Efficient and Scalable Virtio (aka ELVIS)

Nadav Har’El[×] **Abel Gordon**[×] Alex Landau[×]

Muli Ben-Yehuda[×,¤] Avishay Traeger[×] Razya Ladelsky[×]

[×] IBM Research – Haifa
[¤] Technion and Hypervisor Consulting

Sheraton Grand Hotel Edinburgh
Edinburgh, UK
October 21 - 23, 2013

Why (not) Virtio ?

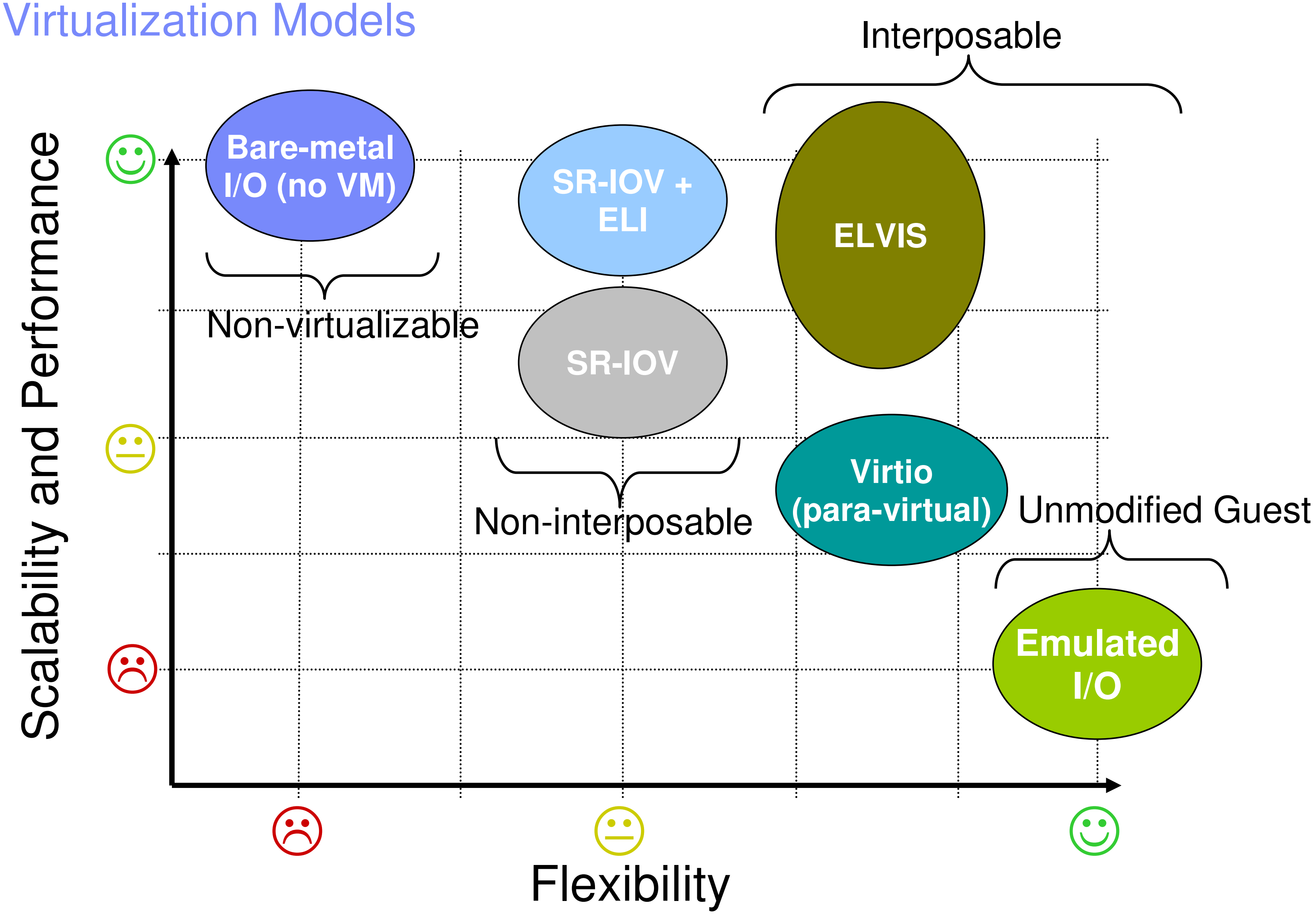
- Pros
  - Software Defined Networking
  - File based images
  - Live Migration
  - Fault Tolerance
  - Security
  - ….

