Bare-Metal Performance for x86 I/O Virtualization

Muli Ben-Yehuda

Technion & IBM Research

HiPEAC Autumn Computing Systems Week in Barcelona

Background: x86 machine virtualization

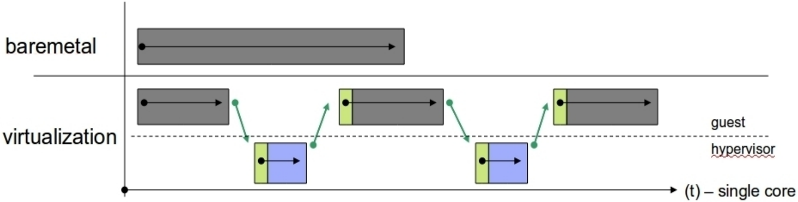
- Running multiple different unmodified operating systems
- Each in an isolated virtual machine
- Simultaneously
- On the x86 architecture
- Many uses: live migration, record & replay, testing, security, ...
- Foundation of IaaS cloud computing
- Used nearly everywhere

The problem is performance

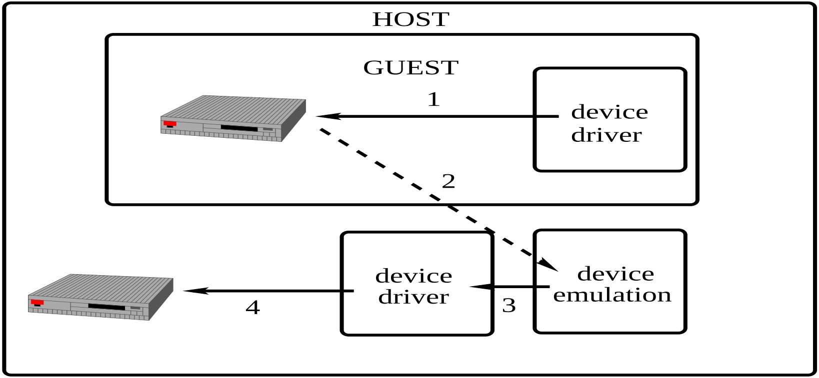
- Machine virtualization can reduce performance by orders of magnitude [Adams06,Santos08,Ram09,Ben-Yehuda10,Amit11,. . .]
- Overhead limits use of virtualization in many scenarios
- We would like to make it possible to use virtualization everywhere
- Where does the overhead come from?

