vRIO: Paravirtual Remote I/O

ASPLOS 2016

Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir

Technion – Israel Institute of Technology
IBM Research – Haifa

http://www.cs.technion.ac.il/~dan/papers/vrio-asplos-2016.pdf

vRIO in a nutshell

- **Interesting optimization for I/O of virtual machines**

How CPUs and I/O devices interact

PHYSICAL I/O

I/O devices

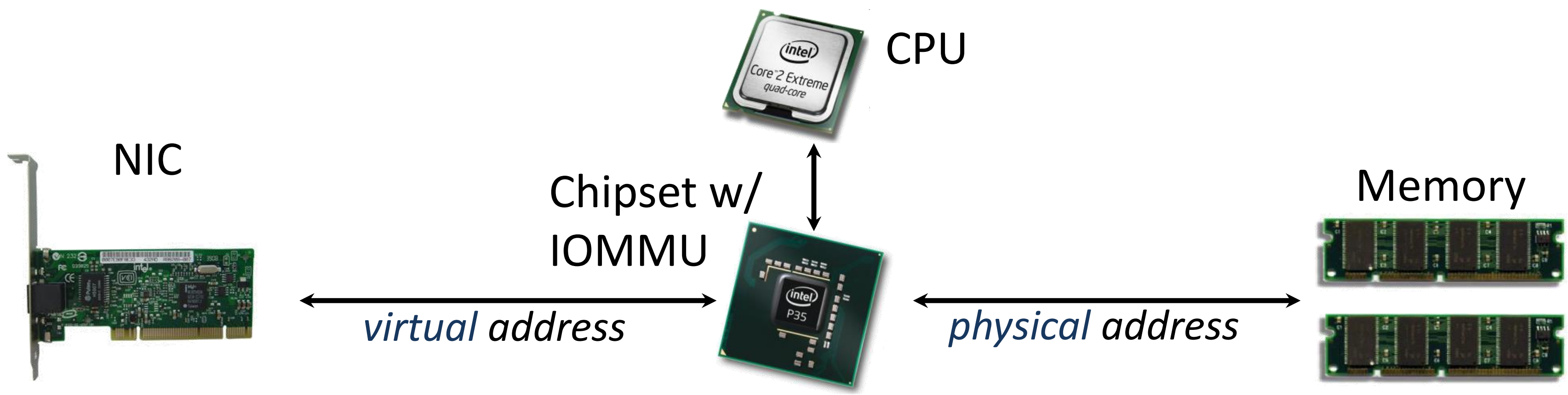
- **Disk, network, keyboard, mouse, GPU, ..**
- **We'll focus on NICs**
  - Network interface controllers
  - PCIe

XL710 Intel Ethernet Converged Network Adapter 10/40 GbE

Exactly four I/O mechanisms

1. **DMA**
2. **MMIO**
3. **Interrupts**
4. **PIO**

NIC updates memory via DMA



DMA = direct memory access (asynchronous)

- CPU asks NIC to do stuff for it (receive & transmit Ethernet packets)
- NIC accesses memory on its own via DMA
- IOMMU is to devices what MMU is to processes

OS talk to NIC via MMIO

- **MMIO = memory mapped I/O**
  - Map device registers to memory

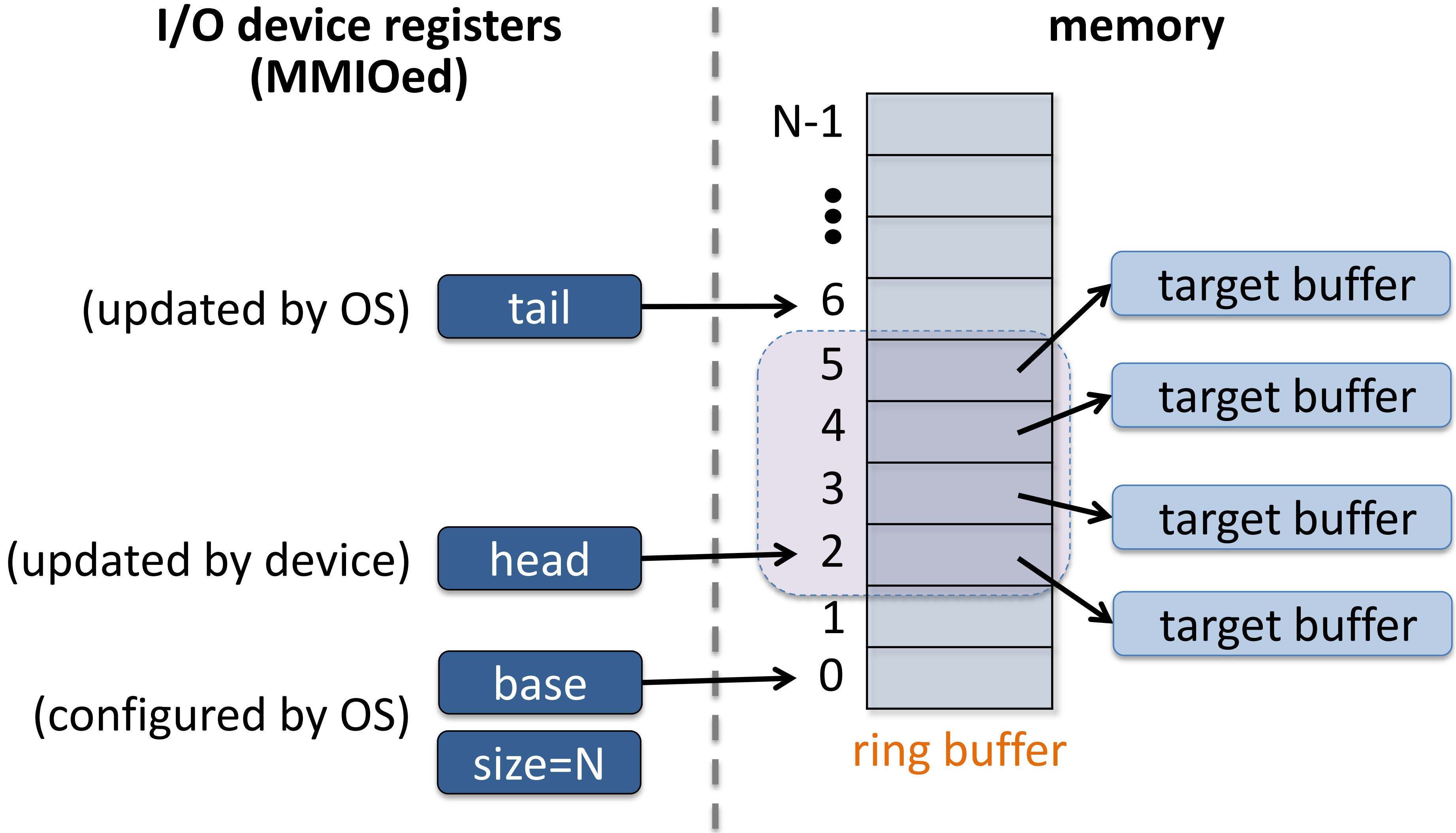
- **Usage**
  - System routes "regular" loads/stores to device
  - Programmers access them like normal memory
  - No need for special instructions

NIC talks to OS via interrupts

- **Can think of interrupt as a form of DMA**
  - NIC “writes” to memory
  - Side-effect
    - Corresponding interrupt handler is invoked (OS code)

- **Different devices are associated with different interrupts**
  - “Vectors”

Shared OS/NIC memory: ring buffer



OS-NIC interaction

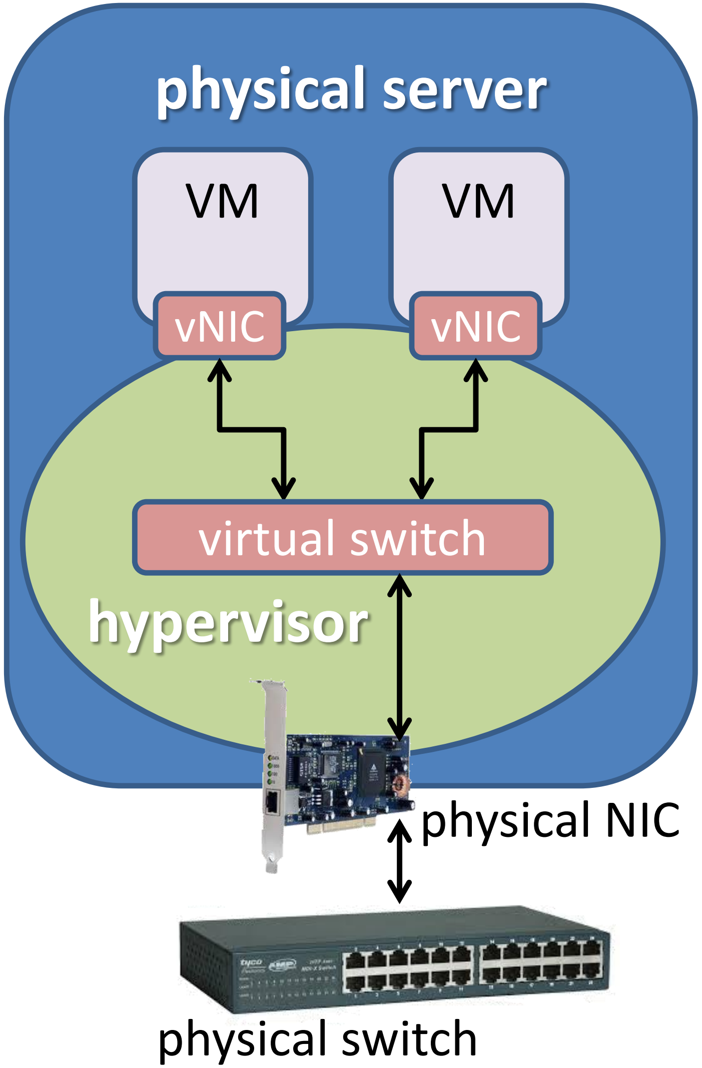
- **Transmit path (“Tx”)**
  - Simplistic OS NIC driver pseudo-code (SW)
    - NIC->registers->start = pointer to beginning of packet
    - NIC->registers->size = size of packet
    - NIC->registers->go = 1 // let NIC know something’s ready
  - NIC pseudo-code (HW)
    - Reads packet *[start, start + size)*
    - Sends packet through wire
    - Trigger interrupt: Tx completed (OS can now free memory buffer)
  - OS NIC interrupt handler (SW)
    - Notify network stack
    - Can know free/reuse memory buffer

For virtual machines, devices aren’t physical

VIRTUAL I/O

Virtual I/O

- **VMs = virtual machines**
  - VM encapsulates OS
- **Need to share HW**
  - With host and other guests
  - So use vNIC instead of NIC
  - Purely SW-based
    - Hypervisor fakes it
- **Virtual I/O is**
  - I/O done through virtual devices
- **Said to be “interposable”**
  - Visible to the host
  - Which can control & manipulate it



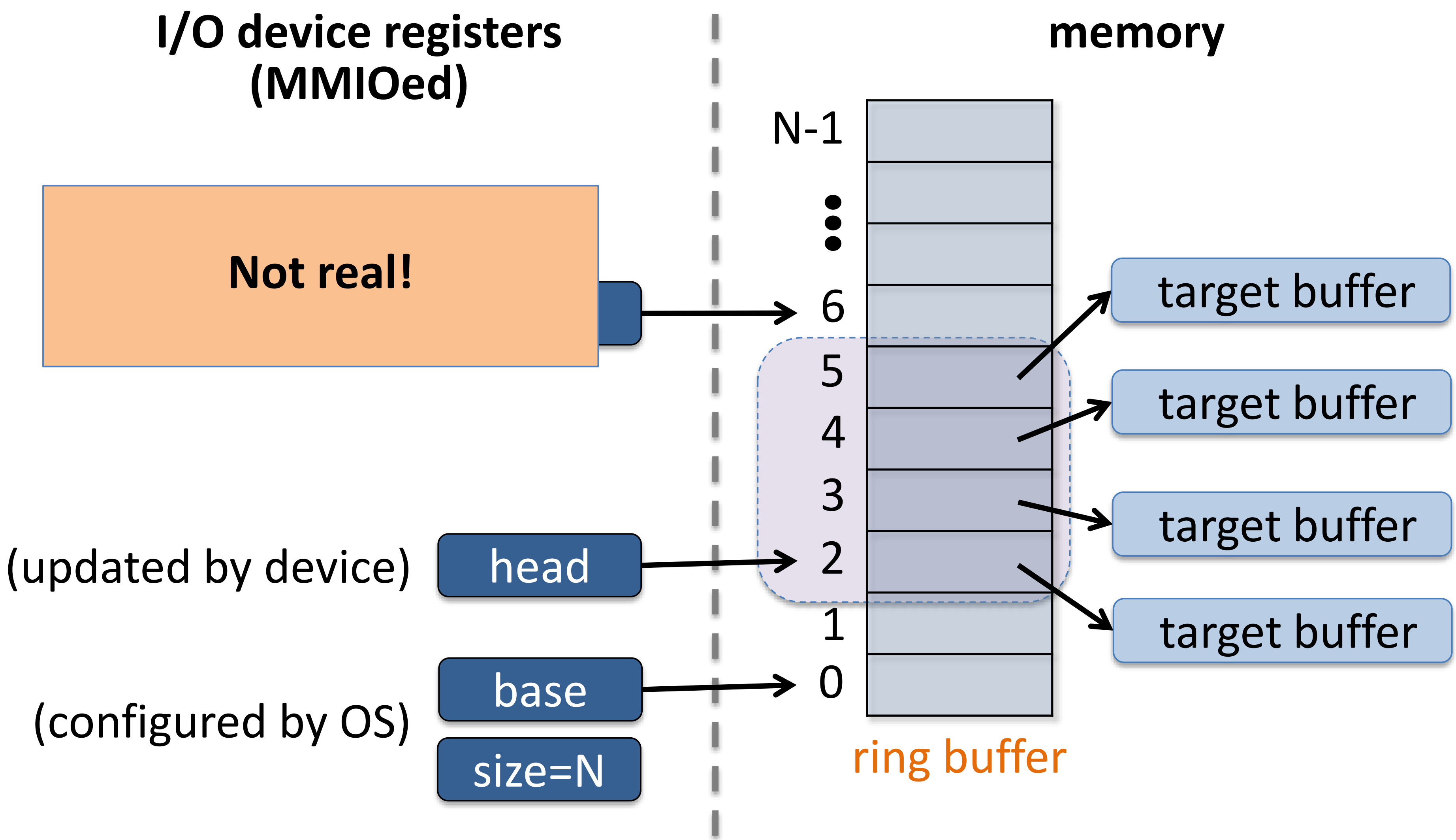
Benefits of I/O interposition

- Multiplexing & improved device utilization
  - Via time/space sharing
- Live migration
  - Use indirection to decouple & recouple VMs from/to physical device
- Suspend/resume
- Seamless switching between I/O channels
- Device aggregation, load balancing, failure masking
- Monitoring & metering
- NVF, SDN
- File-based images
- Replication
- Deduplication
- Snapshots
- Record-replay
- Encryption
- Firewalls
- Deep packet inspection
- Intrusion detection
- …
- It's programmable
  - Everything that can be programmed

Cost of I/O interposition

- Degrades performance
- Here’s why…

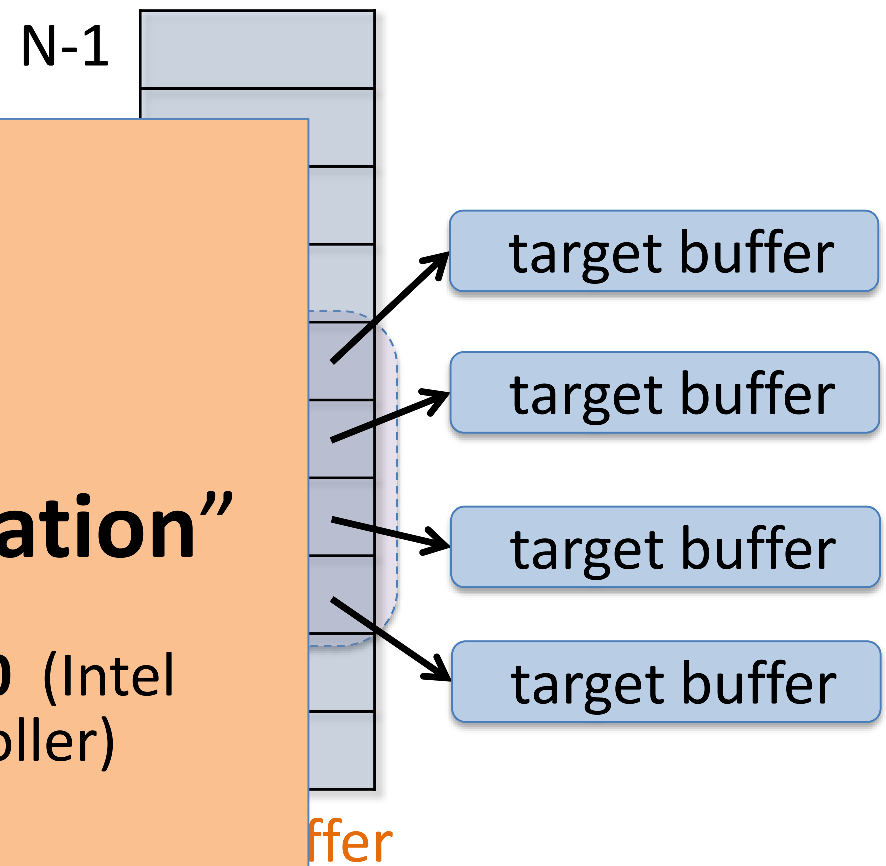
How are vNICs implemented?