- Cons
  - Scalability Limitations
  - Performance Degradation

–Scalability Limitations

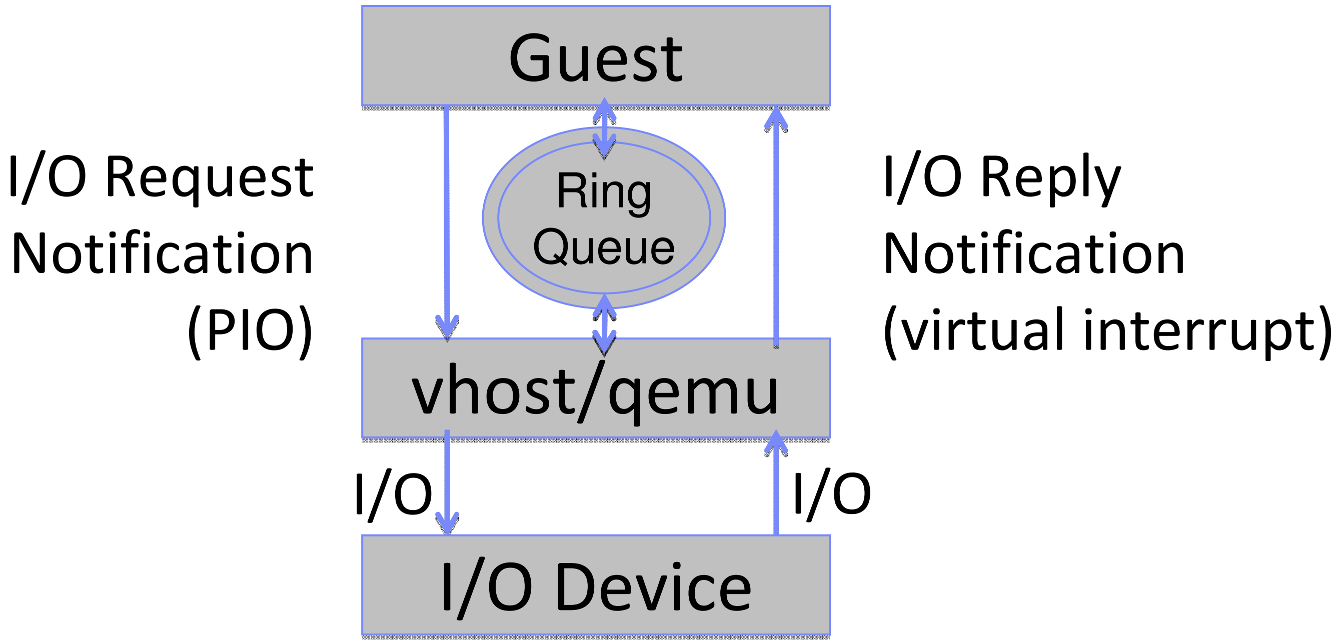
–Performance Degradation

I/O Virtualization Models



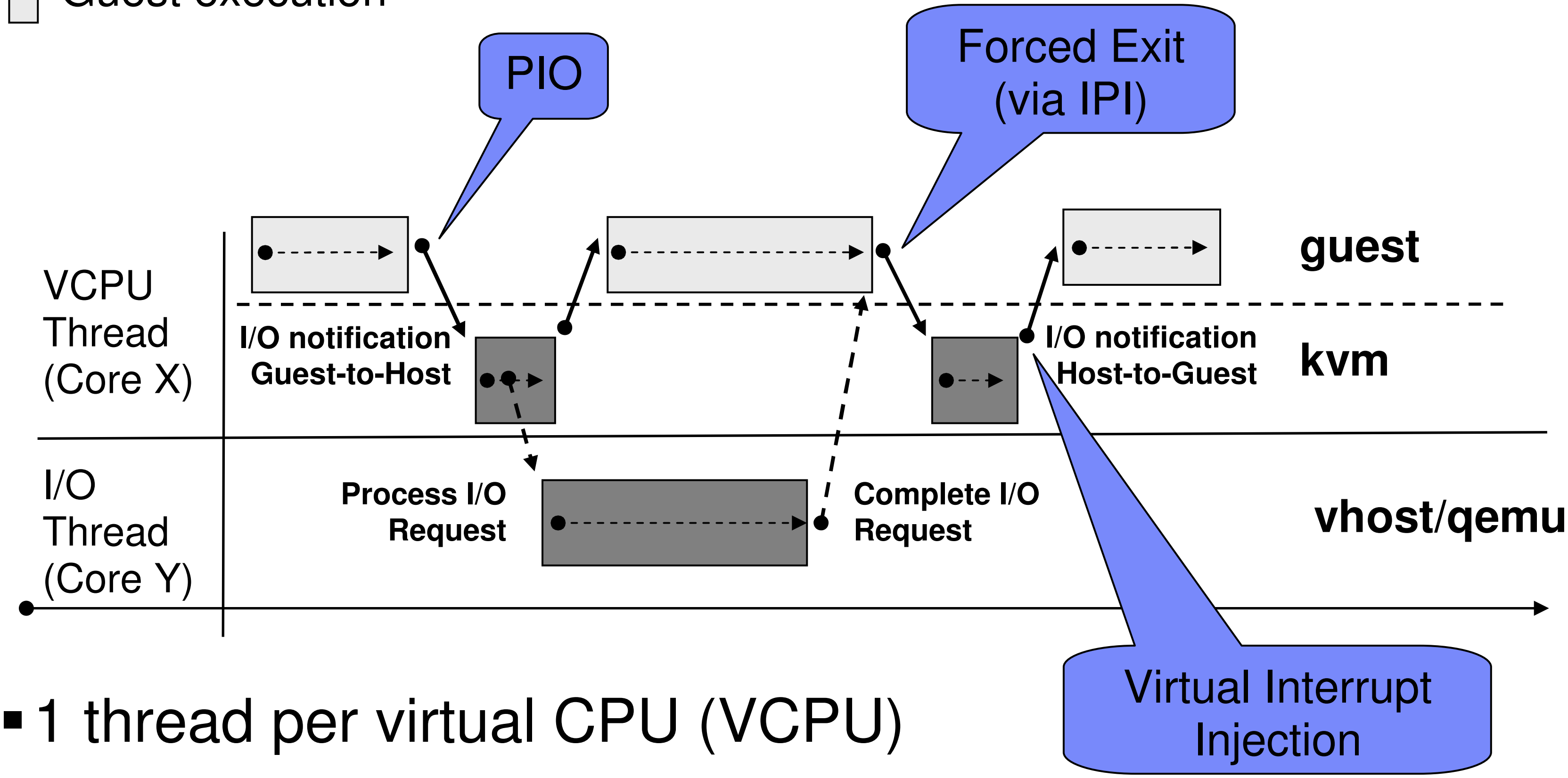
How Virtio works today ?