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

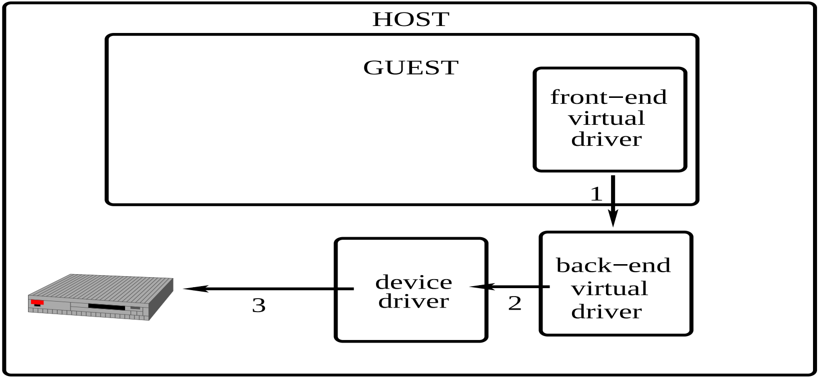


I/O virtualization via device emulation



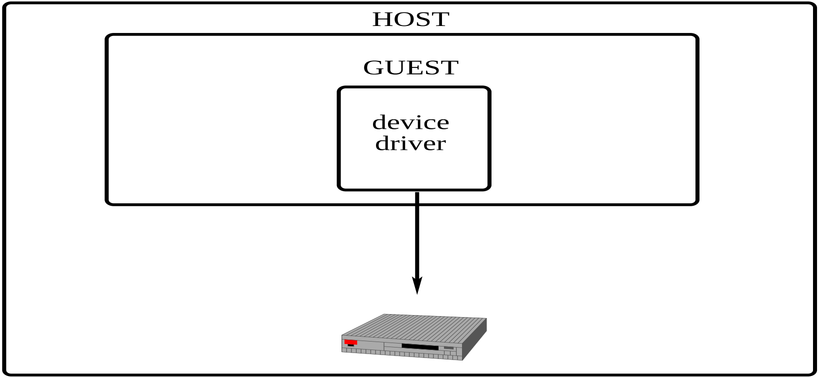
- Emulation is usually the default [Sugerman01]
- Works for unmodified guests out of the box
- Very low performance, due to many exits on the I/O path

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
- Device assignment still 25% worse than bare metal. Why?

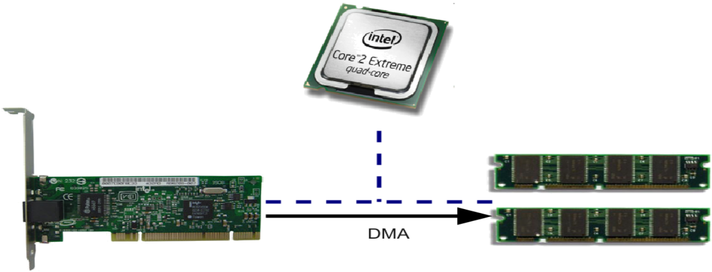
"The Turtles Project: Design and Implementation of Nested Virtualization", Ben-Yehuda, Day, Dubitzky, Factor, Hare'El, Gordon, Liguori, Wasserman and Yassour, OSDI '10

What does it mean, to do I/O?

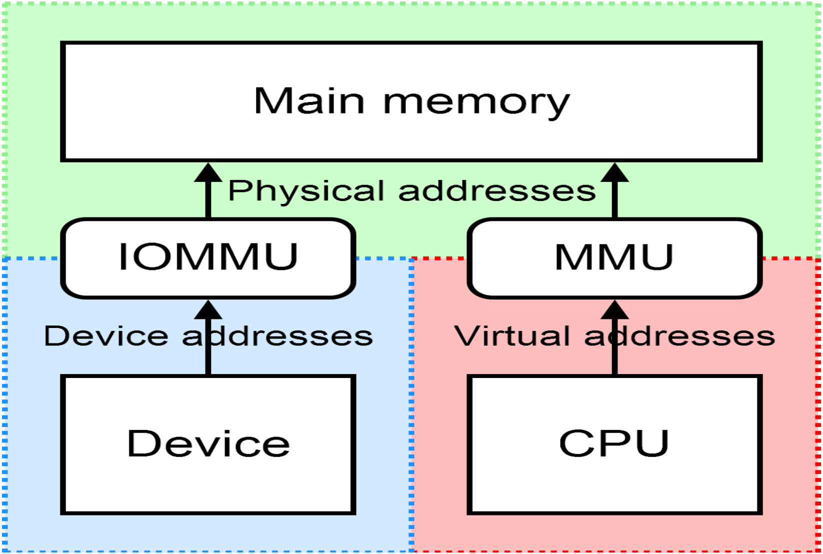
- Programmed I/O (in/out instructions)
- Memory-mapped I/O (loads and stores)
- Direct memory access (DMA)
- Interrupts

Direct memory access (DMA)

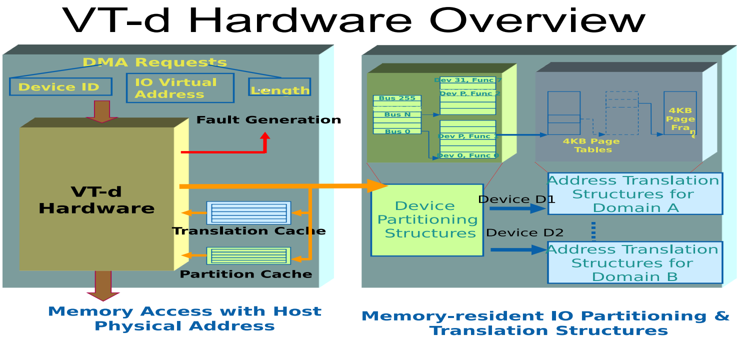
- All modern devices access memory directly
- On bare-metal:
  - A trusted driver gives its device an address
  - Device reads or writes that address
- Protection problem: guest drivers are not trusted
- Translation problem: guest memory $\neq$ host memory
- Direct access: the guest bypasses the host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