How are vNICs implemented?

I/O device registers
~~(MMIOed)~~

memory



Instead of MMIO:
- Use regular memory
- Read/write-protected
- **Trap & emulate**

This technique is called **"emulation"**

De-facto standard for NICs: **e1000** (Intel 82545EM Gigabit Ethernet Controller)

How are vNICs implemented?

- **Transmit path ("Tx")**
  - Simplistic OS NIC driver pseudo-code (SW)
    - NIC->registers->start = pointer to beginning of packet
    - NIC->registers->size = size of packet
    - NIC->registers->go = 1 // let NIC know something's ready
  - NIC pseudo
    - Reads p
    - Sends p
    - Trigger
  - OS NIC inte
    - Notify network stack
    - Can know free/reuse memory buffer

The **problem** with emulation

- Lots of guest-host context switches
- A.k.a. "**exits**"

With e1000, there are 8 per packet!

How are vNICs implemented?

- **Transmit path ("Tx")**
  - Simplistic OS NIC driver pseudo-code (SW)
    - NIC->registers->start = pointer to beginning of packet
    - NIC->registers->size = size of packet
    - NIC->registers->go = 1 // let NIC know something's ready
  - NIC pseudo
    - Reads p
    - Sends p
    - Trigger
  - OS NIC inte
    - Notify r
    - Can kno

Alleviating the problem

- Since vNICs are implemented in SW
- We can invent our own NIC, which minimizes exits

Called "**paravirtualization**"

- Make **guest aware** it's being virtualized
- In this case, by installing a driver

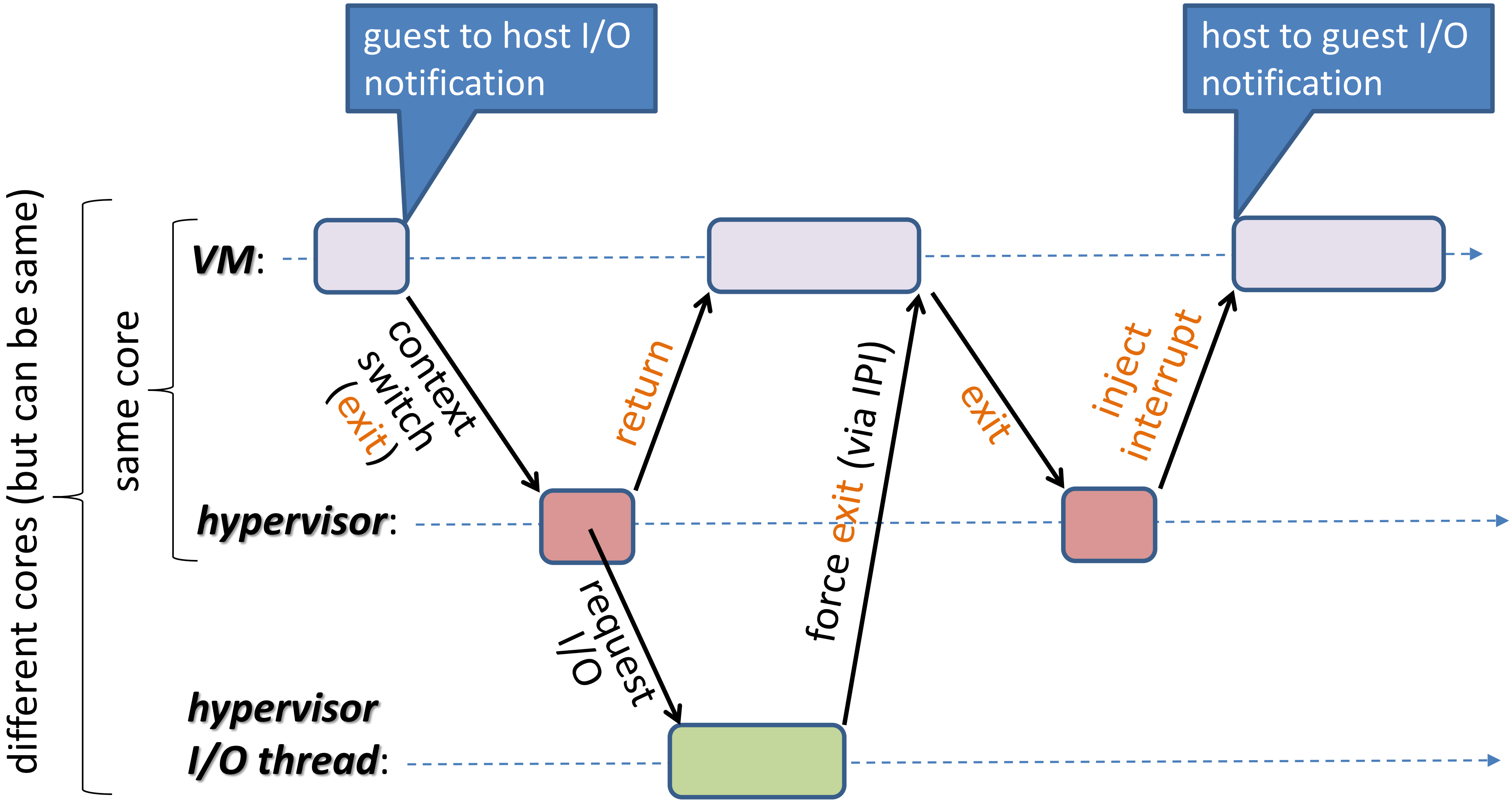
How are vNICs implemented?

- **Transmit path ("Tx")**
  - Simplistic OS NIC driver pseudo-code (SW)
    - NIC->registers->start = pointer to beginning of packet
    - NIC->registers->size = size of packet
    - NIC->registers->go = 1 // let NIC know something's ready
  - NIC pseudo
    - Reads p
    - Sends p
    - Trigger
  - OS NIC inte
    - Notify network stack
    - Can know free/reuse memory buffer

Popular, but host-specific

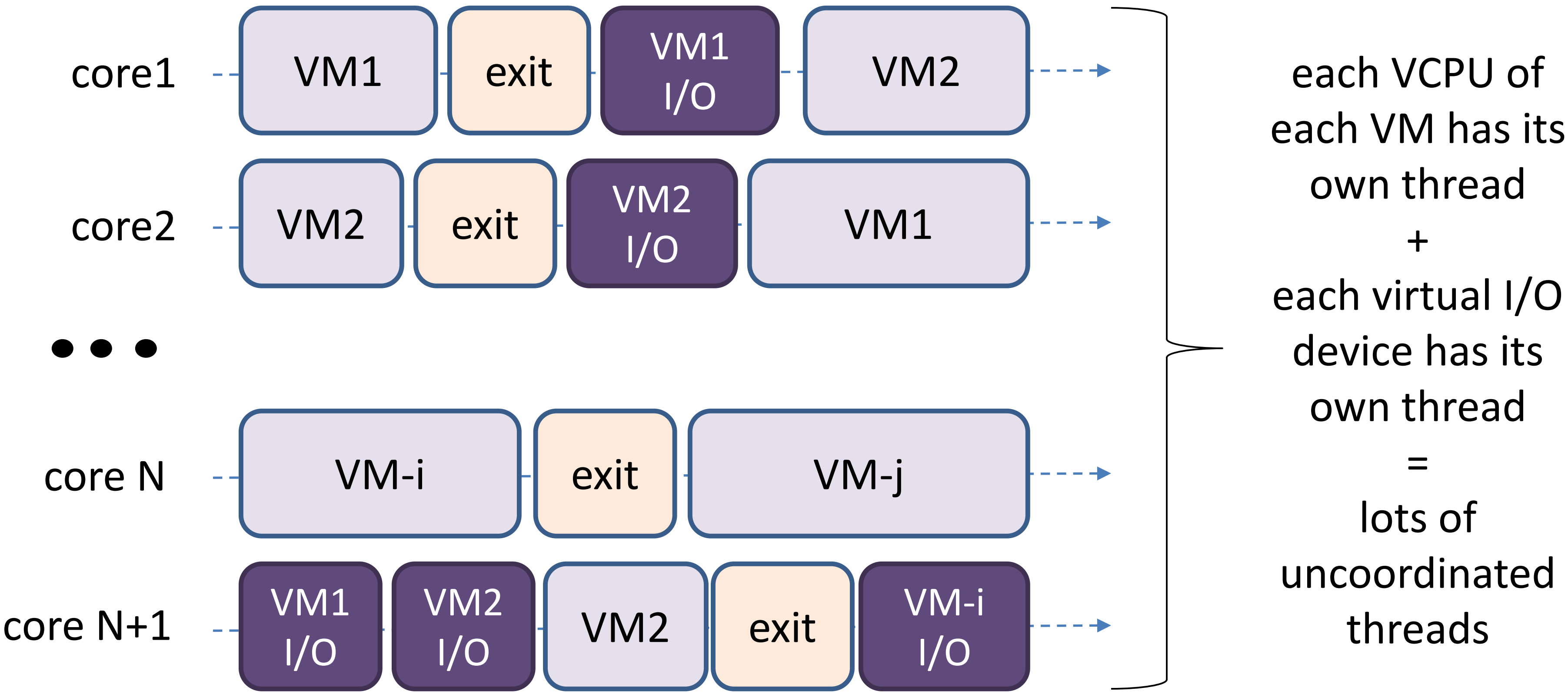
hypervisor	paravirtual driver
KVM	virtio
XEN	PV
VMware ESX	VXNET

Even with paravirtualization



lots of exits (& cache pollution) => poor performance

Even with paravirtualization



unaware host scheduling => poor performance

Virtual I/O models thus far

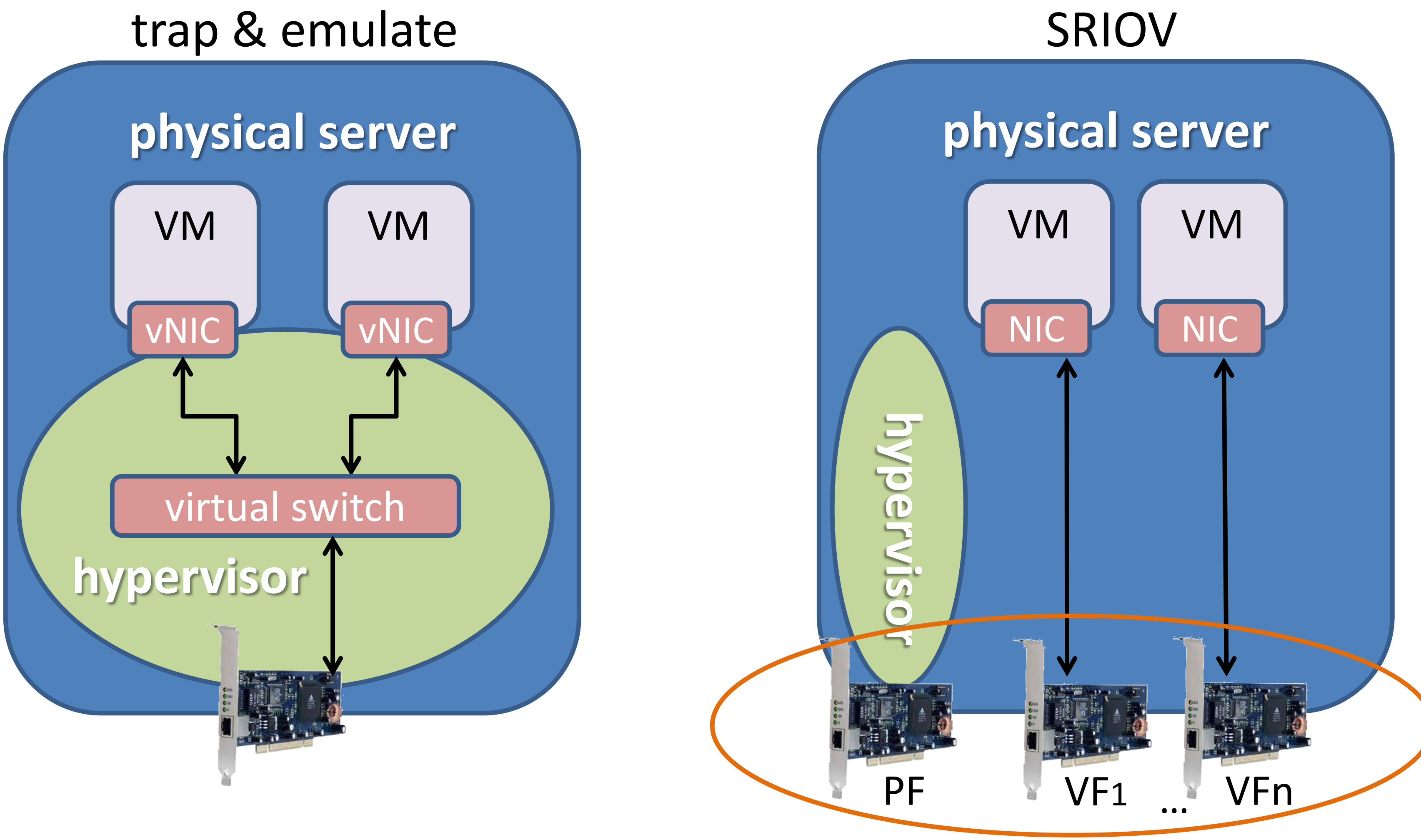
1. **Emulation**
   – Portable
   – Lots of exits
   – Scheduling mess
2. **Paravirtualization**
   – Non-portable
   – Fewer exists, but still
   – Scheduling mess