- The guest posts I/O requests in ring-queue (shared with the QEMU or vhost) and sends a request notification (PIO)
- QEMU or vhost processes the requests and sends a reply notification (virtual interrupt)



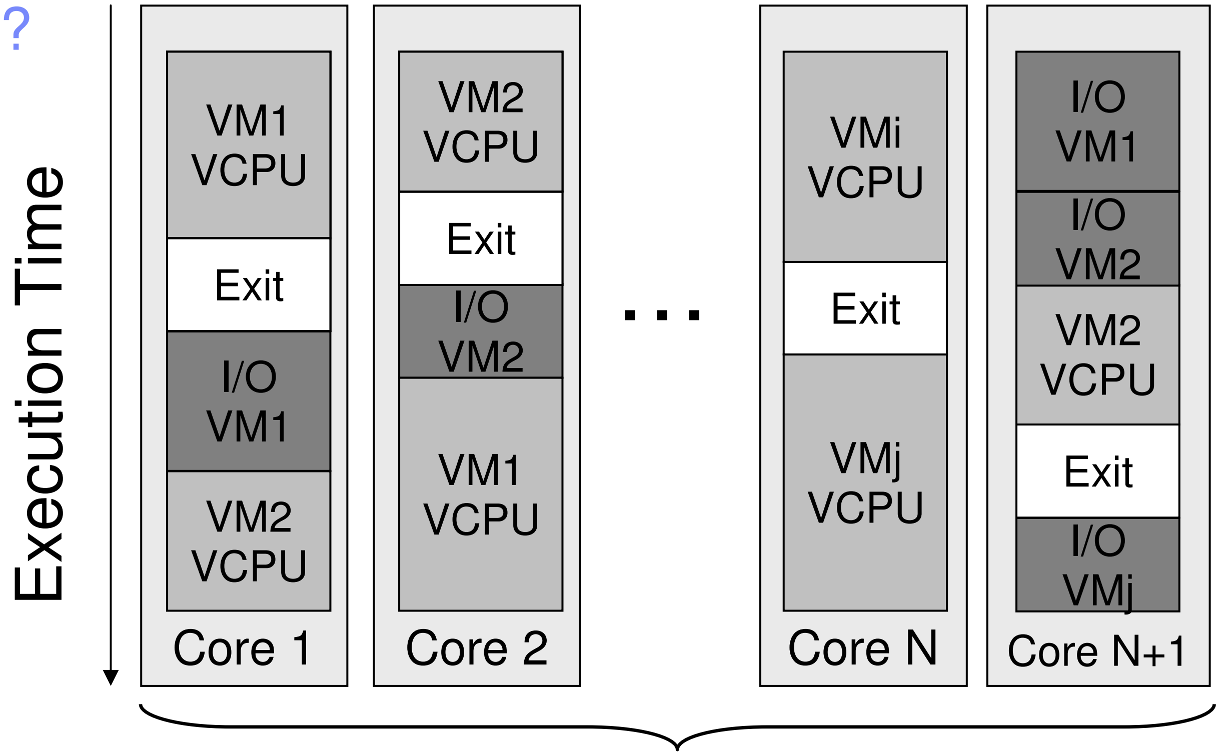
How I/O notifications are sent/received

●→ CPU context switch (VMExits and VMEntries)
■ I/O processing
□ Guest execution



- 1 thread per virtual CPU (VCPU)
- 1 or more threads per virtual I/O device

Is this model scalable with the number of guests, cores and I/O bandwidth ?



VCPU and I/O thread-based scheduling for all cores
(host Linux scheduler)

