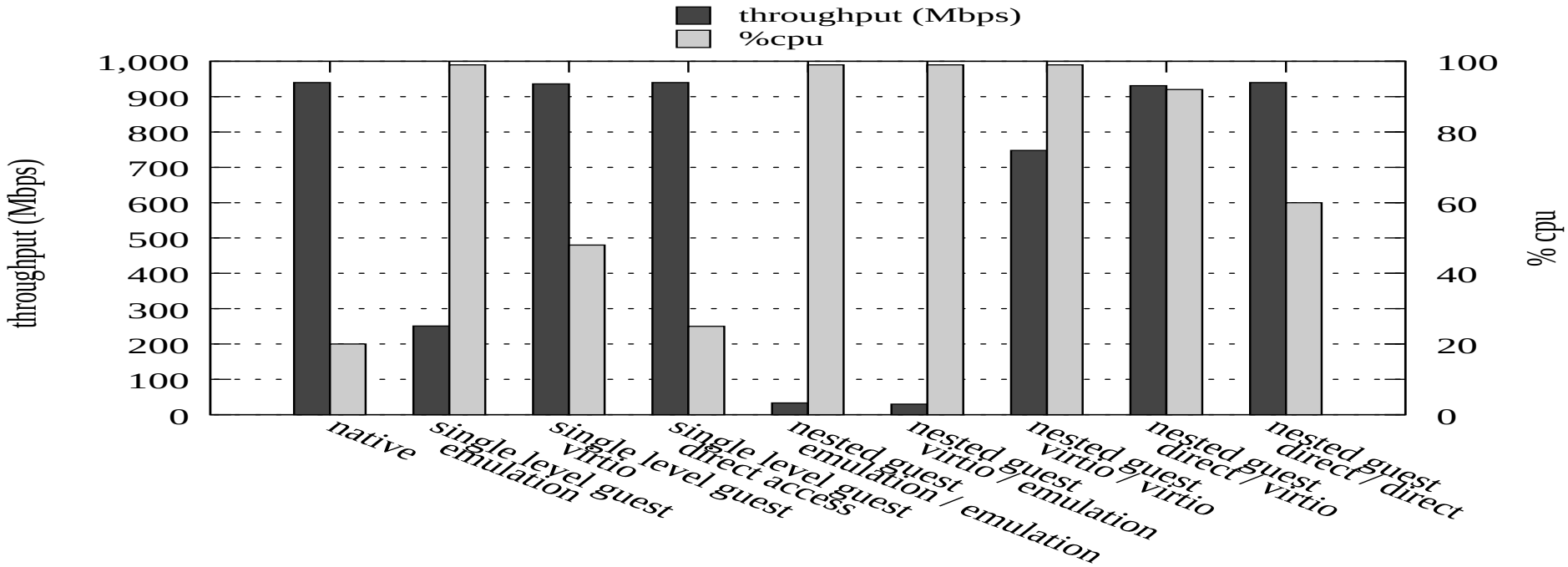
- Exit multiplication effect not as bad as we feared
- Direct vmread and vmwrite (DRW) give an immediate boost
- Take-away: each level of virtualization adds approximately the same overhead!

Macro: multi-dimensional paging



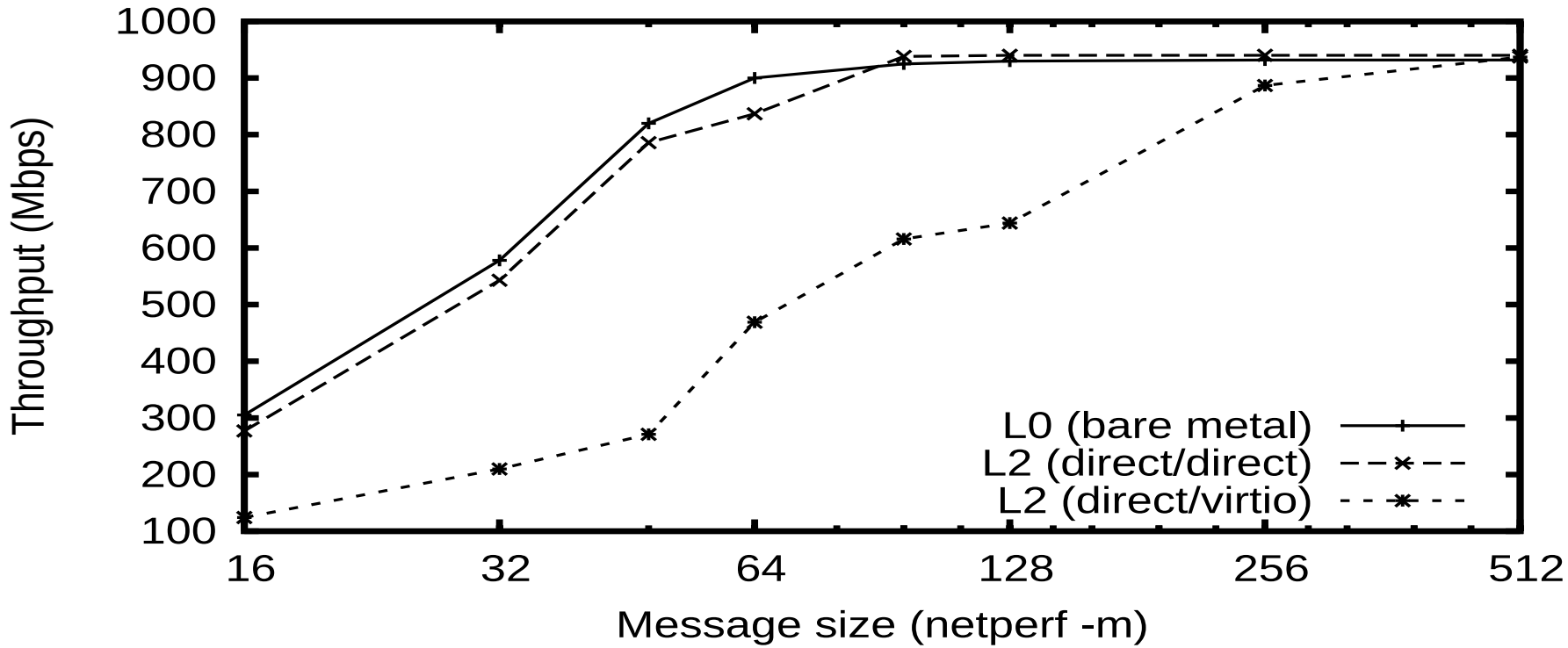
- Impact of multi-dimensional paging depends on rate of page faults
- Shadow-on-EPT: every L_2 page fault causes L_1 multiple exits
- Multi-dimensional paging: only EPT violations cause L_1 exits
- EPT table rarely changes: $\#(\text{EPT violations}) << \#(\text{page faults})$
- Multi-dimensional paging huge win for page-fault intensive `kernbench`