Hardware to the rescue

1. **Emulation**
   - Portable
   - Lots of exits
   - Scheduling mess
2. **Paravirtualization**
   - Non-portable
   - Fewer exists, but still
   - Scheduling mess
3. **SRIOV (+IOMMU)**
   - Single-Root I/O Virtualization PCIe standard
   - Supports virtual I/O in HW
   - A.k.a.: pass-through I/O, self-virtualizing devices, device assignment

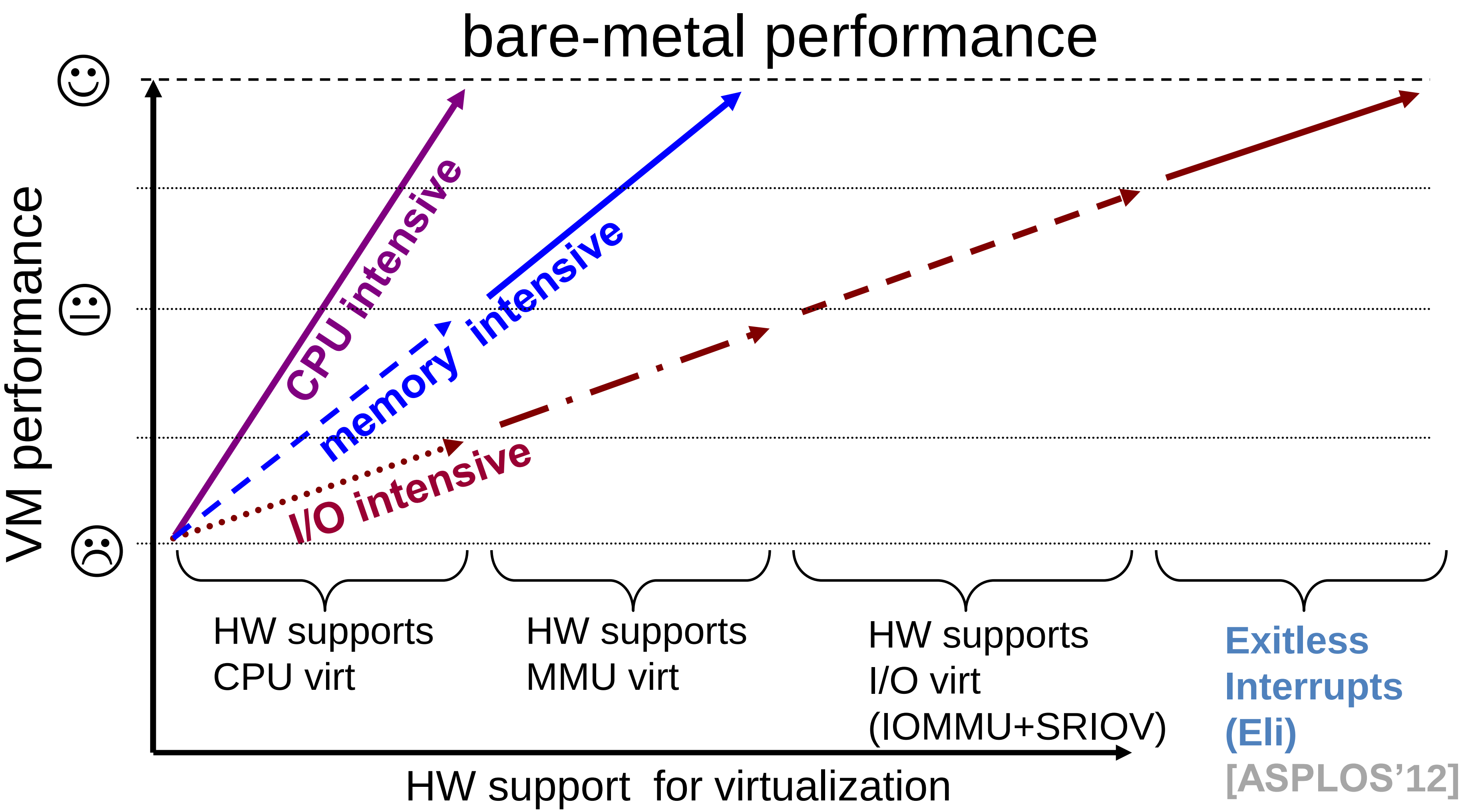
trap & emulate

SW vs. HW virtual I/O



a single physical NIC presents multiple "virtual" instances of itself to SW

x86 virtualization performance



So now that we have SRIOV+Eli

- **Are we done?**

~~Benefits of I/O interposition~~

- Multiplexing & improved device utilization
  - Via time/space sharing
- Live migration
  - Use indirection to decouple & recouple VMs from/to physical device
- Suspend/resume
- Seamless switching between I/O channels
- Device aggregation, load balancing, failure masking
- Monitoring & metering
- NVF, SDN
- File-based images
- Replication
- Deduplication
- Snapshots
- Record-replay
- Encryption
- Firewalls
- Deep packet inspection
- Intrusion detection
- …
- **It's programmable**
  - Everything that can be programmed

Additional SRIOV difficulties

- **DMA assumes memory is there**
  - IOMMUs don’t know how to handle (I/O) page faults
    - What to do when NIC receives data and has no where to put it?
- **Consequently, no memory overcommitment**
  - If VM is given an SRIOV instance
  - Must pin its entire image to the physical memory

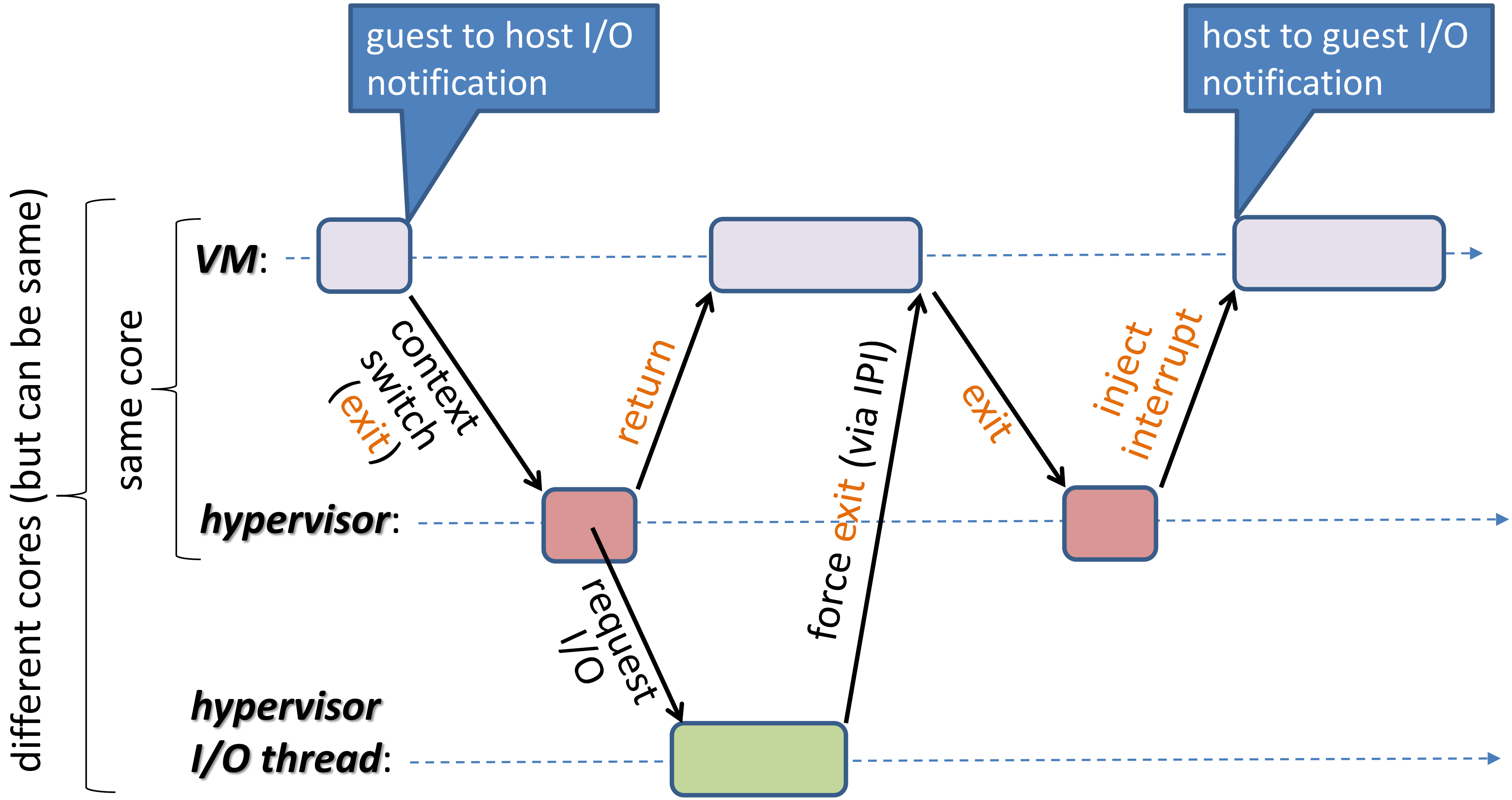
Virtual I/O models thus far

1. **Emulation**
   – Lots of exits
2. **Paravirtualization**
   – Fewer exists, but still
   – Scheduling mess
3. **SRIOV + Eli (= “optimum”)**
   – Single-Root I/O Virtualization PCIe standard
   – Supports virtual I/O in HW

The “sidecore” paradigm

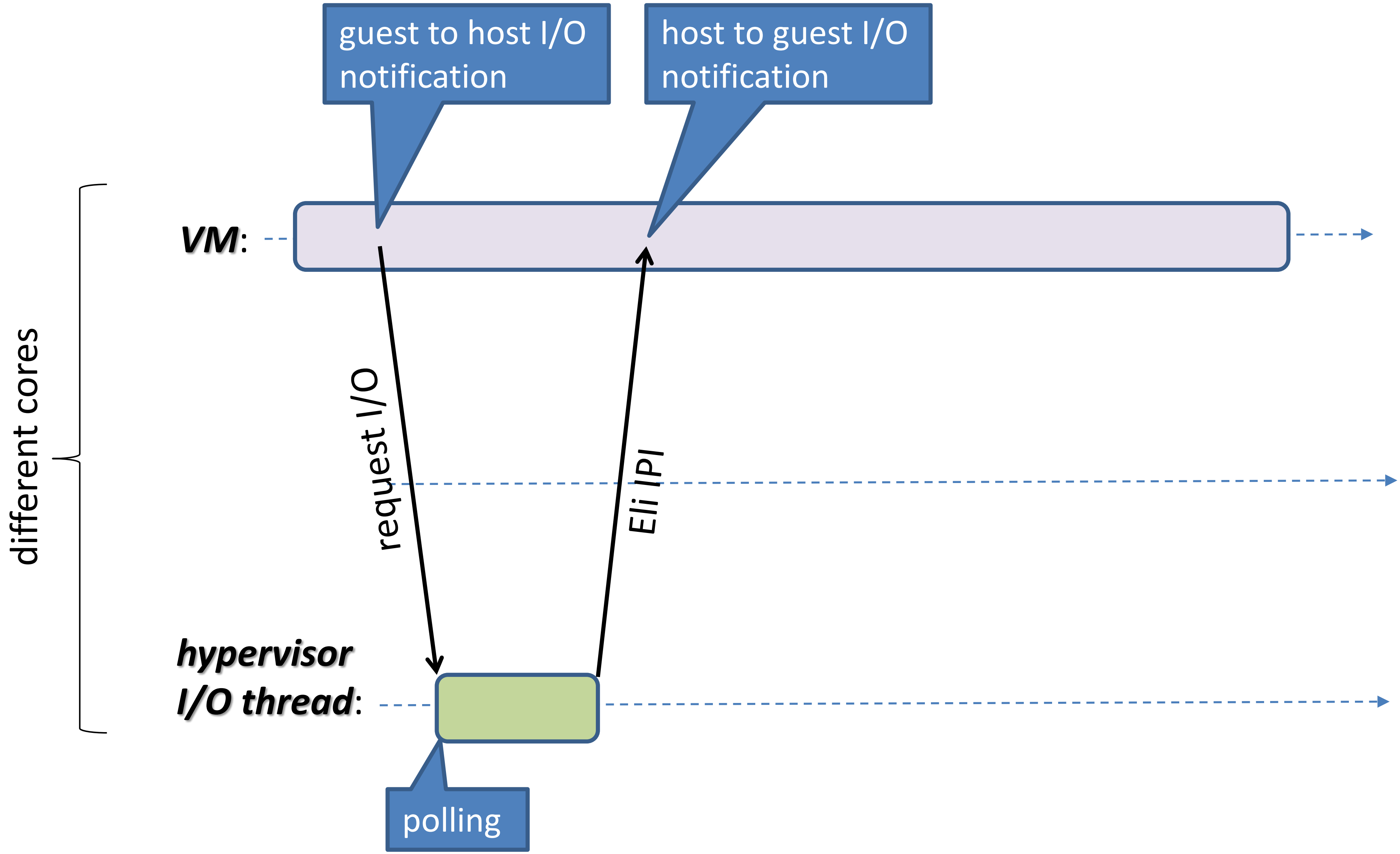
1. **Emulation**
   – Lots of exits
2. **Paravirtualization (= baseline)**
   – Fewer exists, but still
   – Scheduling mess
3. **SRIOV + Eli (= non-interposable optimum)**
   – Single-Root I/O Virtualization PCIe standard
   – Supports virtual I/O in HW
4. **Sidecores**
   – Paravirtual model
   – But instead of taking exits on the VM core
   – Dedicate another host (side)core to poll the relevant memory
   – Other direction (sidecore => VM): IPI with Eli
   – **Sidecore + Eli = “Elvis”** [ATC’13]