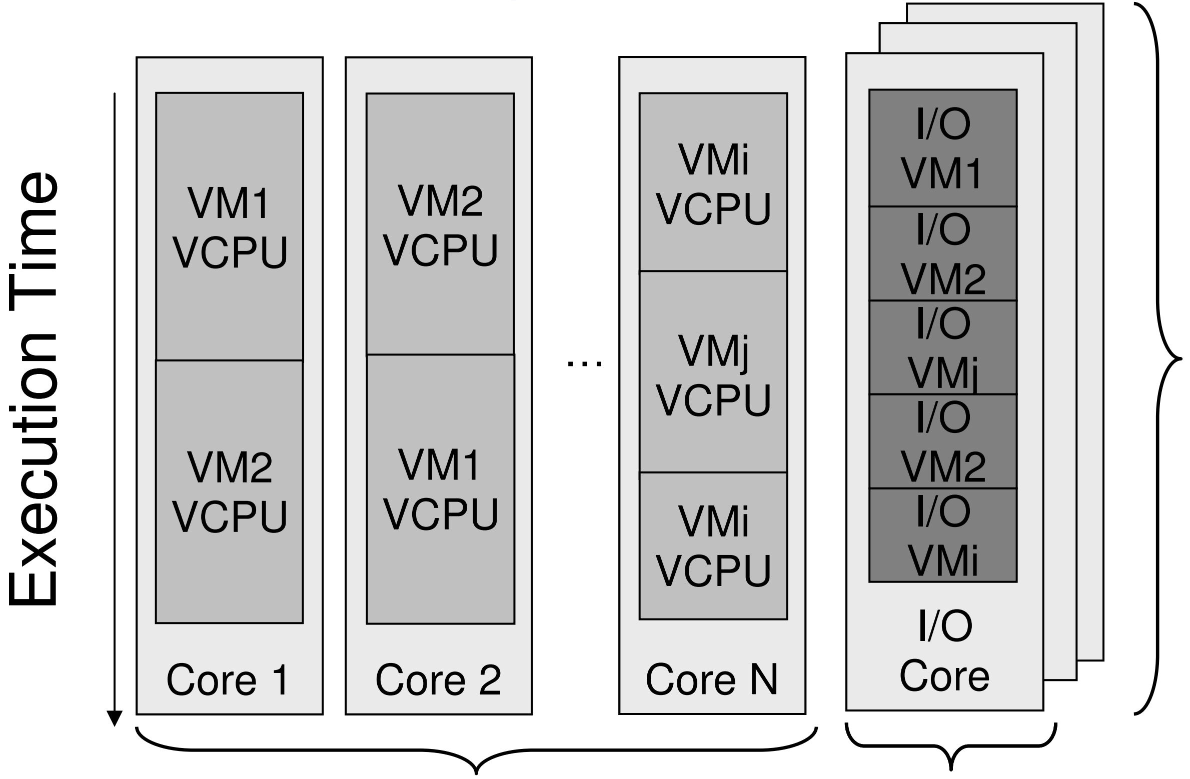
Depends on Linux (host) thread scheduler but the scheduler has no information about the I/O activity of the Virtio queues….

Facts and Trends

- Notifications cause exits (context switches) == overhead!
- Current trend is:
  - Towards multi-core systems with an increasing numbers of cores per socket (4->6->**8**->16**->**32) and guests per host
  - Faster networks with expectation of lower latency and higher bandwidth (1GbE->10GbE->**40GbE**->100GbE)
- I/O virtualization is a CPU intensive task, and may require more cycles than the available in a single core

We need a Virtio back-end that considers these facts and trends!

ELVIS (based on vhost): use fine-grained I/O scheduling and dedicate cores to improve scalability and efficiency



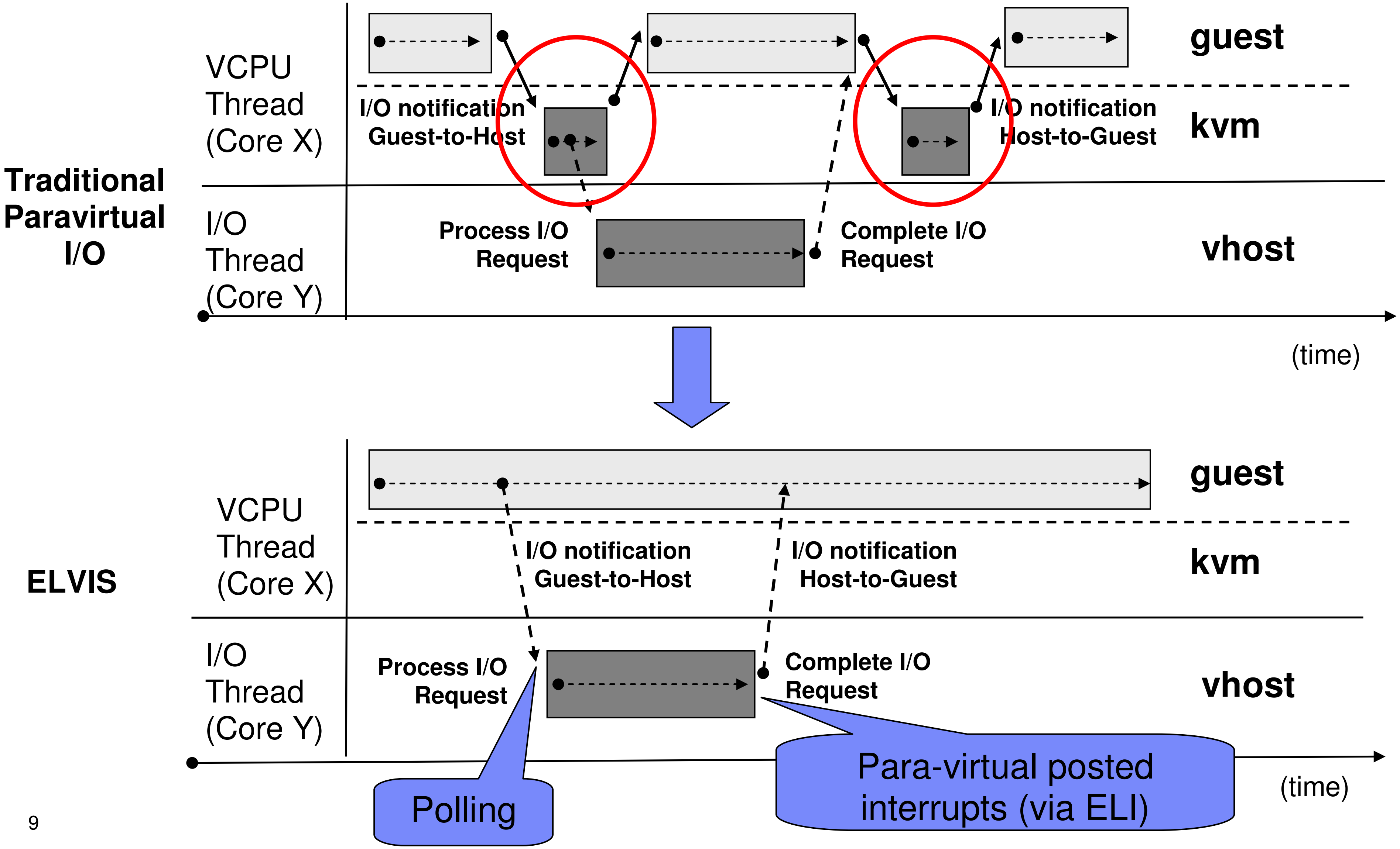
1 vhost thread per I/O core handles requests of many VMs/Virtio devices