Macro: multi-level device assignment



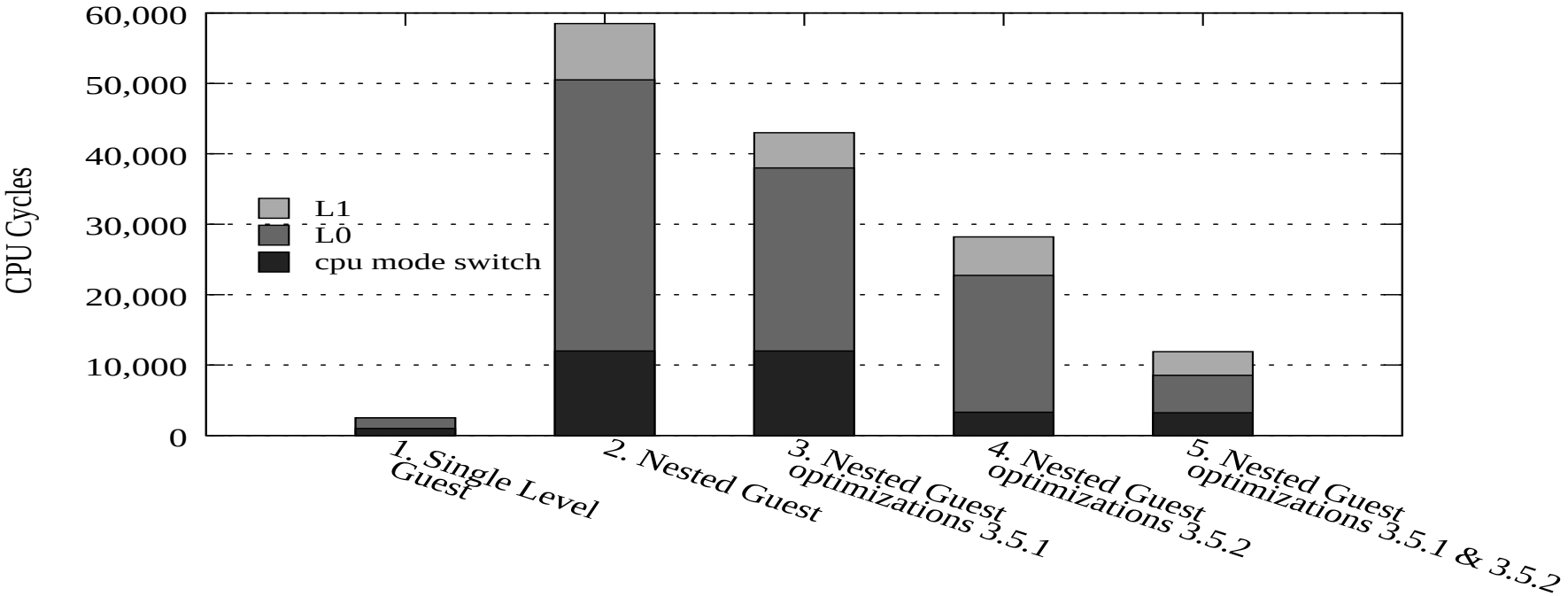
- Benchmark: `netperf TCP_STREAM` (transmit)
- Multi-level device assignment best performing option
- But: native at 20%, multi-level device assignment at 60% (x3!)
- Interrupts considered harmful, cause exit multiplication

Macro: multi-level device assignment (sans interrupts)



- What if we could deliver device interrupts directly to L_2?
- Only 7% difference between native and nested guest!