IOMMU


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

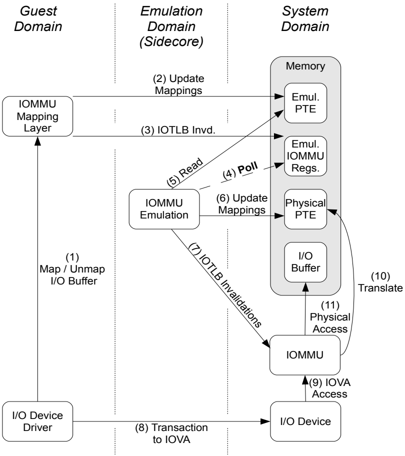
The IOMMU mapping memory/performance tradeoff



- 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

vIOMMU: efficient IOMMU emulation

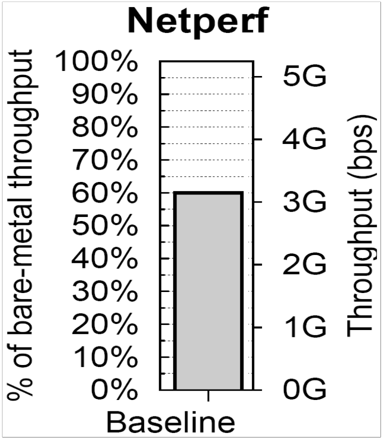
- Emulate an IOMMU so that we know when to map and unmap
- Use a sidecore [Kumar07] for efficient emulation: avoid costly exits by running emulation on another core in parallel
- Optimistic teardown: relax protection to increase performance by caching translation entries
- vIOMMU provides high performance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



"vIOMMU: Efficient IOMMU Emulation", Amit, Ben-Yehuda, Schuster, Tsafrir, USENIX ATC '11

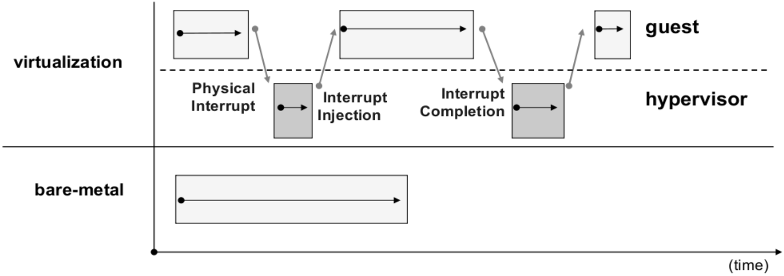
Problem solved?

- 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?

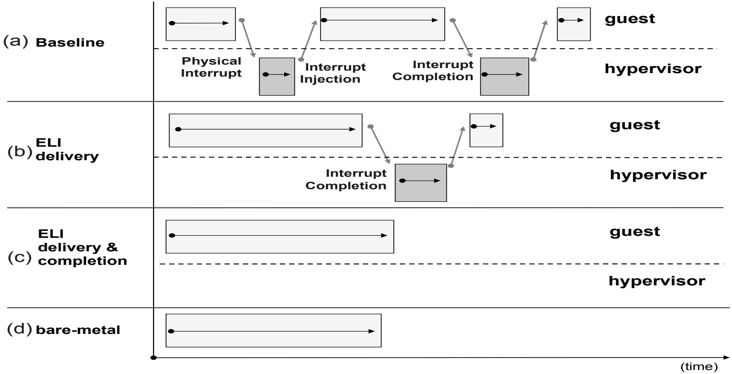


Recap: doing I/O

- Programmed I/O (in/out instructions)
- Memory-mapped I/O (loads and stores)
- Direct memory access (DMA)
- Interrupts: approximately 49,000 interrupts per second with Linux