thread-based scheduling

fine-grained I/O scheduling

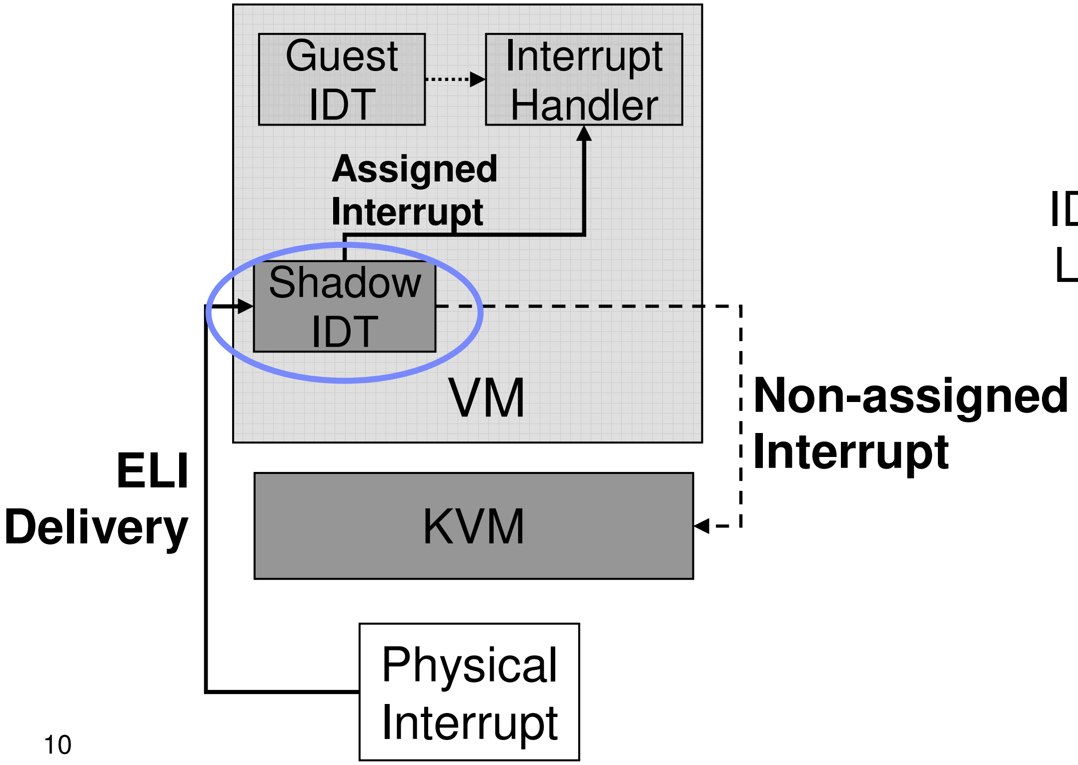
- Process queues based on the I/O activity
- Balance between throughput and latency
- No process/thread context switches for I/O
- Exitless communication (next slide)
- Consider if the queue’s owner (VM) is running or not (not yet implemented)

ELVIS: remove notifications overhead to further improve efficiency



ELI: Exitess Interrupts to simulate Posted Interrupts

- ELI configures the CPU to deliver all interrupts to the guest
- ELI runs the guest using a shadow IDT
- Host interrupts are bounced back to the host in the form of exceptions and re-generated with software interrupts/self IPI
- …without the guest being aware of it