Micro: synthetic worst case `CPUID` loop



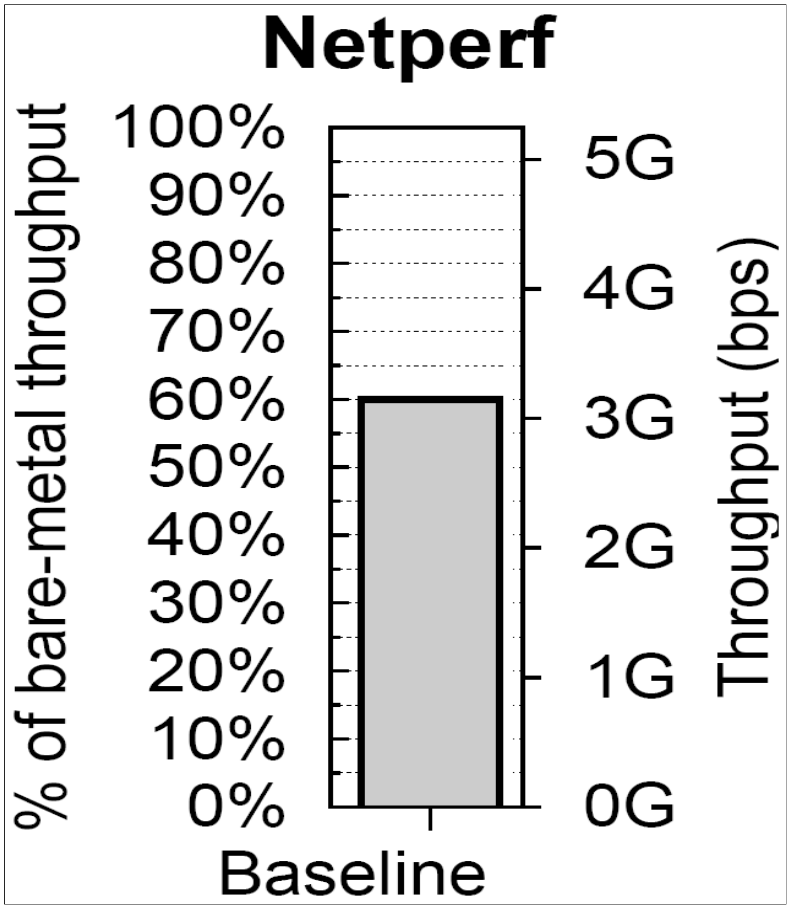
- CPUID running in a tight loop is not a real-world workload!
- Went from 30x worse to “only” 6x worse
- A nested exit is still expensive—minimize both single exit cost and frequency of exits

Interim conclusions: Turtles

- Efficient nested x86 virtualization is challenging but feasible
- A whole new ballpark opening up many exciting applications—security, cloud, architecture, ...
- Current overhead of 6-14%
  - Negligible for some workloads, not yet for others
  - Work in progress—expect at most 5% eventually
- Code is available
- Why Turtles?
  It's turtles all the way down

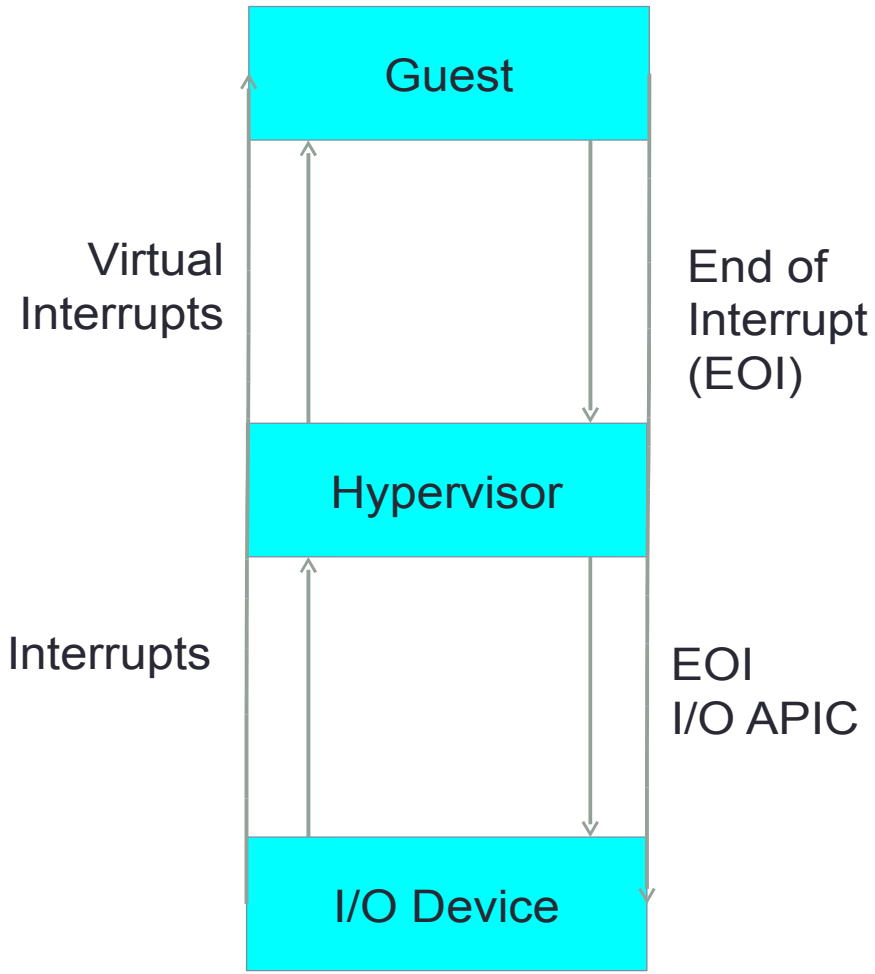
Back to interrupts: How bad is it?