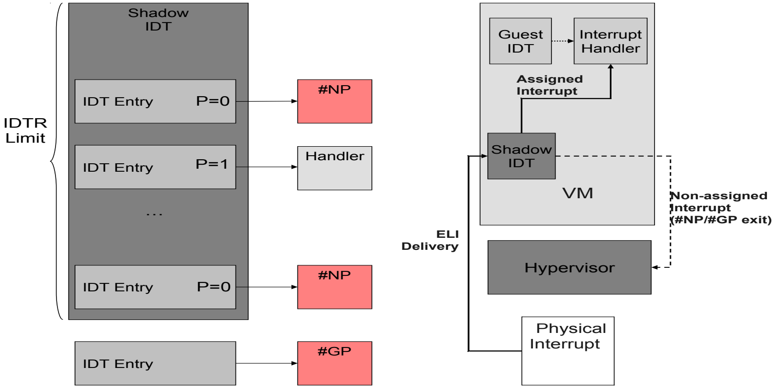
ELI: ExitLess Interrupts



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

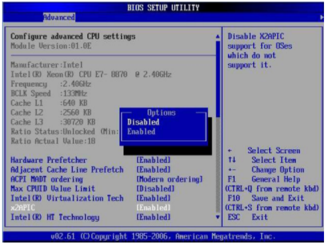
ELI: delivery



- 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

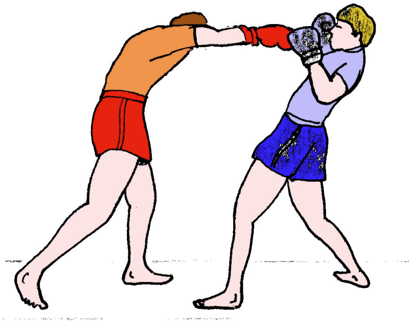
ELI: signaling completion

- 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

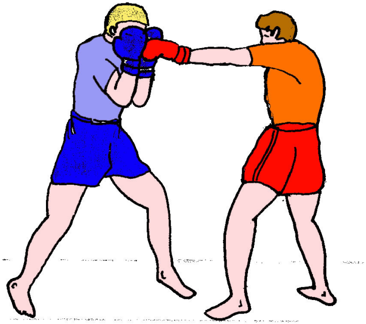
ELI: threat model



Threats: malicious guests might try to:

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

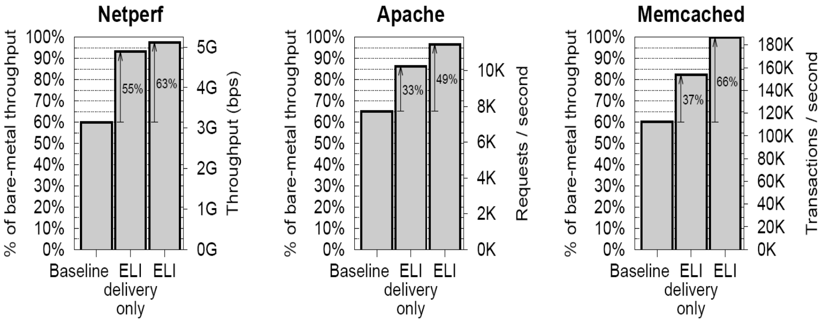
ELI: protection



- 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

IOLanes

Bare-metal Performance for I/O Virtualization



- Throughput is scaled so 100% means bare-metal throughput
- All workloads reach 97–100% of bare metal with ELI!
- CPU is saturated; host uses huge pages to back guest memory
- Full experimental details and analysis in ASPLOS paper

Conclusion

- IOMMUs take the host out of the DMA path
- ELI takes the host out of the interrupt path
- Achievement unlocked: bare-metal performance for x86 VMs

Thank you! Questions?

IOLanes