Guest IDT

Interrupt Handler

Assigned Interrupt

Shadow IDT

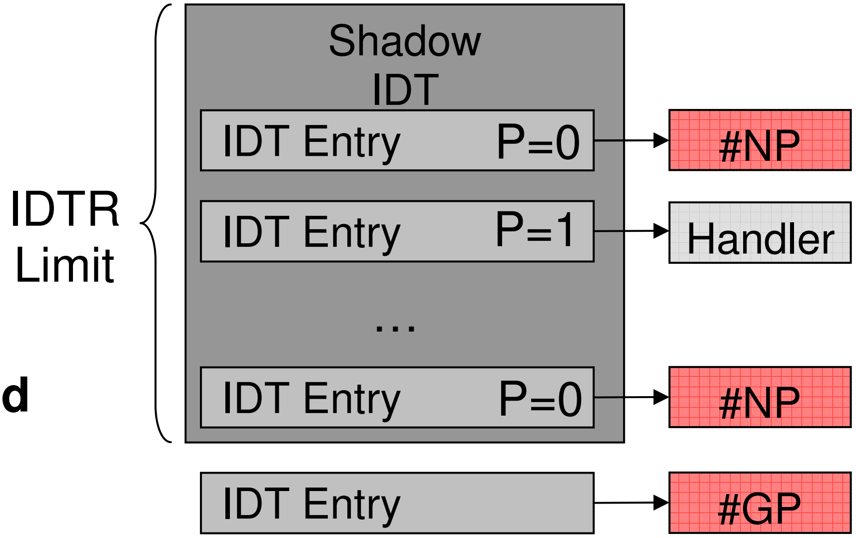
VM

KVM

Non-assigned Interrupt

ELI Delivery

Physical Interrupt


IDTR Limit

Shadow IDT

IDT Entry P=0 → #NP

IDT Entry P=1 → Handler

…

IDT Entry P=0 → #NP

IDT Entry → #GP

ELI: Exitless Interrupts - Completion

- Guests write to the LAPIC EOI register
- Old LAPIC interface:
  - KVM traps memory accesses → **page granularity**
- New LAPIC interface (x2APIC), required for Exitless Completions
  - KVM traps accesses to MSRs → **register granularity**

ELI gives direct access only to the EOI register

```
Cache L1     :640 KB
Cache L2     :2560 KB
Cache L3     :30720 KB
Ratio Status:Unlocked (Min:
Ratio Actual Value:18

Hardware Prefetcher                  [Enabled]
Adjacent Cache Line Prefetch         [Enabled]
ACPI MADT ordering                   [Modern ordering]
Max CPUID Value Limit                [Disabled]
Intel(R) Virtualization Tech         [Enabled]
x2APIC                               [Enabled]
Intel(R) HT Technology               [Enabled]
```

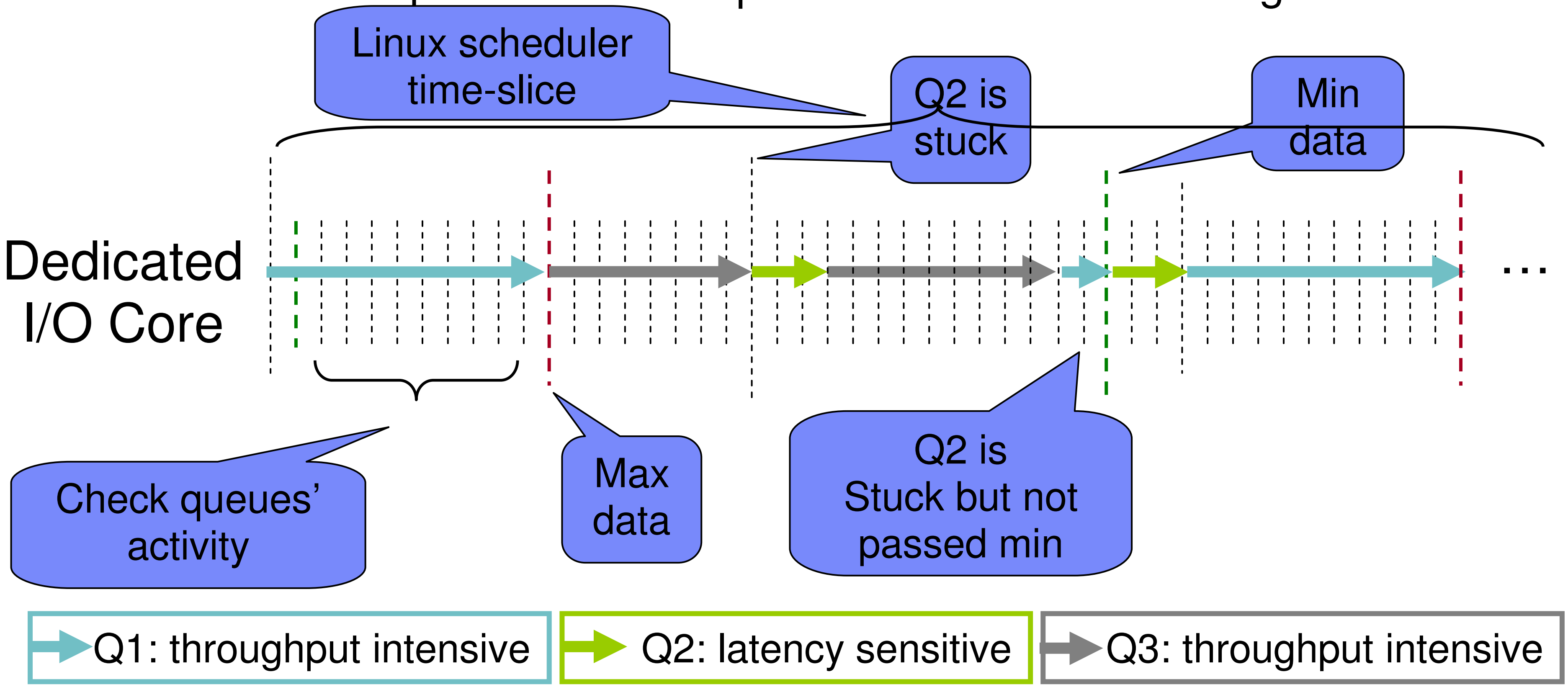
Para-Virtual Posted Interrupts based on ELI