- `netperf` TCP_STREAM sender on 10Gb/s Ethernet with 256 byte messages
- Using device assignment with direct mapping in the IOMMU
- Only achieves 60% of bare-metal performance
- Same results for `memcached` and `apache`
- Where does the rest go?
- Interrupts: approximately 49,000 interrupts per second with Linux

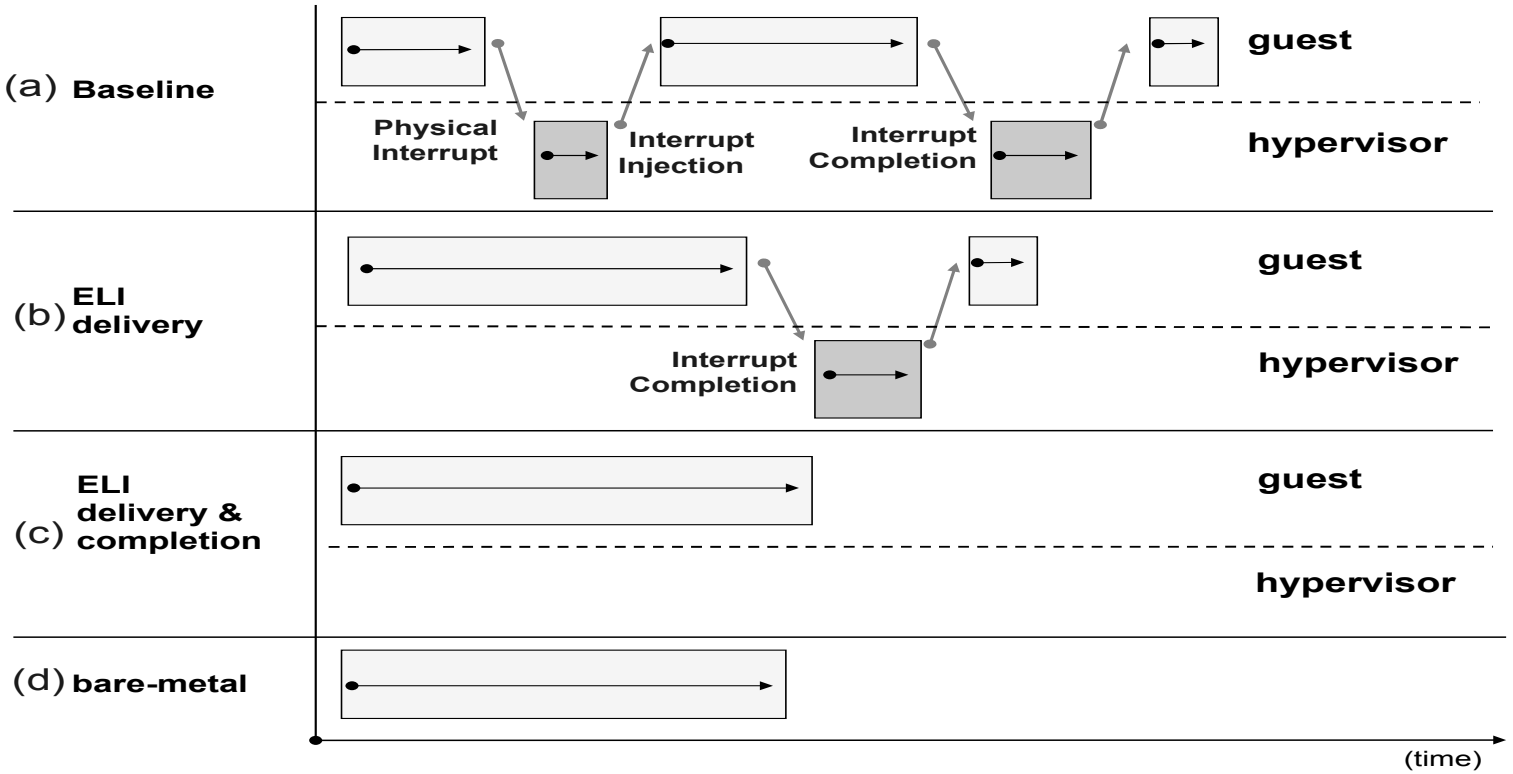


Interrupts cause exits

- Each interrupt causes at least two exits
- One to deliver
- One to signal completion