Trap-and-emulate baseline

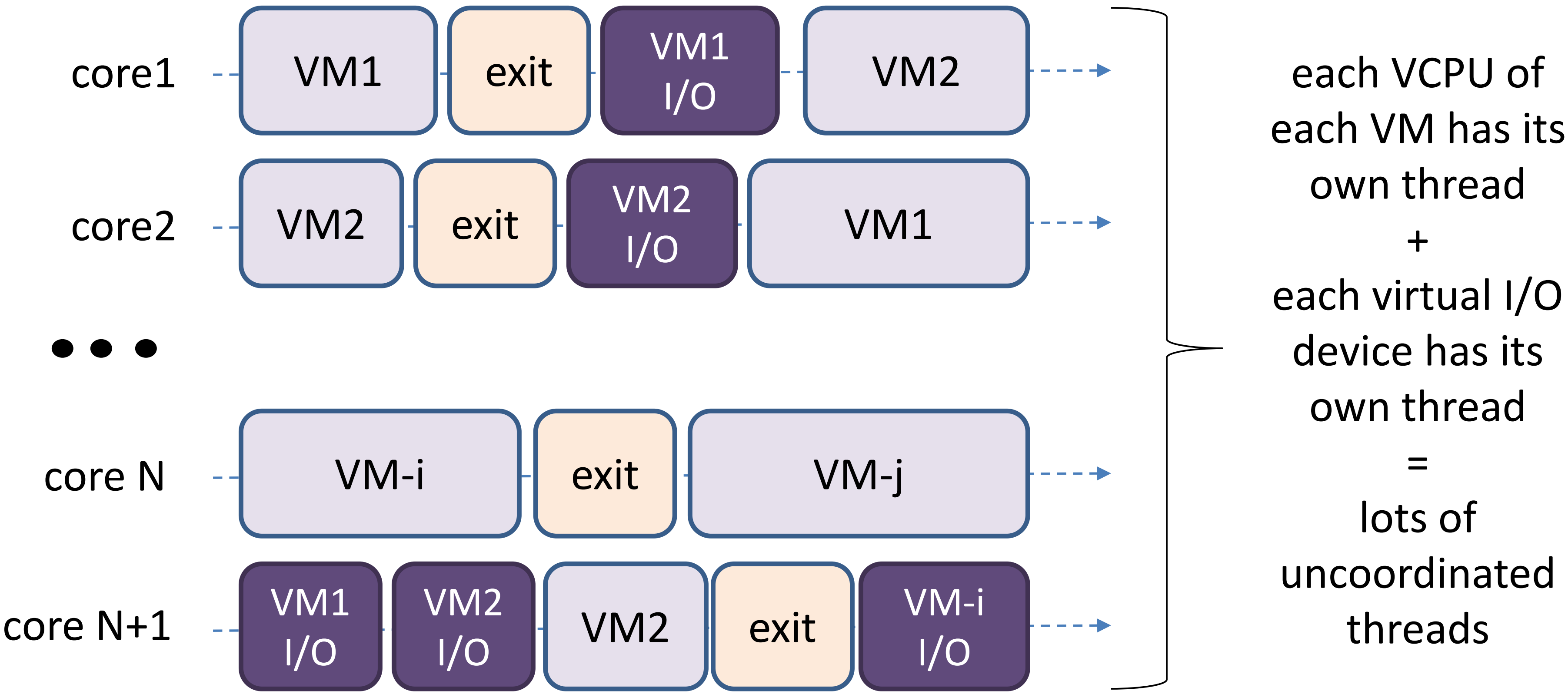


lots of exits (& cache pollution) => poor performance

Elvis sidecore

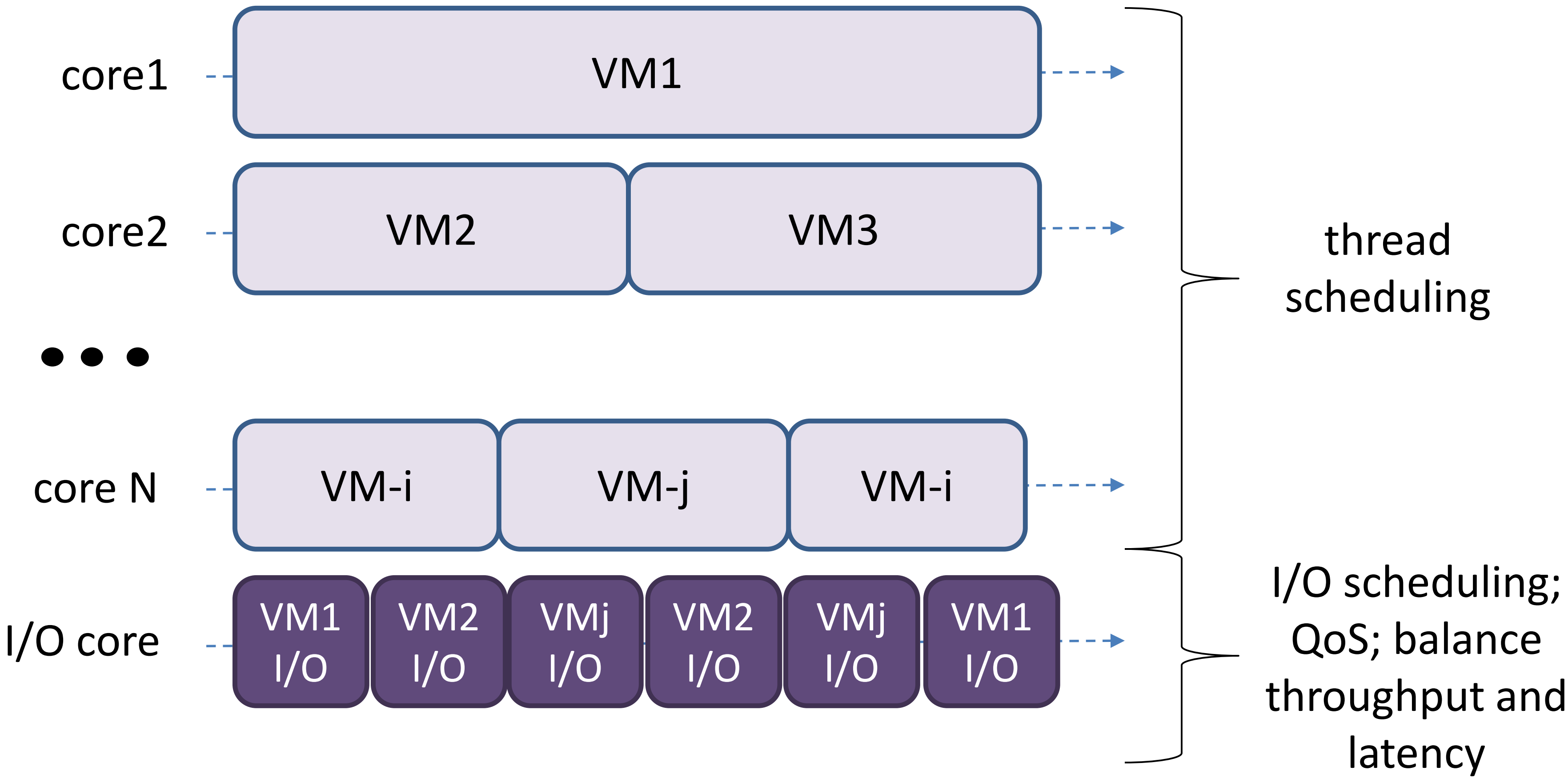


Trap-and-emulate baseline

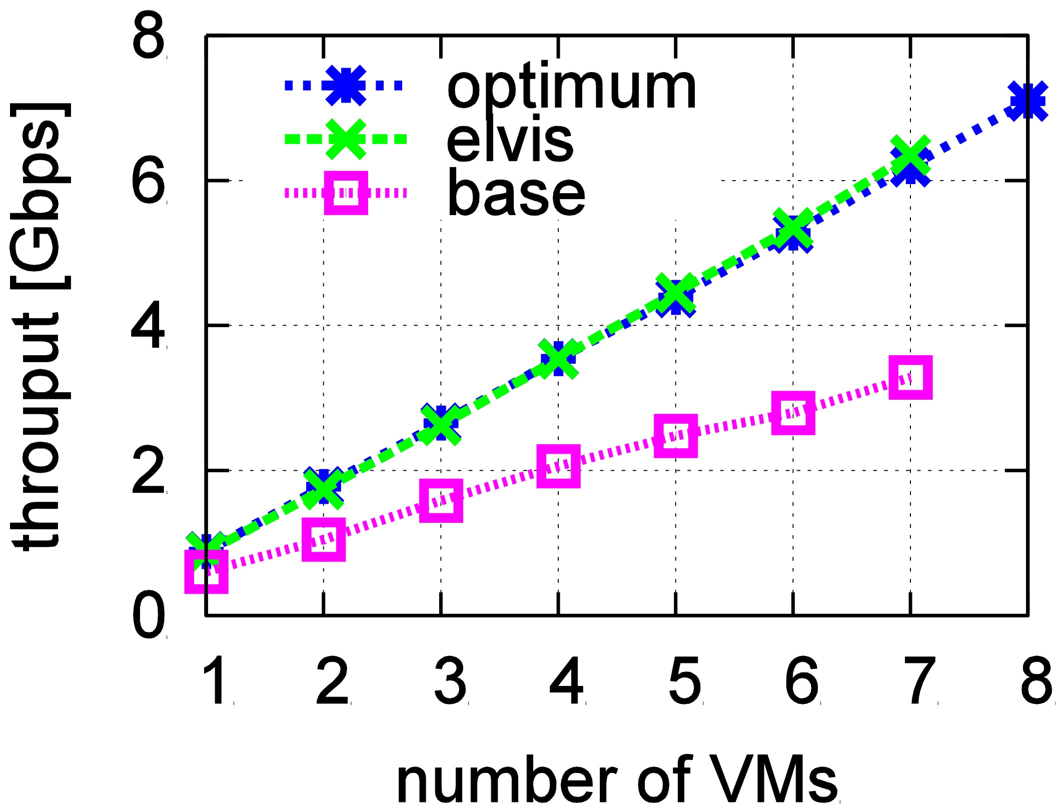


unaware host scheduling => poor performance

Elvis sidecore



Netperf TCP stream (64B msg size)

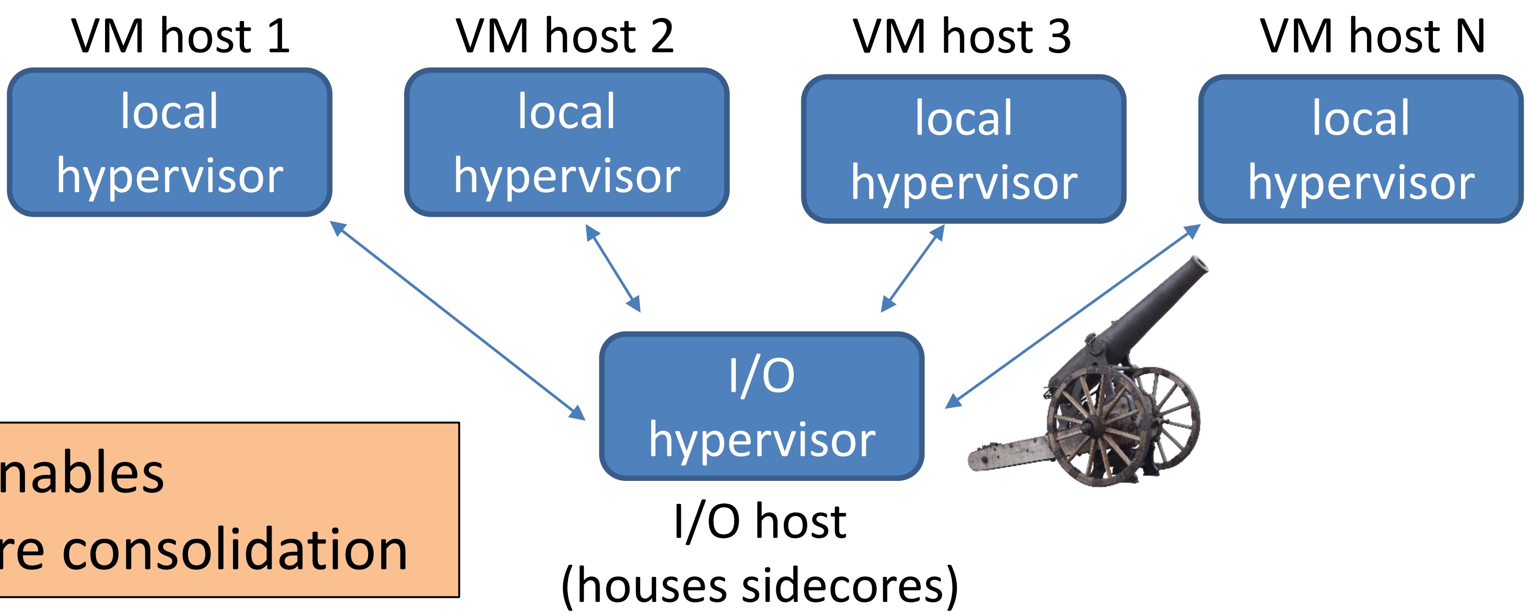


BUT how many sidecores?

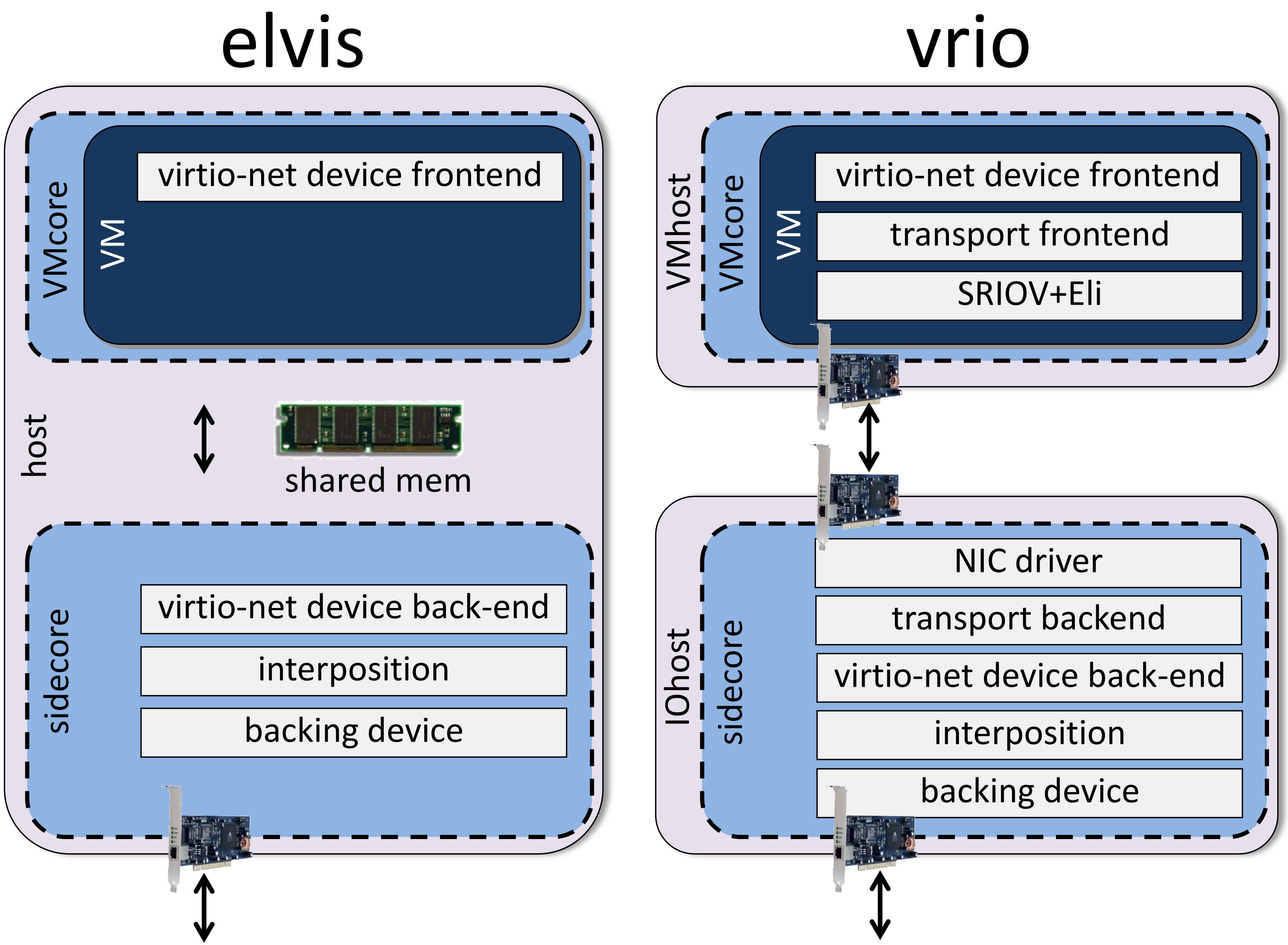
- **Elvis [ATC'13]:**
  - 1-sidecore per 8-core CPU
- **If you take a closer look, however…**
  - It depends on the I/O activity generated by the VMs on the host
  - Can easily require 4 of the 8 cores (half!)
- **What shall well do, then?**
  - Reserve max sidecores
    - And leave them idle until they are needed? ☹
  - Reserve min sidecores
    - And live-migrate away from host when I/O processing exceeds what's available? ☹

BUT how many sidecores?