- Posted-Interrupts: new HW feature to inject a virtual interrupt from a core running in root mode to a VM running in a different core (guest mode) without forcing an exit

- Para-Virtual Posted interrupts:
  - Write the virtual interrupt vector to be injected in a descriptor (shared memory between KVM and the guest's kernel)
  - Send IPI (pre-defined vector) using ELI
  - Guest is modified to handle the IPI and call the corresponding (virtual) interrupt handler

ELVIS: Fine-grained I/O scheduling in a nutshell

- Single vhost-thread in a dedicated core:
  - Monitors the activity of all queues (number of pending requests, how long the requests are waiting, queue progress…)
  - Decide which queue should be processed and for how long



Linux scheduler time-slice

Q2 is stuck

Min data

Dedicated I/O Core

…

Check queues' activity

Max data

Q2 is Stuck but not passed min

Q1: throughput intensive

Q2: latency sensitive

Q3: throughput intensive

ELVIS: Placement of threads, memory and interrupts

- Dedicate 1 I/O core per CPU socket
  - Cores per socket continue to increase year by year
  - More cores are required to virtualize more bandwidth at lower latencies (network links continue to be improved)
  - NUMA awareness: shared LLC cache and memory controller, DDIO technology
- Deliver interrupts to the “corresponding” I/O core
  - Interrupts are processed by I/O cores and do not disturb the running the guests
  - Improve locality
  - Multi-queue, Multi-port and SR-IOV adapters can dedicate interrupts per queue/port/virtual function

From Research to Practice: Status, Work in Progress and Future Work

- Patches published in github (based on Kernel 3.9)
  - https://github.com/abelg/virtual_io_acceleration/commits/ibm-io-acceleration-3.9-github

- Work in progress by Eyal Moscovici <eyalmo@il.ibm.com>
  - Control mechanism (sysfs interface) to:
    - allocate or de-allocate vhost threads on the fly
    - migrate a Virtio device/queue to a different vhost thread on the fly
  - Policy framework to monitor the system and orchestrate the control mechanism

- Get support to upstream the following features:
  1. Shared vhost-thread: same thread handles many virtio devices – default 1 thread per virtio device as it is today
  2. Control mechanism (sysfs): allocate/de-allocate vhost-threads and assign queues to vhost-threads
  3. Vhost statistics (sysfs): expose virtio queues and vhost-thread progress/load
  4. Polling optimization: poll queues to remove PIO exits (guest-to-host notifications)
  5. Policy mechanism: framework and rules to orchestrate the system (Python ?)
  6. Porting to PowerKVM

Performance Evaluation

- Implementation
  - Based on KVM (Linux Kernel 3.1 / QEMU 0.14)
  - With VHOST, in-kernel paravirtual I/O framework
  - Use ELI patches to implement para-virtual posted-interrupts and to improve hardware-assisted non-interposable I/O (SR-IOV)

- Experimental Setup
  - IBM System x3550 M4, dual socket 8 cores per socket Intel Xeon E2660 2.2GHz (SandyBridge)
  - Dual port 10GbE Intel x520 SRIOV NIC
  - 2 identical servers: one used to host the VMs and the other used to generate load on bare-metal

Methodology

- Repeated experiments using 1 to 14 UP VMs
  - 1x10GbE when running up-to 7 VMs
  - 2x10GbE when running more than 7 VMs

- Compared ELVIS against 3 other configurations

- **No interposition**
  - Each VM runs on a dedicated core and has a SR-IOV VF assigned using ELI
  - The closer ELVIS is to this configuration, the smaller the overhead is (used to evaluate ELVIS efficiency)

Methodology (cont.)

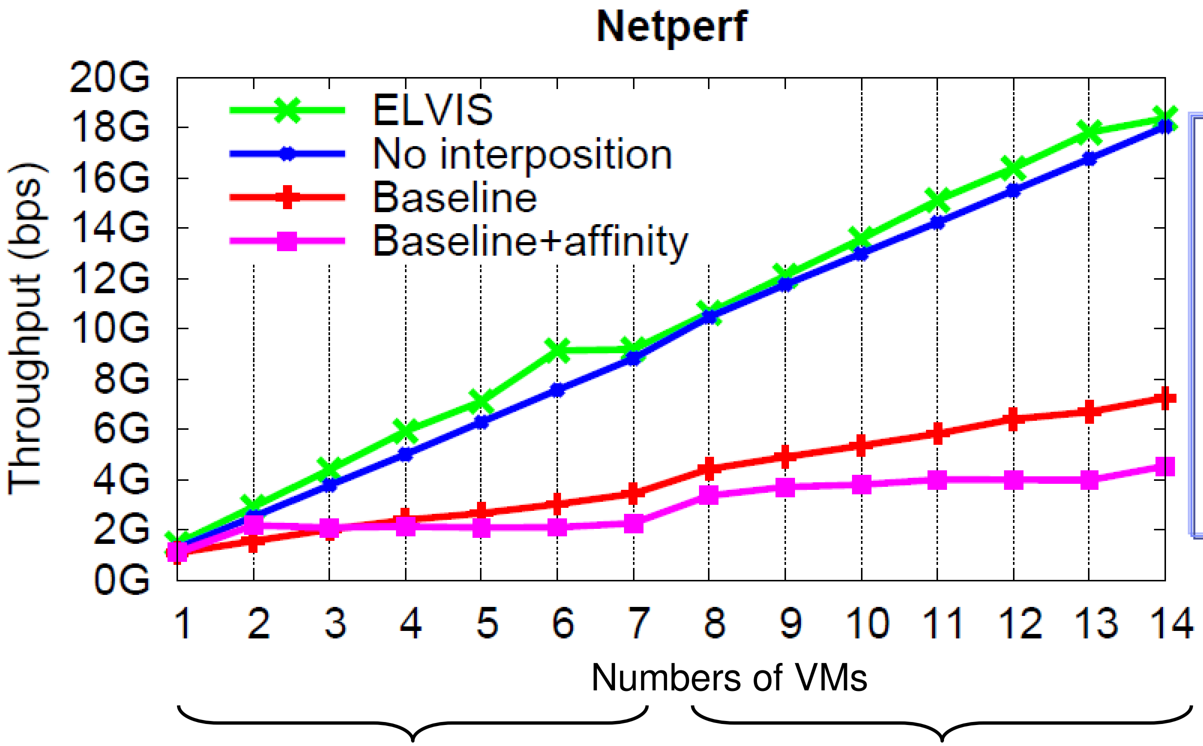
- N=number of VMs (1 to 14)
- Used N+1 cores (N≤ 7) or N+2 cores (N>7)
  - This is the resource overhead for I/O interposition