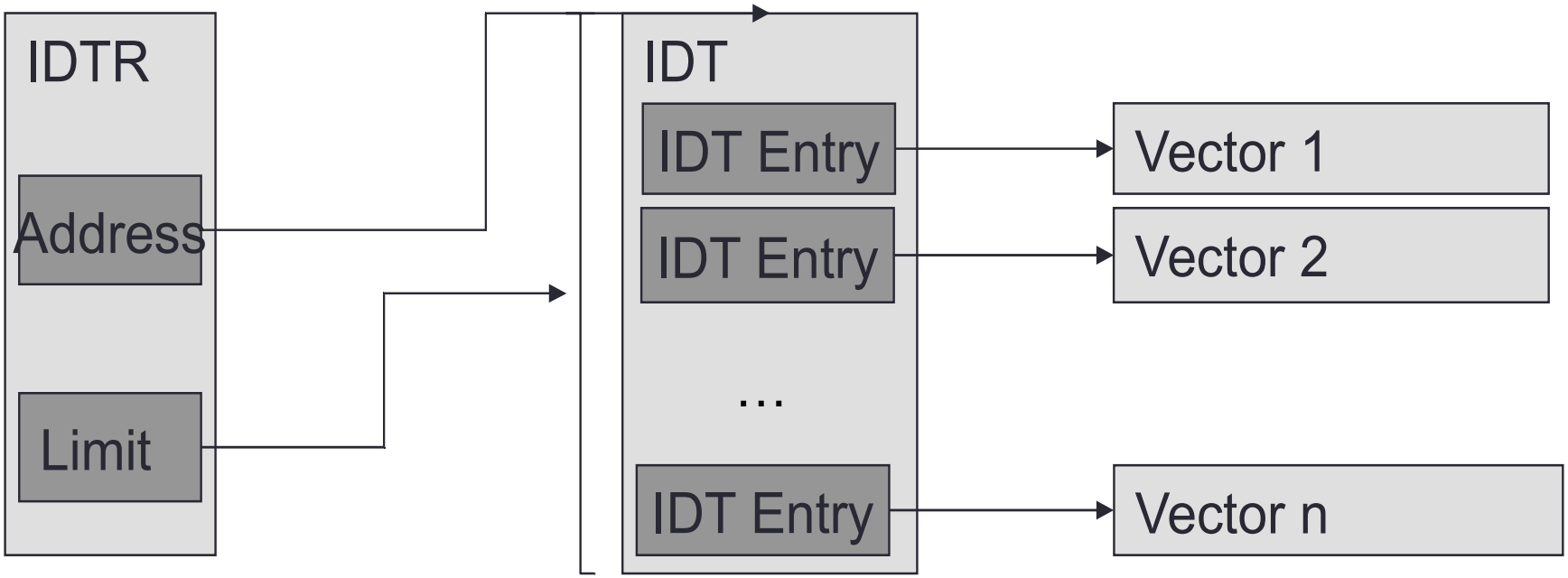
ELI: Exitless Interrupts



ELI: direct interrupts for unmodified, untrusted guests

"ELI: Bare-Metal Performance for I/O Virtualization", Gordon, Amit, Hare'El, Ben-Yehuda, Landau, Schuster, Tsafrir, ASPLOS '12