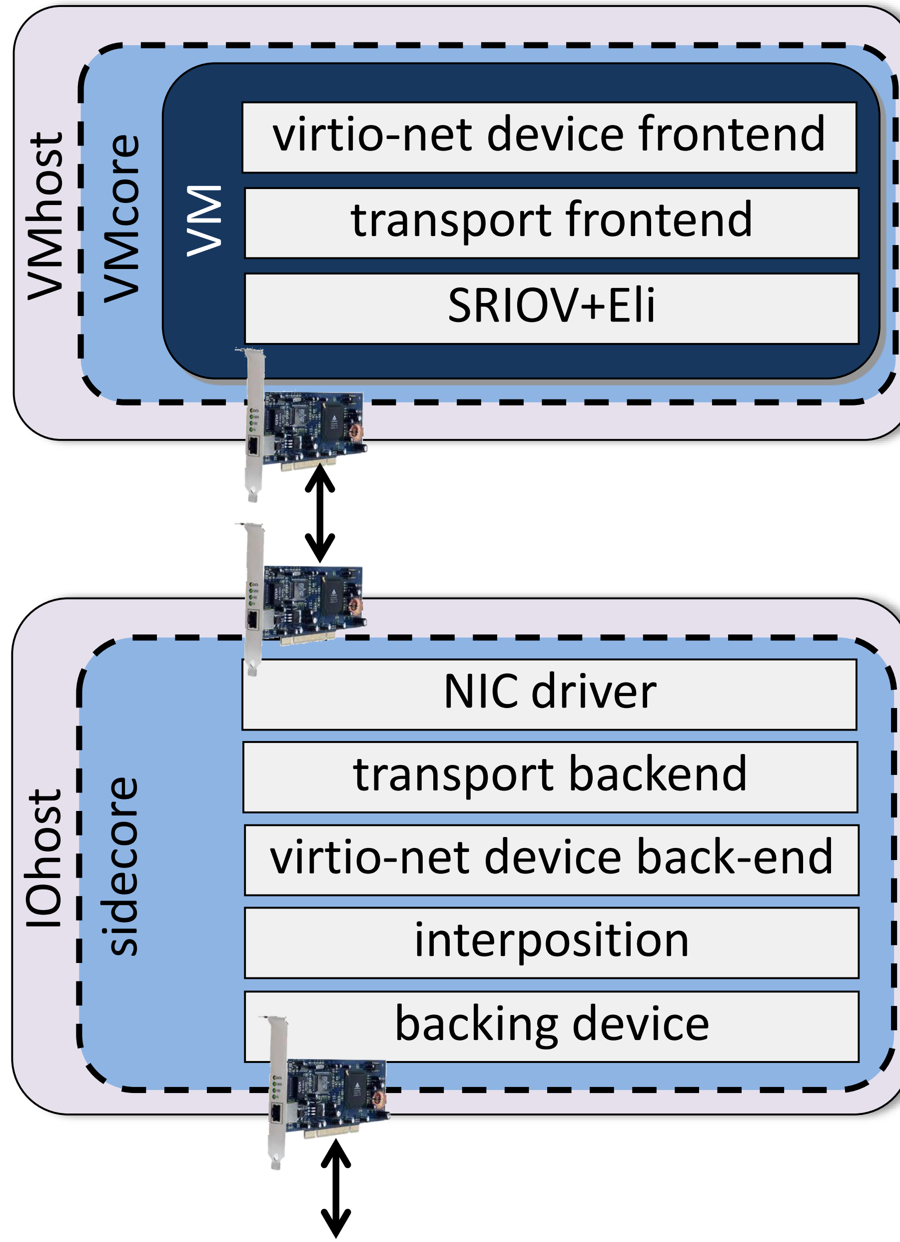
- **If only there was a way to**
  - Treat all sidecores, across all hosts in a rack as belonging to one pool
  - Such that idle sidecores can help busy hosts
- **In other words, if only there was a way to do**
  - "Sidecore consolidation",



vRIO enables sidecore consolidation


elvis
vrio
virtio-net device frontend
VM
VMcore
host
shared mem
sidecore
virtio-net device back-end
interposition
backing device
virtio-net device frontend
transport frontend
SRIOV+Eli
VM
VMcore
VMhost
NIC driver
transport backend
virtio-net device back-end
interposition
backing device
IOhost
sidecore

vrio



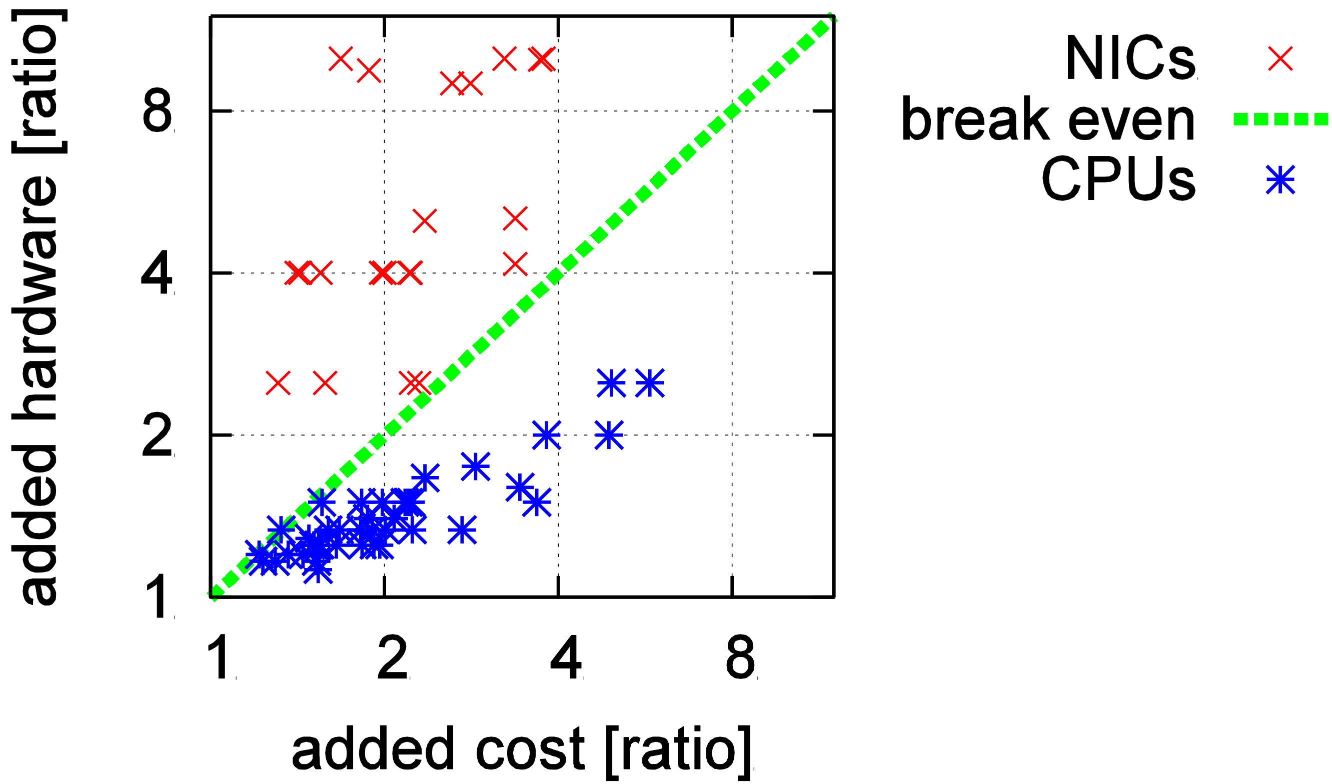
Challenges:

1. **Latency** – added a hop
2. **Cost** – added network HW (IOhost needs *double* the BW of its VMhosts)

Tradeoffs made possible with vrio

- **Use same number of sidecores as elvis across rack**
  - Get better performance (imbalance)
- **Use less sidecores across the rack**
  - Get comparable performance
- **Inherently, rests on assumption underlying virtualization**
  - Not everybody's using *all* their resources *all* the time
  - Which is exactly why we can get by with less physical HW
- **Let's explore that**
  - Assume, for the sake of the argument, that we can get by with ½ of the sidecores
  - Why this might be advantageous, cost wise?

Cost/benefit of upgrading



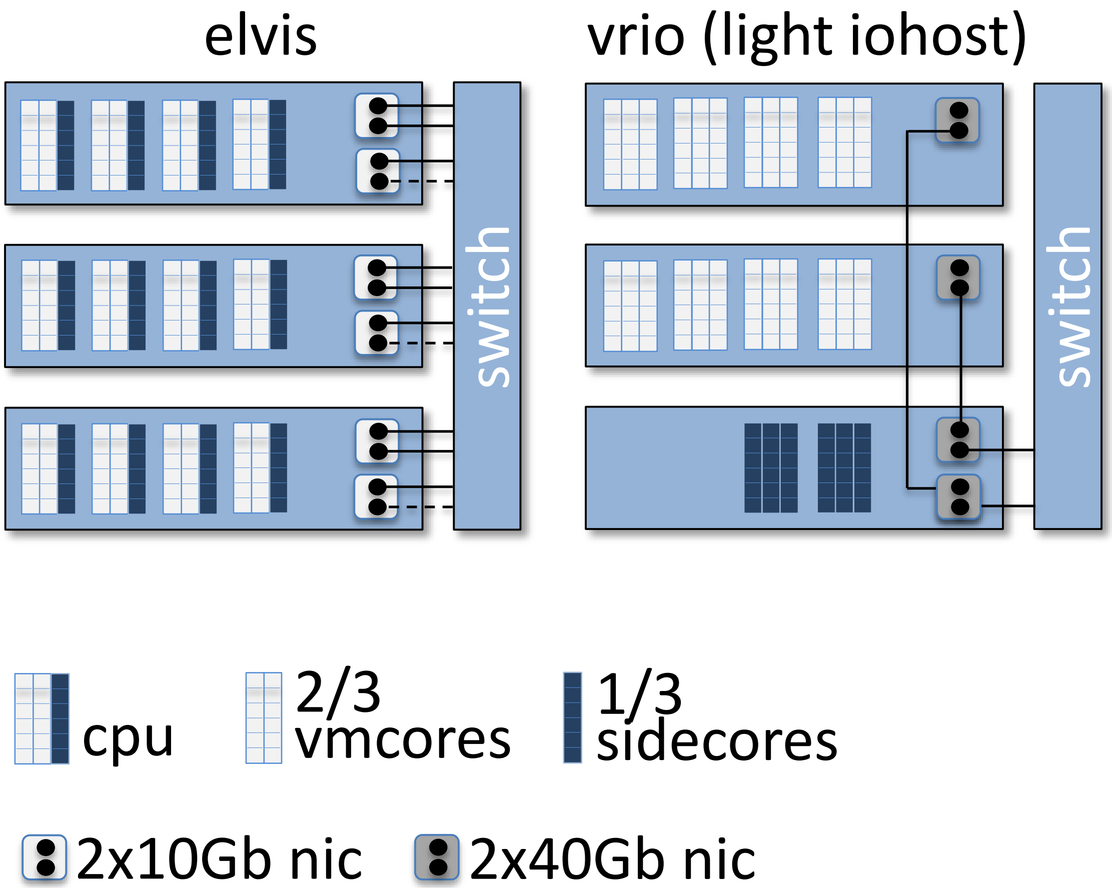
c_1:	\$3,059 12-core 2.3GHz E7-8850 v2 24MB 105W 7.2GT/s QPI 22nm
c_2:	\$4,616 15-core 2.3GHz E7-8870 v2 30MB 130W 8.0GT/s QPI 22nm

$$x = \frac{\$4,616}{\$3,059} \approx 1.51 \text{ and } y = \frac{\breve{15}\text{cores}}{12\text{cores}} = 1.25.$$

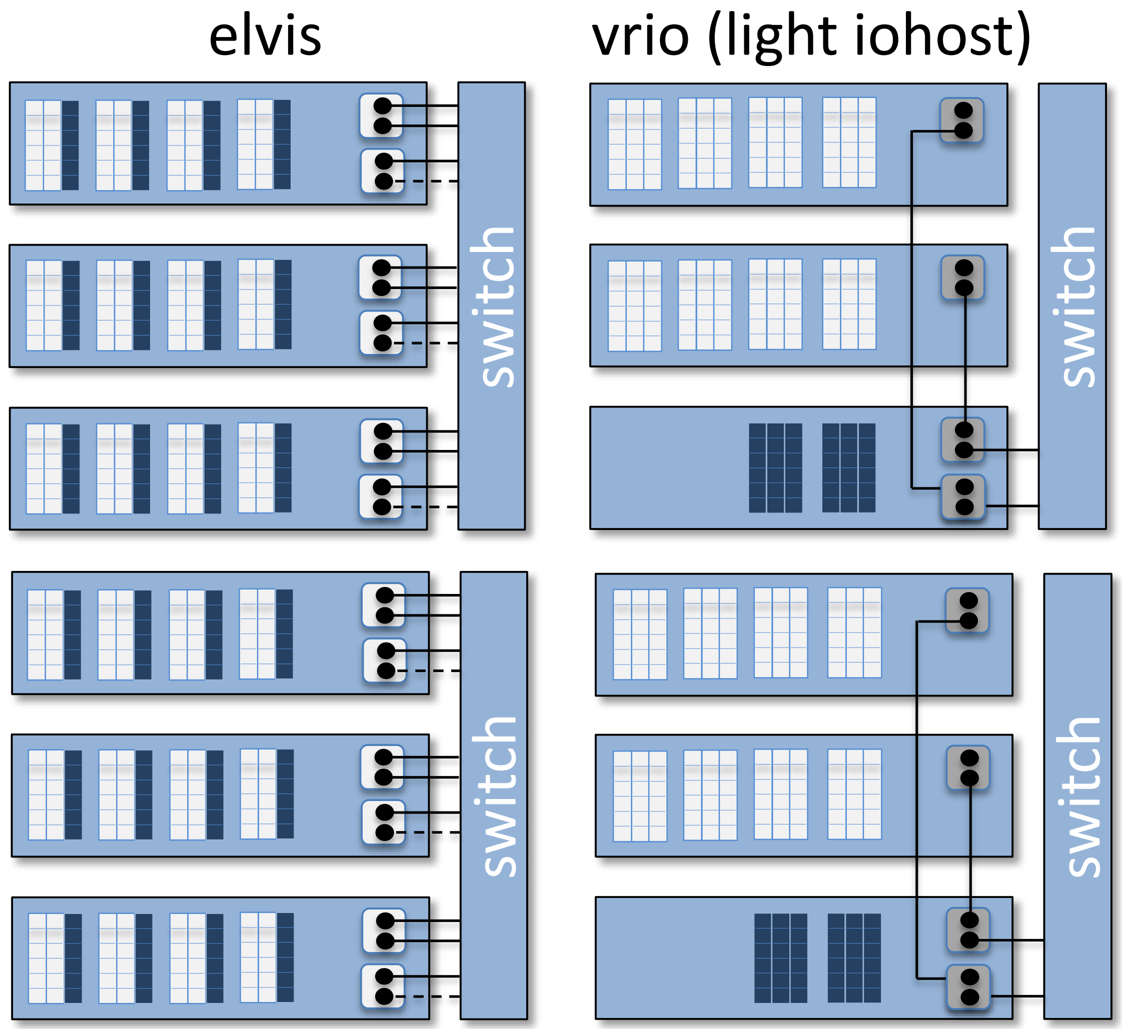
Cost tradeoffs

CONCRETE EXAMPLE

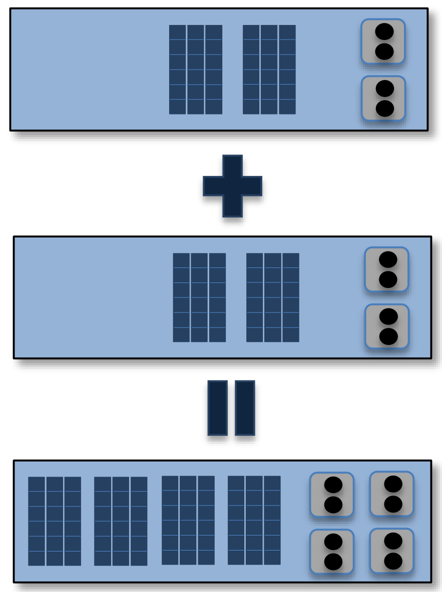
3 x Dell PowerEdge R930



6 x Dell PowerEdge R930



6 x Dell PowerEdge R930



heavy iohost
(only need 5 x R930)

Prices

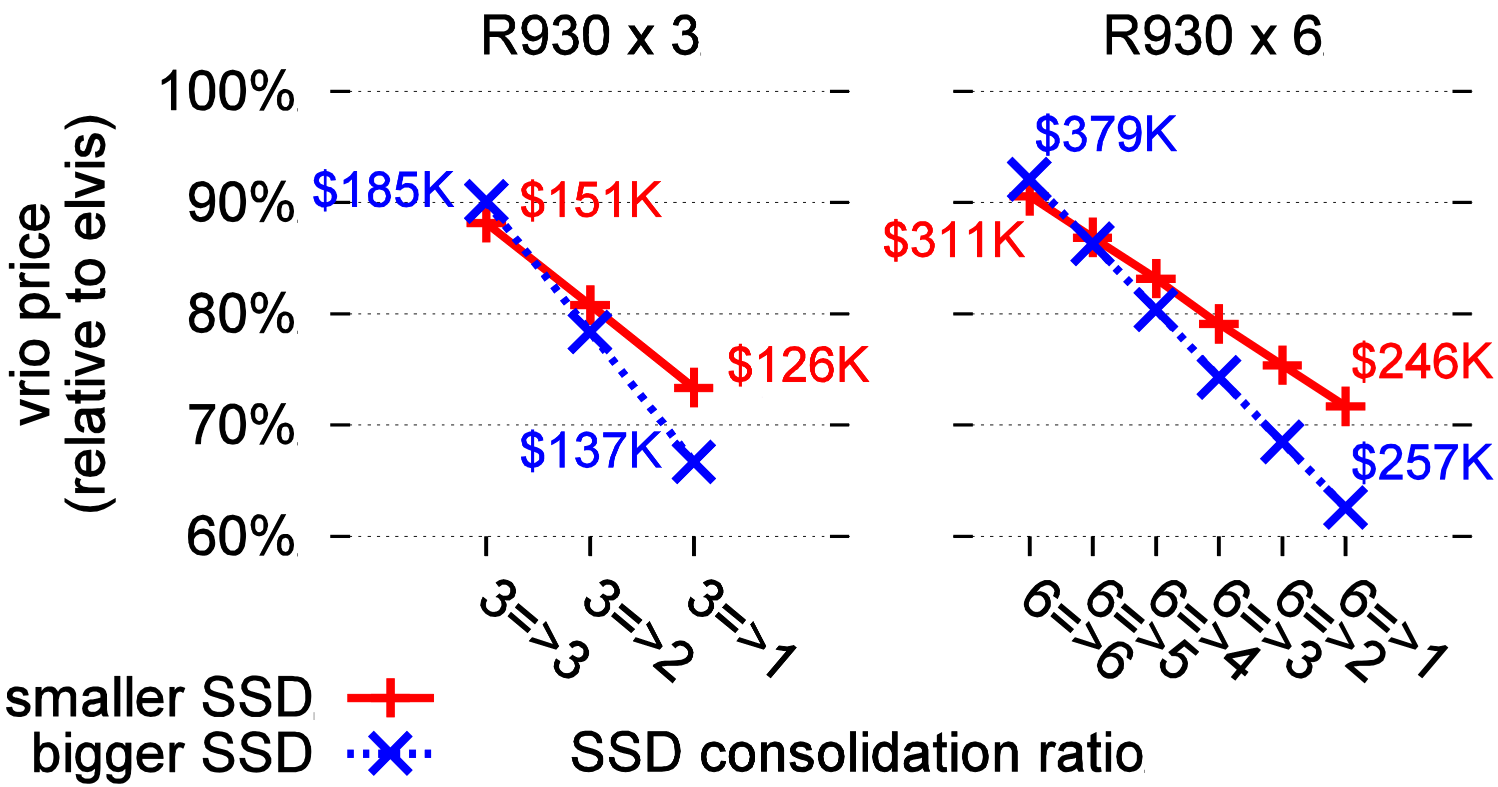
component	*price*	*elvis*	*vmhost*	*light iohost*	*heavy iohost*
base	\$6,407	1	1	1	1
18 core CPU	\$8,006	4	4	2	4
8GB DRAM	\$172		2	8	8
16GB DRAM	\$273	18	26		
10Gbps NIC DP	\$560	2			
40Gbps NIC DP	\$1,121		1	2	4
total server price		\$44.5K	\$47.0K	\$26.0K	\$44.2K
total Gbps		40.00	80.00	160.00	320.00
required Gbps		26.72	40.08	160.31	320.63

setup	*elvis servers*	*vrio servers*	*elvis price*	*vrio price*	*diff.*
R930 x 3	3	2+1	\$133.4K	\$120.0K	-10%
R930 x 6	6	4+1	\$266.9K	\$232.3K	-13%

I/O device consolidation

- **vrio allows for a new way to consolidate devices**
  - While supporting efficient, programmable interposition for VMs
- **Inherently different than other consolidation types**
  - NAS / SAN
- **How is the SAN exposed to VMs?**
  - Assignment? => no interposition
  - Traditional paravirtualization? => poor performance

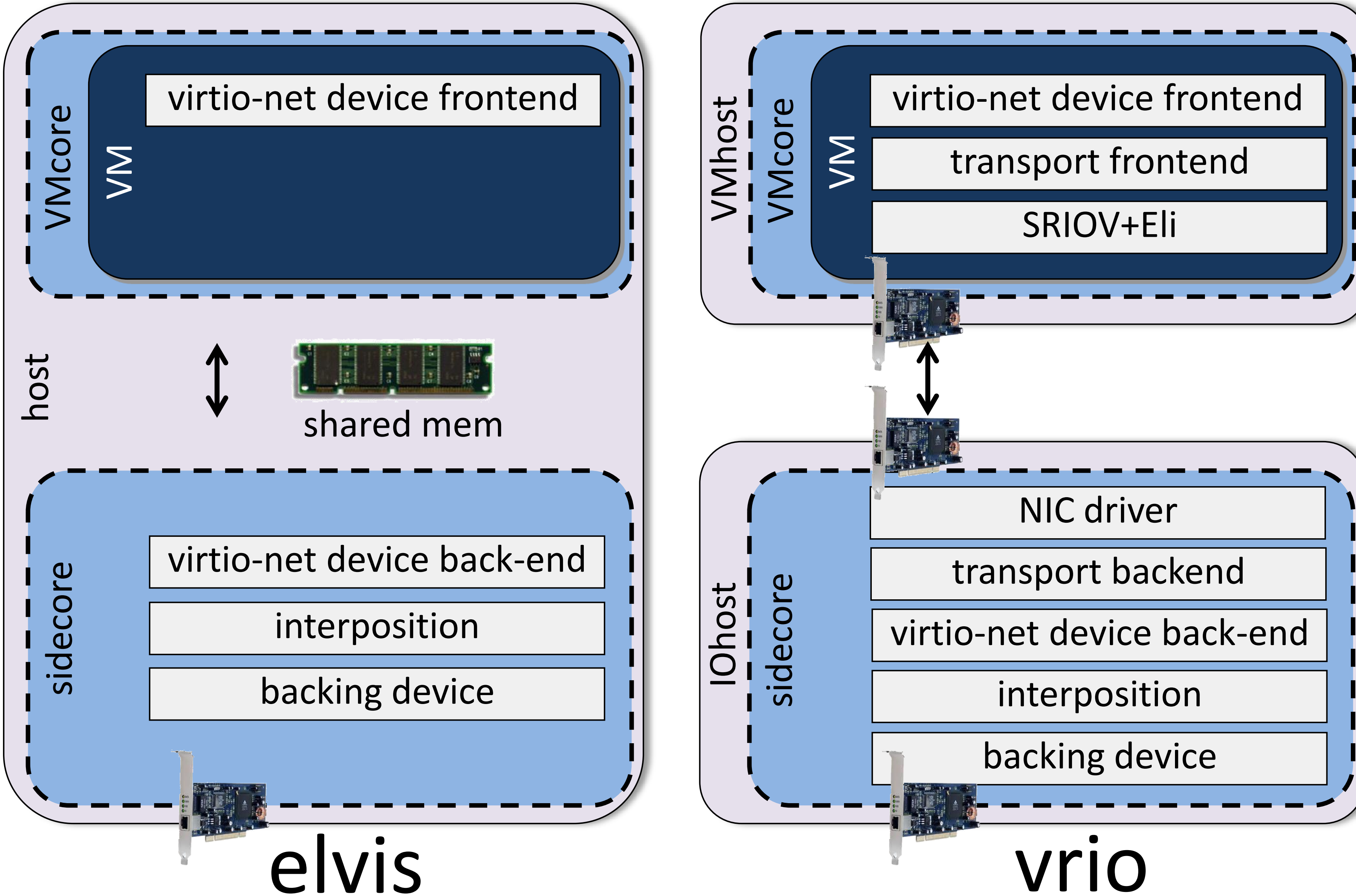
Prices



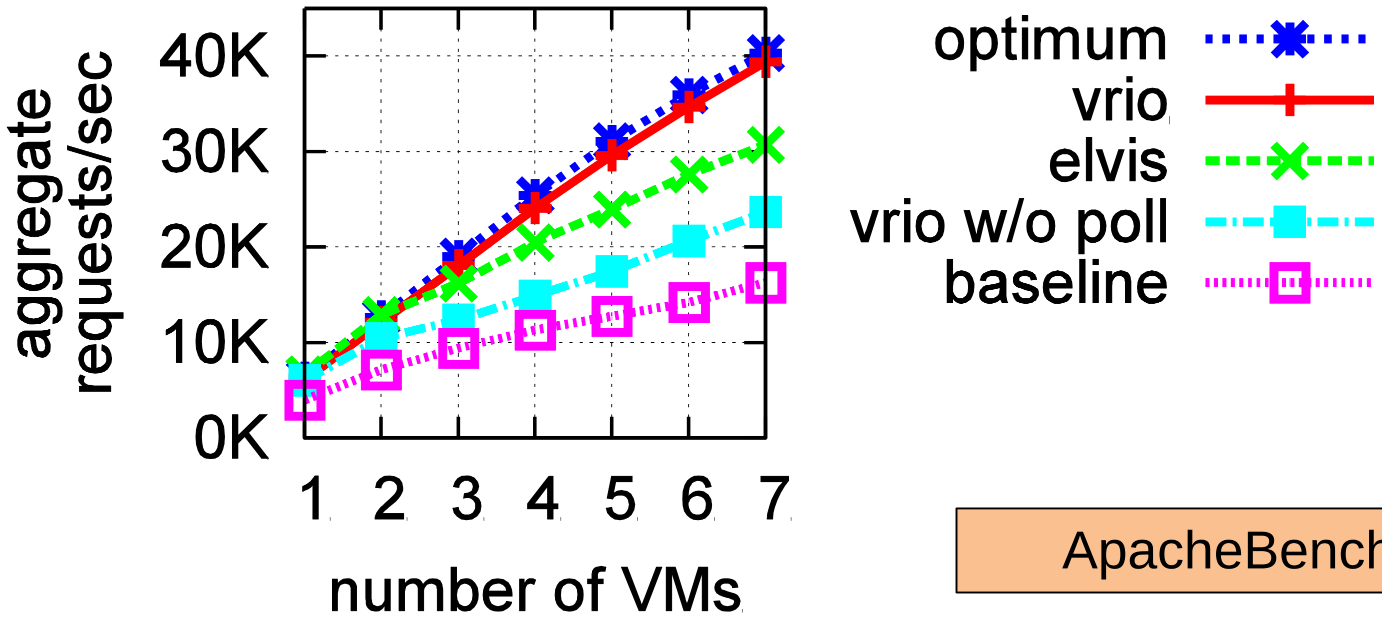
R930 can hold up to eight 3.2TB or 6.4TB FusionIO SX300 PCIe SSDs (that cost $12,706 and $24,063)

DESIGN & IMPLEMENTATION ISSUES

Interrupts & exits



Interrupts & exits



ApacheBench

I/O model	*sync exits*	*guest intrpts*	*intrpt injection*	*host intrpts*	*IOhost intrpts*	*sum*
optimum	0	2	0	0	-	2
vrio	0	2	0	0	0	2
elvis	0	2	0	2	-	4
vrio w/o poll	0	2	0	0	4	6
baseline	3	2	2	2	-	9

Other implementation issues

- **Segmentation & reassembly**
  – Use TSO (HW's TCP segmentation offload extension) despite operating at the Ethernet level
- **Zero copying**
  – SKB reuse tricks and carefully choosing jumbo frame size (of transport) to make the possible
- **Error handling**
  – Transport (=Ethernet) is unreliable
  – Network will get by (UDP, TCP), but block needs retransmit
- **Live migration**
  – Temporarily replace transport NIC instance with virtio

- **(See paper for details)**

Drawbacks

- **Fault tolerance**
  - What happens if an IOhost goes down?
    - VMhosts get disconnected
  - Can we solve this problem?
    - Yes, connect VMhosts to IOhosts via the switch rather than directly
      - Costlier: requires more cables and stronger switches
    - Can also envision having secondary IOhosts
  - In this case (connected despite IOhost going dowm)
    - Fall back on virtio
  - Block: local vs. distributed storage
- **Power**
  - Monitor/mwait (increases latency somewhat)

Benefits

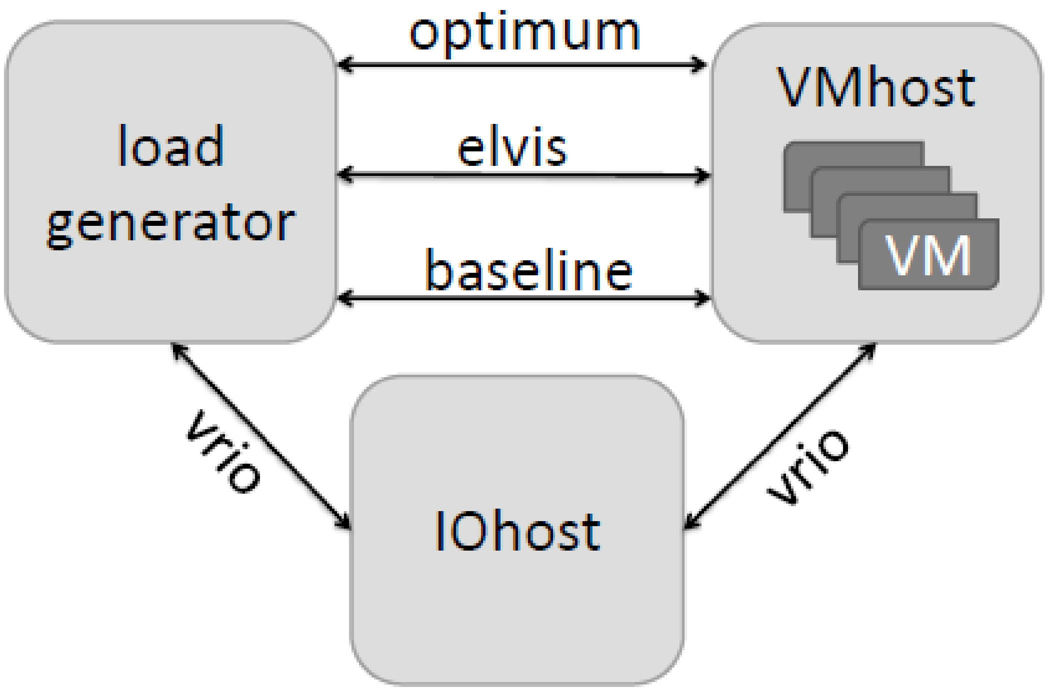
- **Sidecore consolidation**
  - Win in either price or performance

- **New way to do I/O device consolidation for VMs**
  - With efficient, programmable interposition
  - What’s the difference between vRIO and SAN/NAS?