Background: interrupts



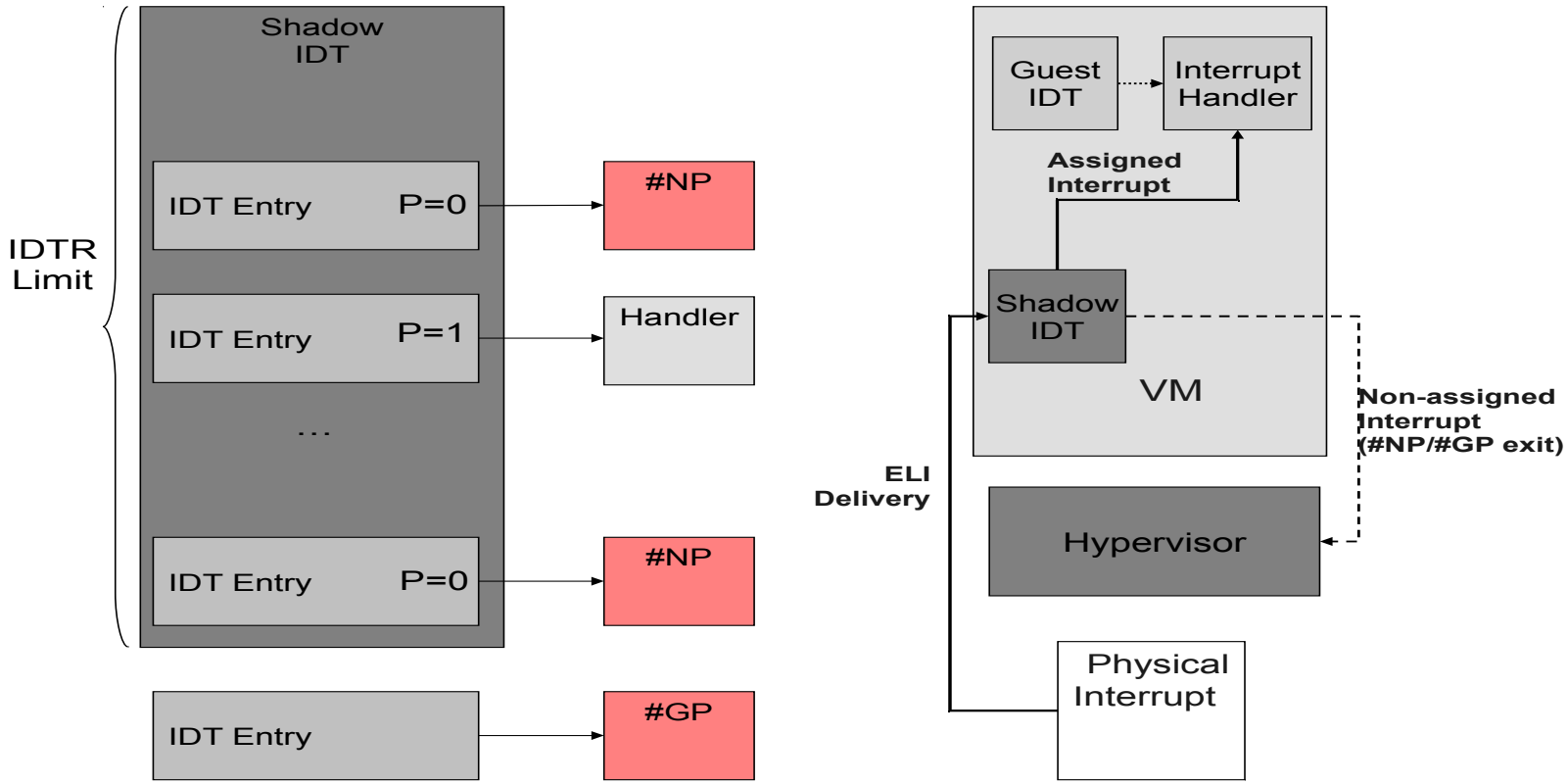
IDT
Register

Interrupt
Descriptor
Table

Interrupt handlers

- I/O devices raise interrupts
- CPU temporarily stops the currently executing code
- CPU jumps to a pre-specified interrupt handler

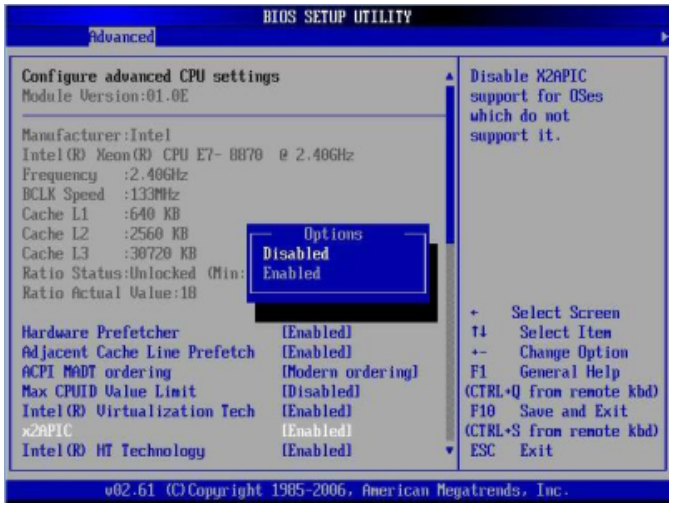
ELI: delivery



- All interrupts are delivered directly to the guest
- Host and other guests' interrupts are bounced back to the host
- . . . without the guest being aware of it

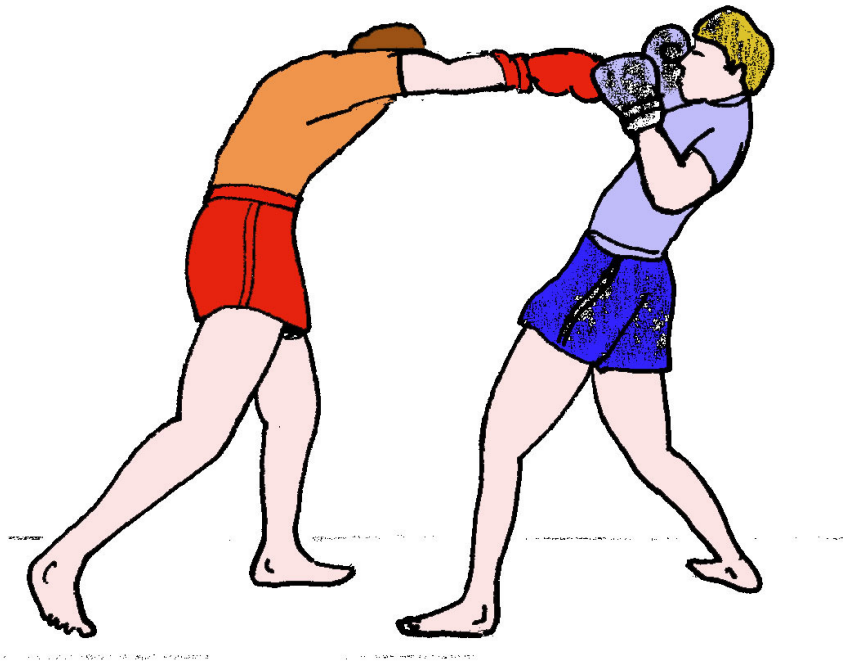
ELI: signaling completion

- Guests signal interrupt completions by writing to the Local Advance Programmable Interrupt Controller (LAPIC) End-of-Interrupt (EOI) register
- Old LAPIC: hypervisor traps load/stores to LAPIC page
- x2APIC: hypervisor can trap specific registers



- Signaling completion without trapping requires x2APIC
- ELI gives the guest direct access only to the EOI register

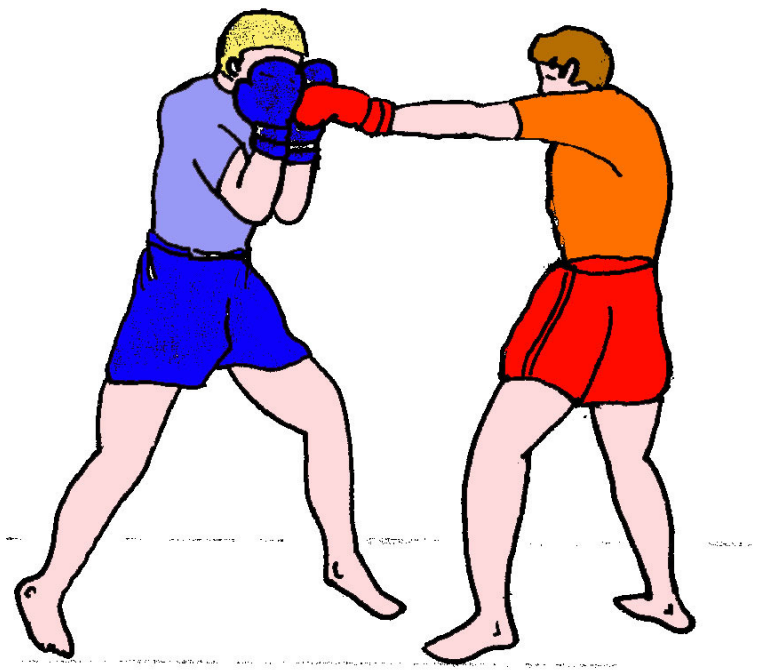
ELI: threat model



Threats: malicious guests might try to:

- keep interrupts disabled
- signal invalid completions
- consume other guests or host interrupts

ELI: protection



- VMX preemption timer to force exits instead of timer interrupts
- Ignore spurious EOIs
- Protect critical interrupts by:
  - Delivering them to a non-ELI core if available
  - Redirecting them as NMIs→unconditional exit
  - Use IDTR limit to force #GP exits on critical interrupts

Evaluation: netperf

102,000 Exits/sec —(+)→ **800**

60% Time in guest —(-)→ **98%**

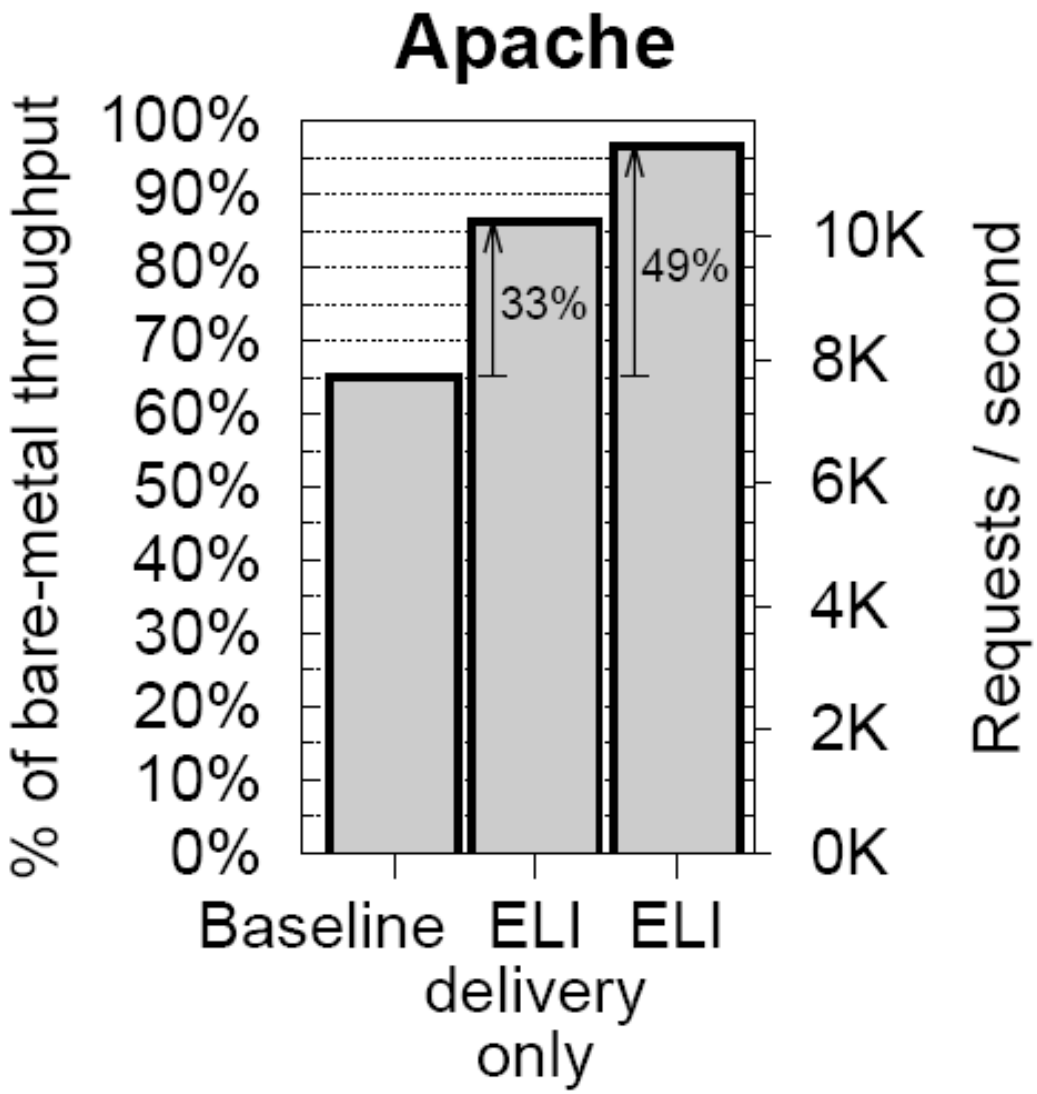
bare-metal performance!