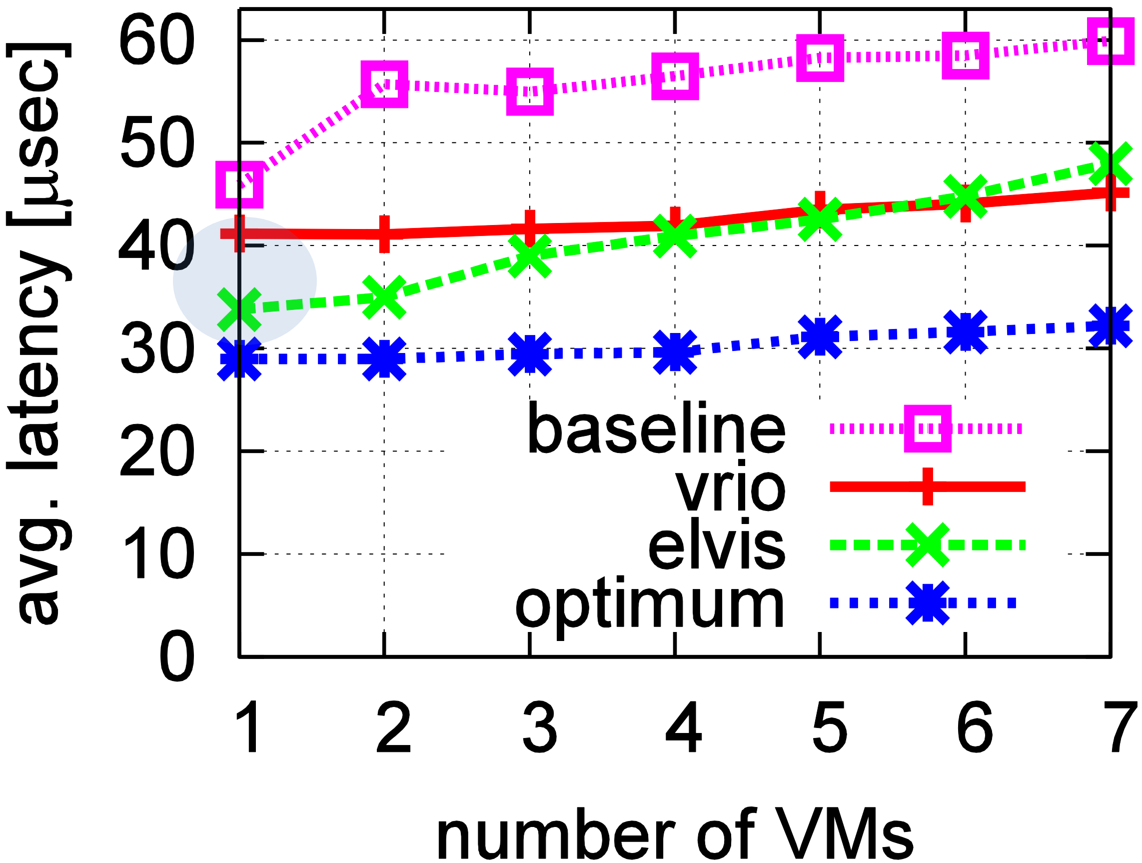
- **Friendly to heterogeneous setups**
  - Can deploy centralized I/O policies/services that are agnostic to
    - Hardware running VMs (x86? POWER? ARM?)
    - Local hypervisor supporting VMs (ESX, HyperV, KVM, Xen, …)
      - Valuable, since today that are many more hypervisors than OSes in production use
    - Even applicable to bare-metal OSes

PERFORMANCE EVALUATION

Simplest experimental setup

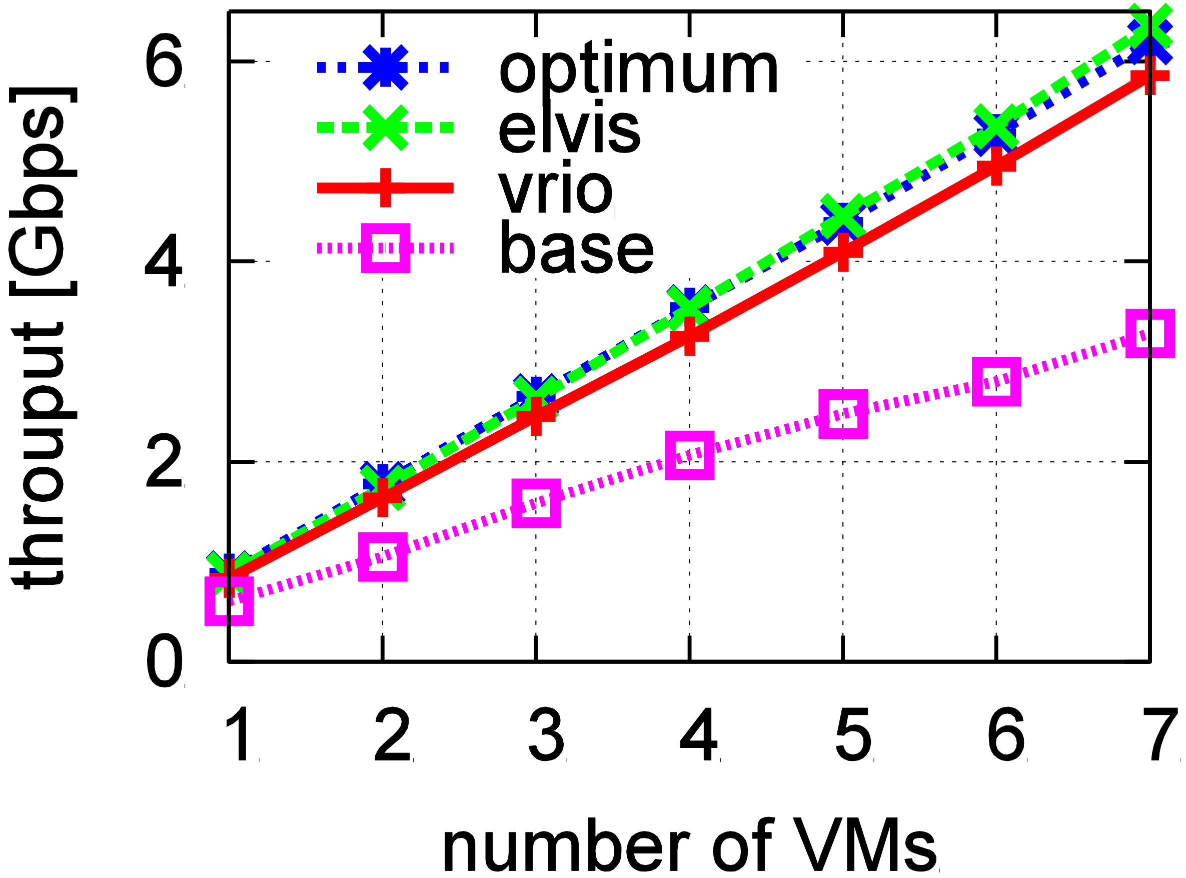


Latency – most problematic aspect



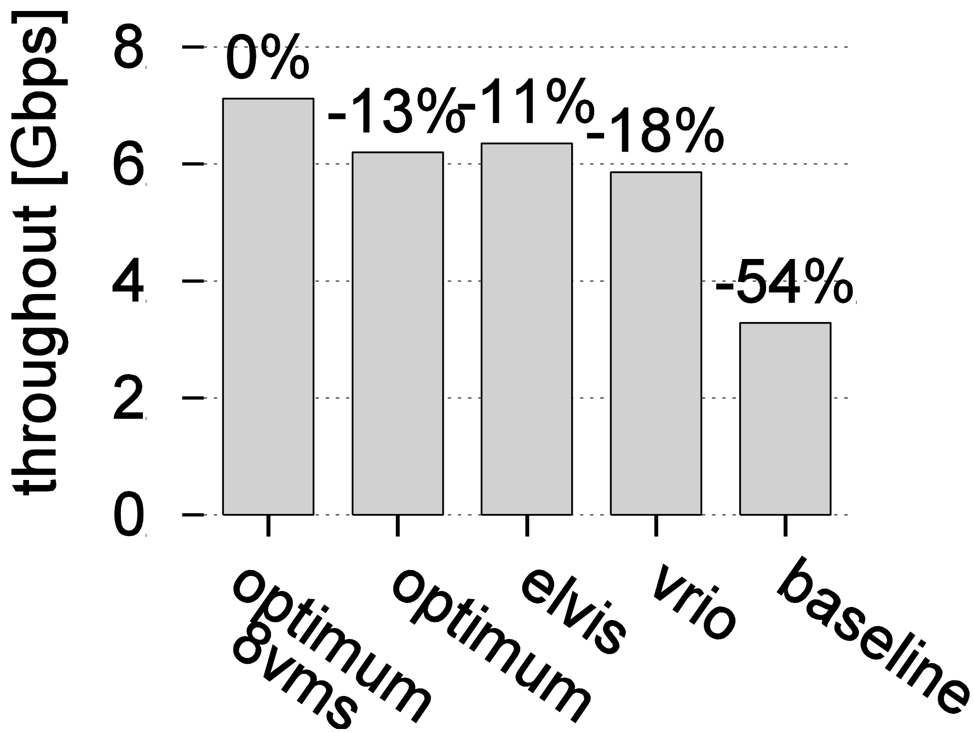
Netperf UDP request-response
(vrio 18% slower than elvis – worst result without interposition!)

Throughput



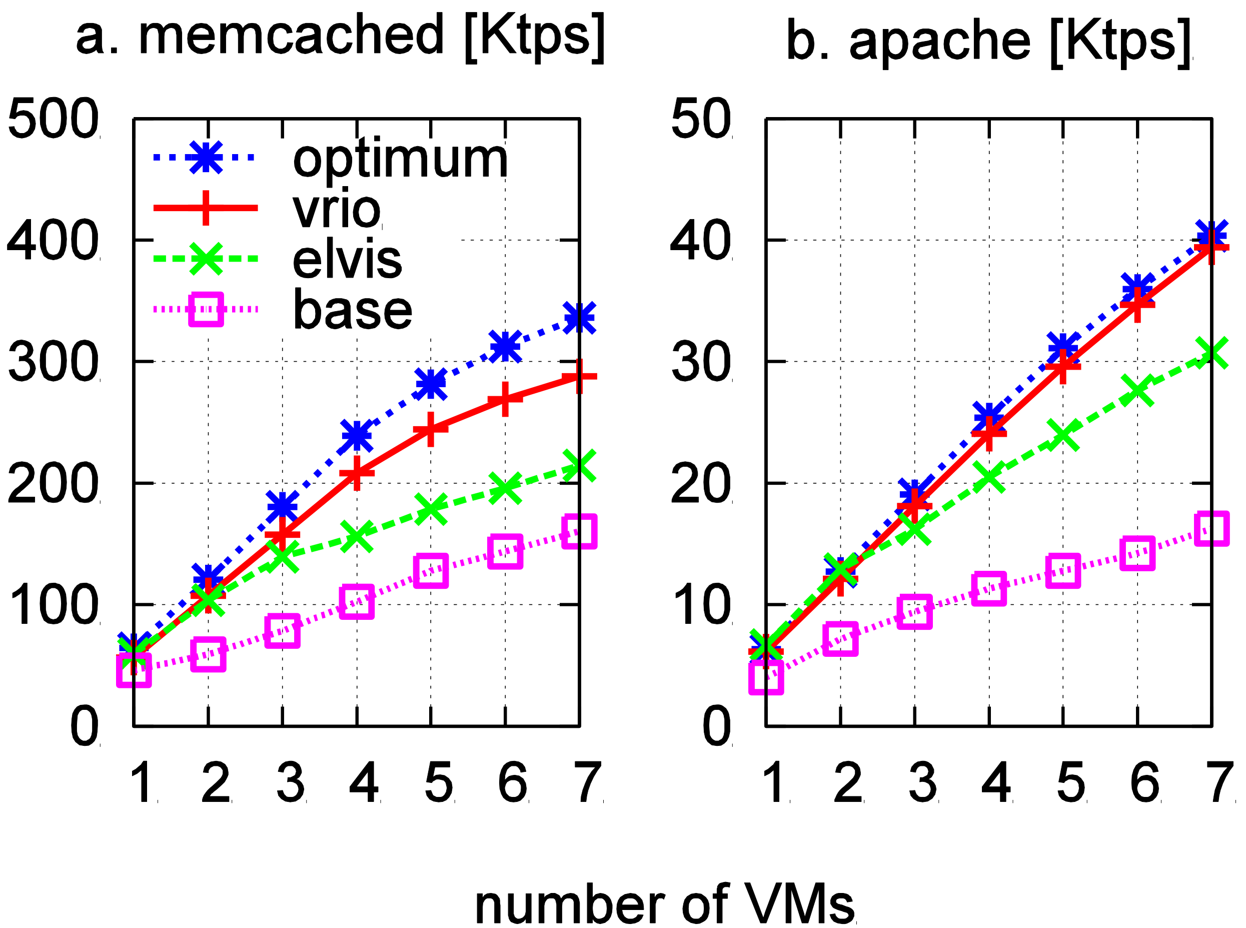
Netperf TCP stream

Throughput – 8 cores



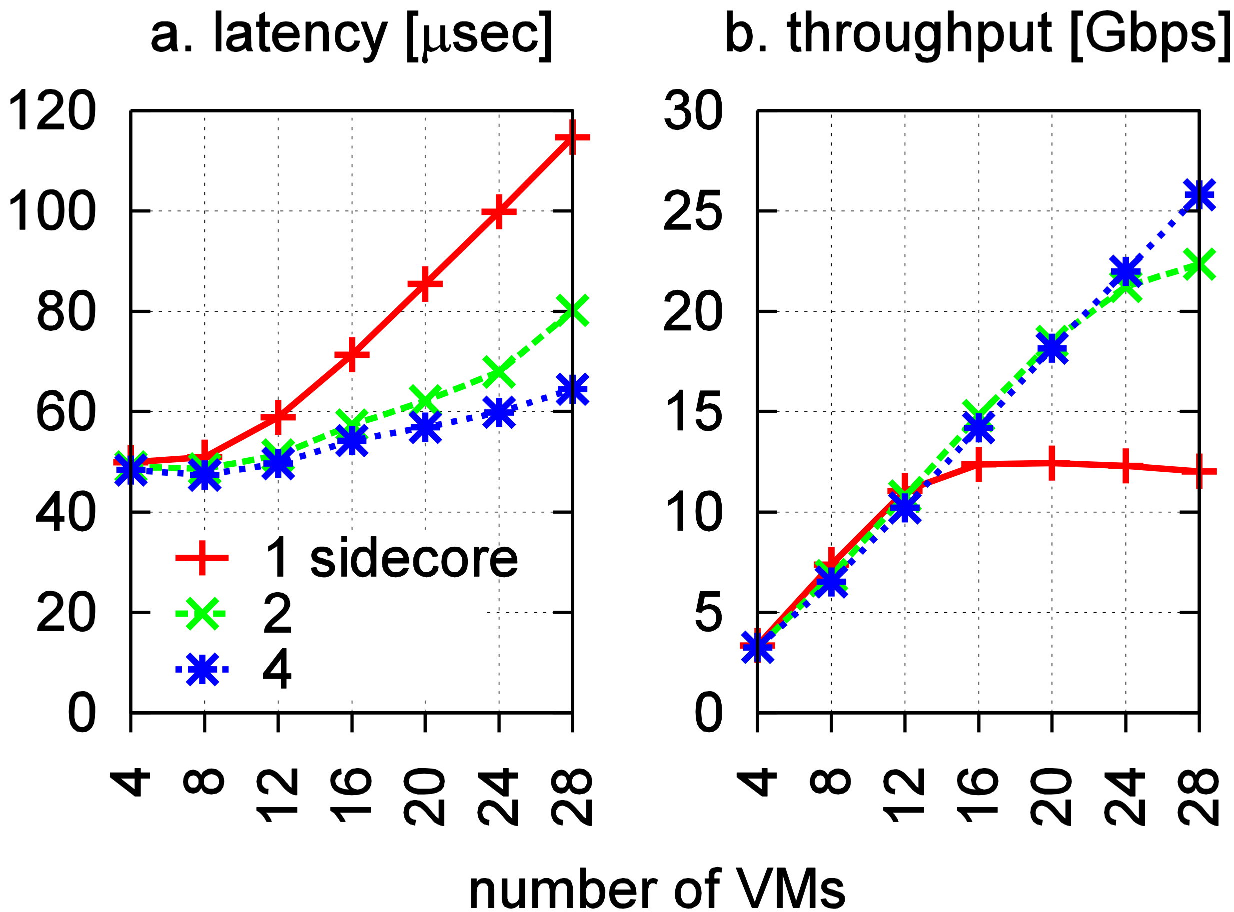
Netperf TCP stream

Macrobenchmarks



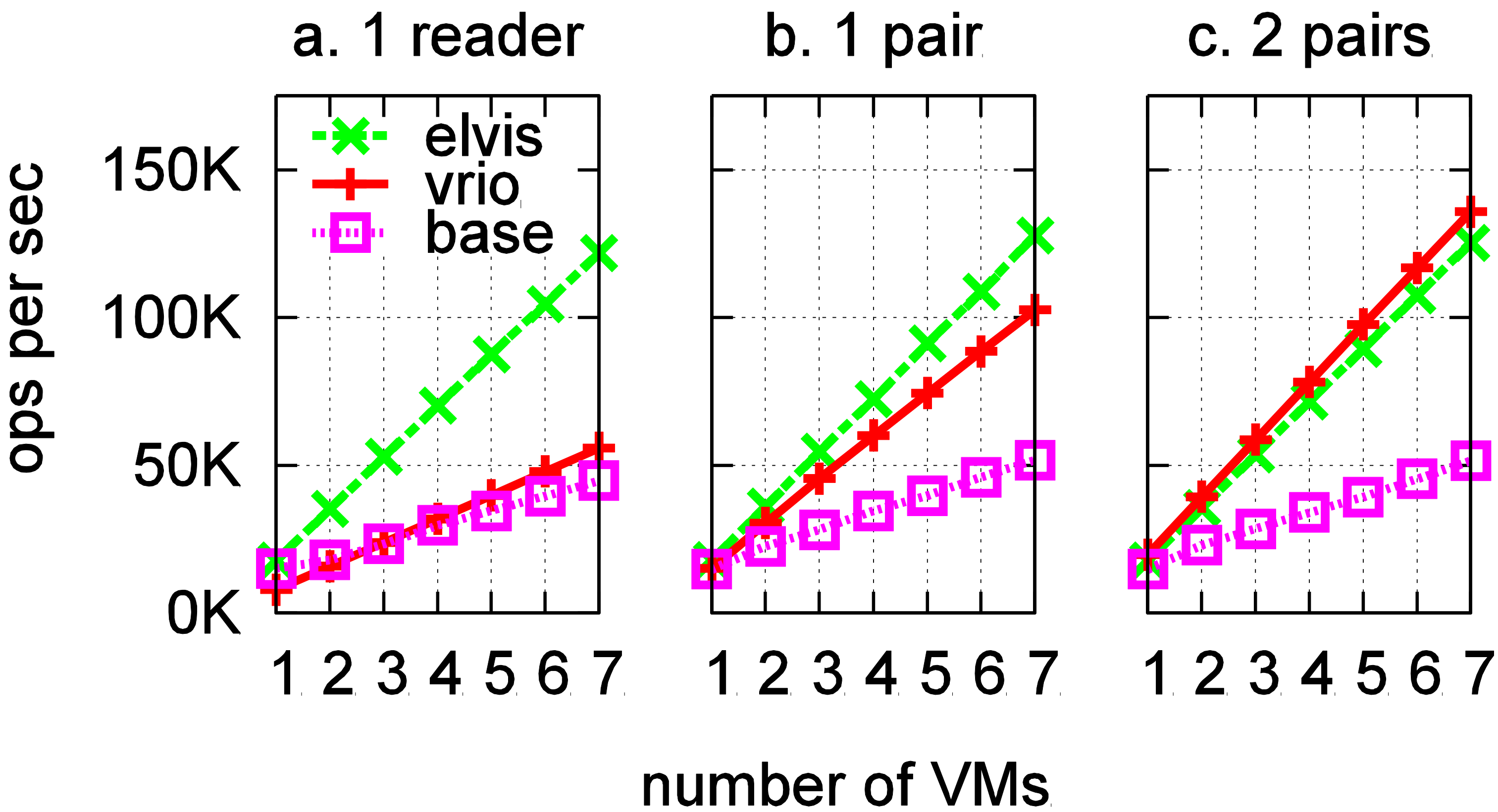
Y = kilo transactions/sec

IOhost scalability



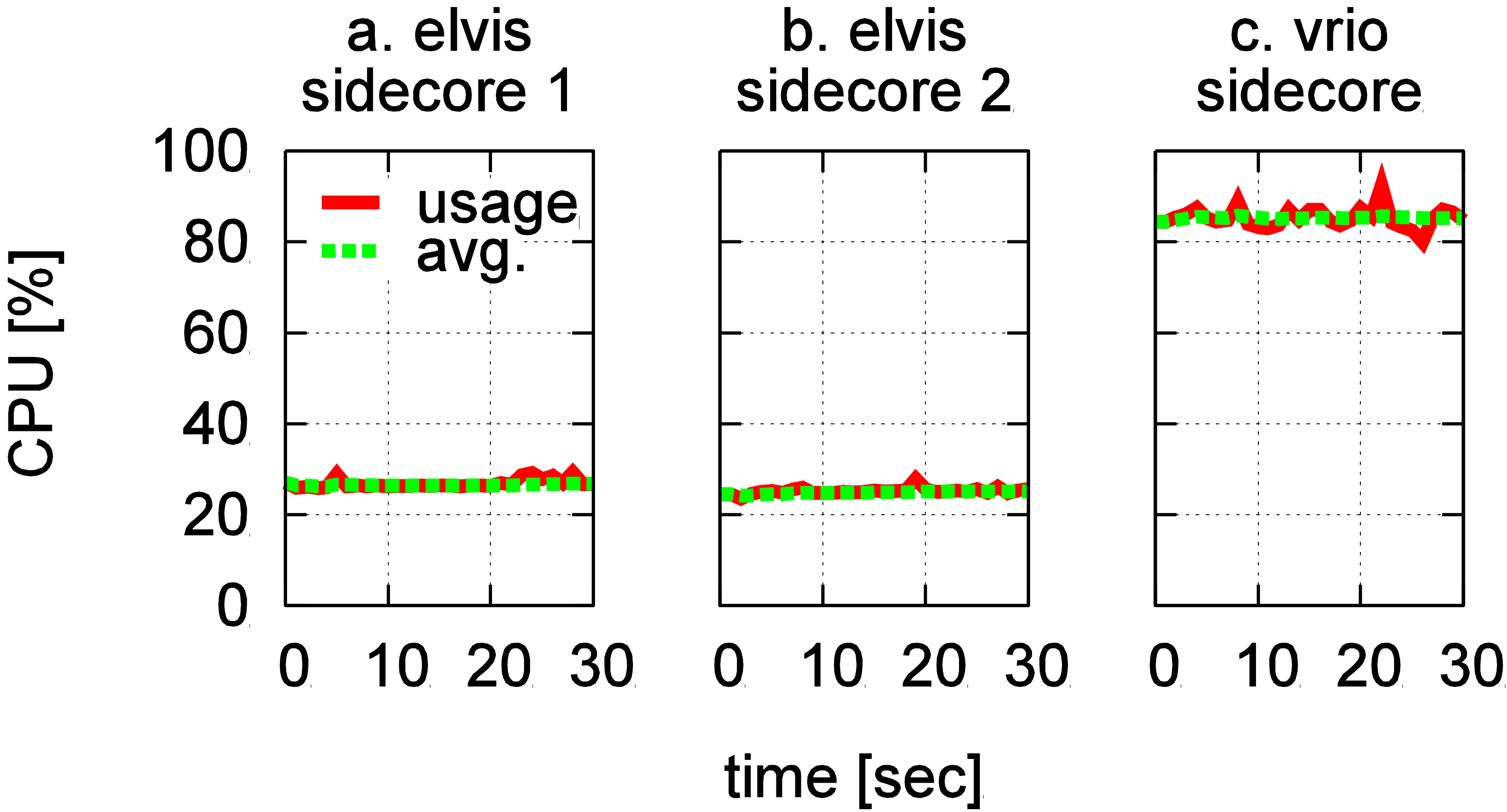
four 8-core VMhosts

Making a local device remote



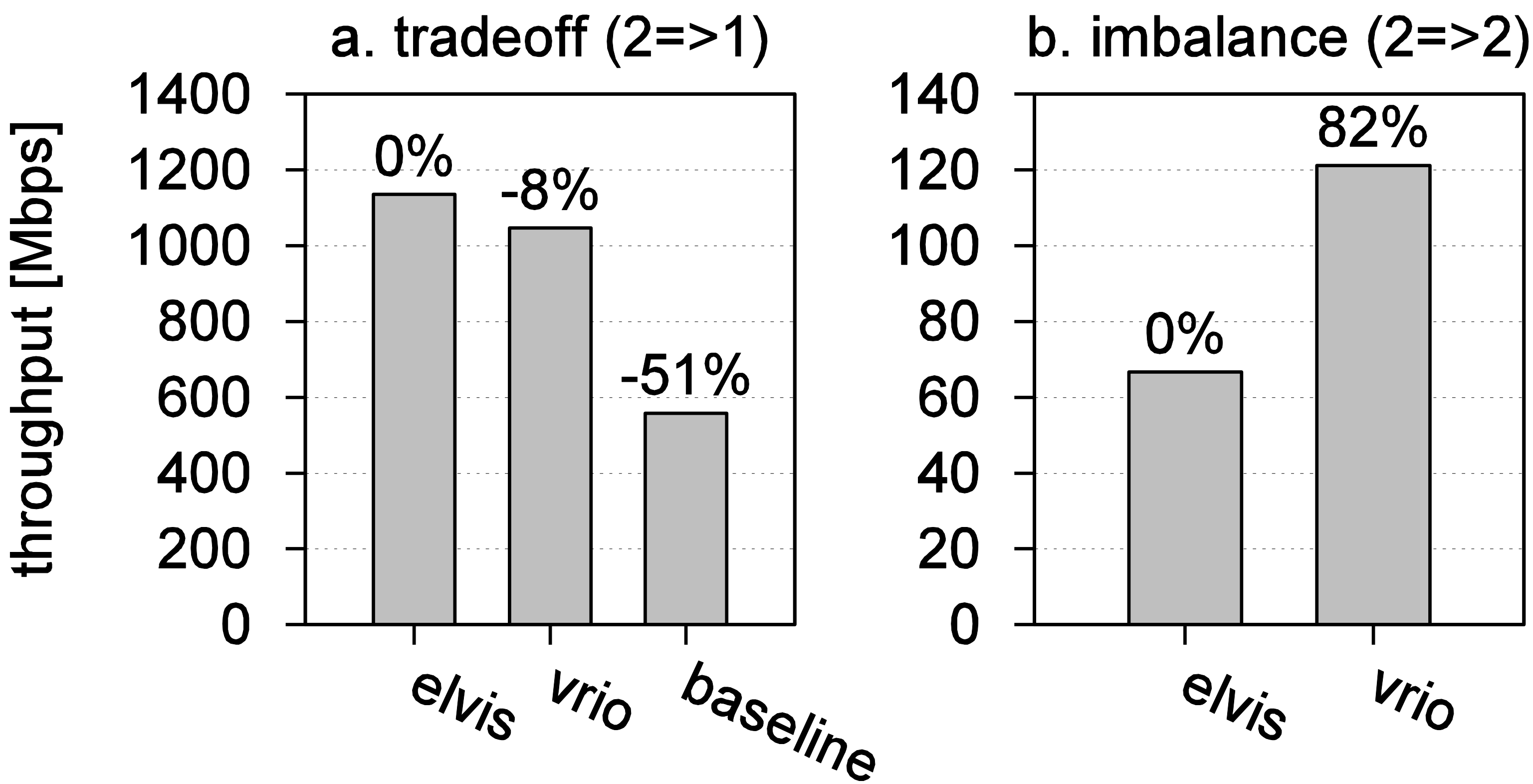
Filebench on ramdisk: random 4KB (O_DIRECT) IOps
(vrio up to 2.2x worse)

Improved utilization



Filebench webserver personality

Sidecore consolidation tradeoff



Filebench webserver personality (a) without and (b) with imbalance + interposition (encryption).

Conclusions

- **It makes sense to consolidate sidecores**
  - In terms of both price
  - And performance

Backup

Throughput study