Evaluation: apache

91,000 —(+ / -)→ 1,100 Exits/sec

65% —(- / +)→ 97% Time in guest

bare-metal performance!

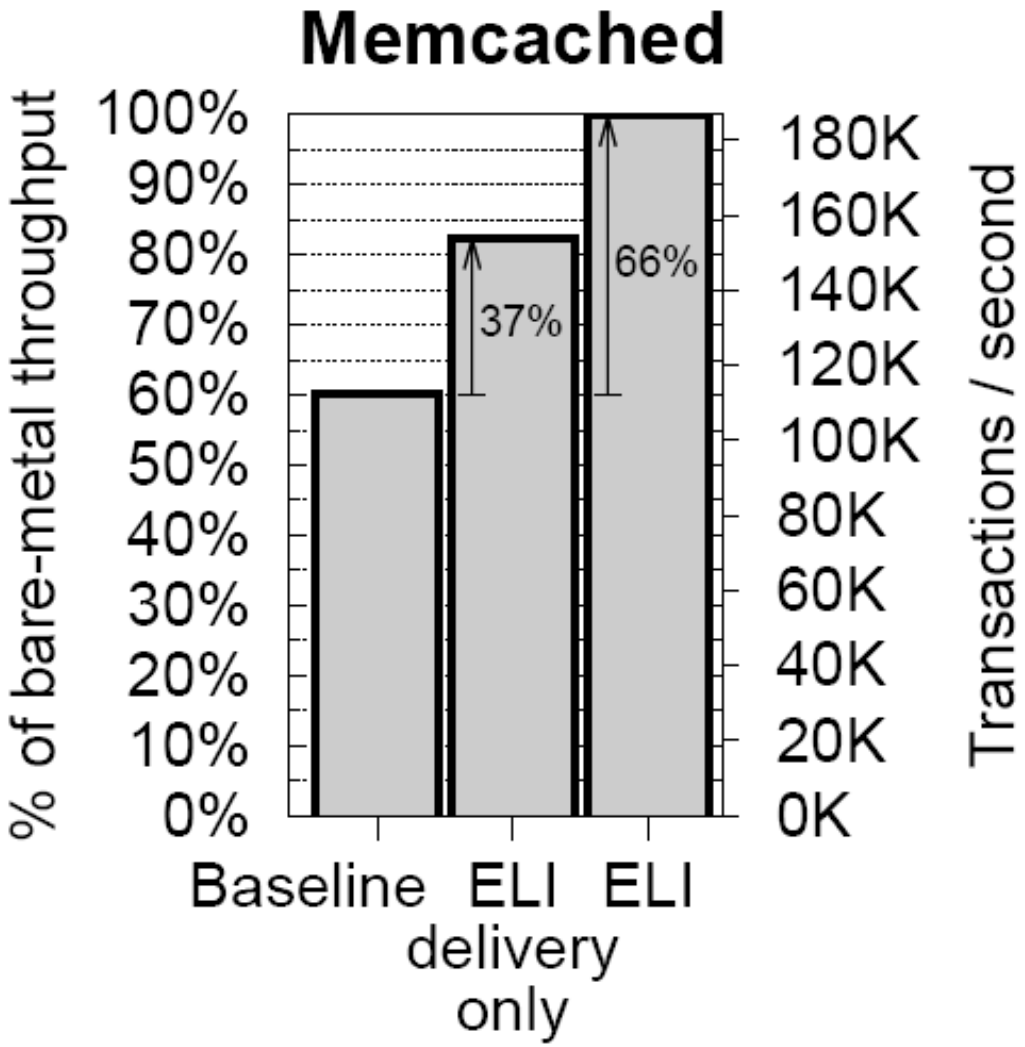


Evaluation: memcached

123,000 —(+ / -)→ 1,000 Exits/sec

60% —(- / +)→ 100% Time in guest

bare-metal performance!



Conclusions: ELI

- Achievement unlocked: bare-metal performance for x86
- . . . if they use device assignment

The saga continues: ELVIS at SYSTOR

Systor 2012

Haifa, Israel June 4 – 6

The 5th Annual International Systems and Storage Conference

In cooperation with ACM, IEEE, USENIX, and TCE 2012

Thank you! Questions?

Hardware wishlist?

- For nested virtualization: "`sie`"-like functionality
- For general virtualization performance: fast inter-core exits, i.e., host-to-guest and guest-to-host inter-core notifications
- For I/O: IOMMU with I/O page faults (PRI)
- In general: better chipset-level visibility. A cycle-accurate simulator for I/O would be really nice. . .