- **ELVIS**
  - 1 dedicated core per VCPU (VM)
  - 1 core (N<=7) or (N>7) 2 cores dedicated for I/O

- **Baseline**
  - N+1 cores (N ≤ 7) or N+2 cores (N>7) to run VCPU and I/O threads (no thread affinity)

- **Baseline+Affinity**
  - Baseline but dedicate 1 core per VCPU and pin I/O threads to dedicated I/O cores

Netperf – TCP Stream 64Bytes (throughput intensive)

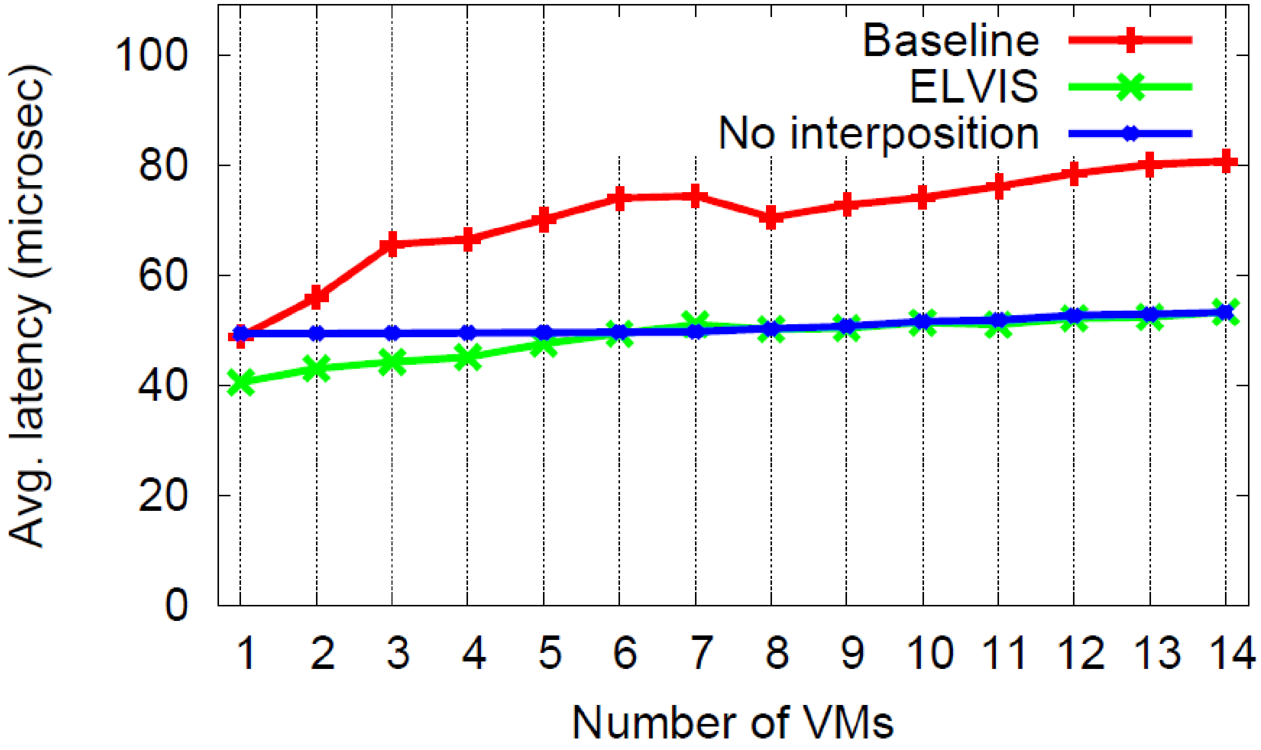


- •Scaled perfectly
- •1 core managed to handle I/O for 7 VMs (cores)
- •Maximum throughput
- •Coalesced more interrupts than the SR-IOV device (4K-11K vs. 30K ints/sec)

1x10Gb port
ELVIS: 1 core dedicated for I/O and 1 dedicated core per VM (N+1 total)
Baseline: N+1 cores (to handle I/O and to run the VMs)
No Interposition: N cores to run the VMs

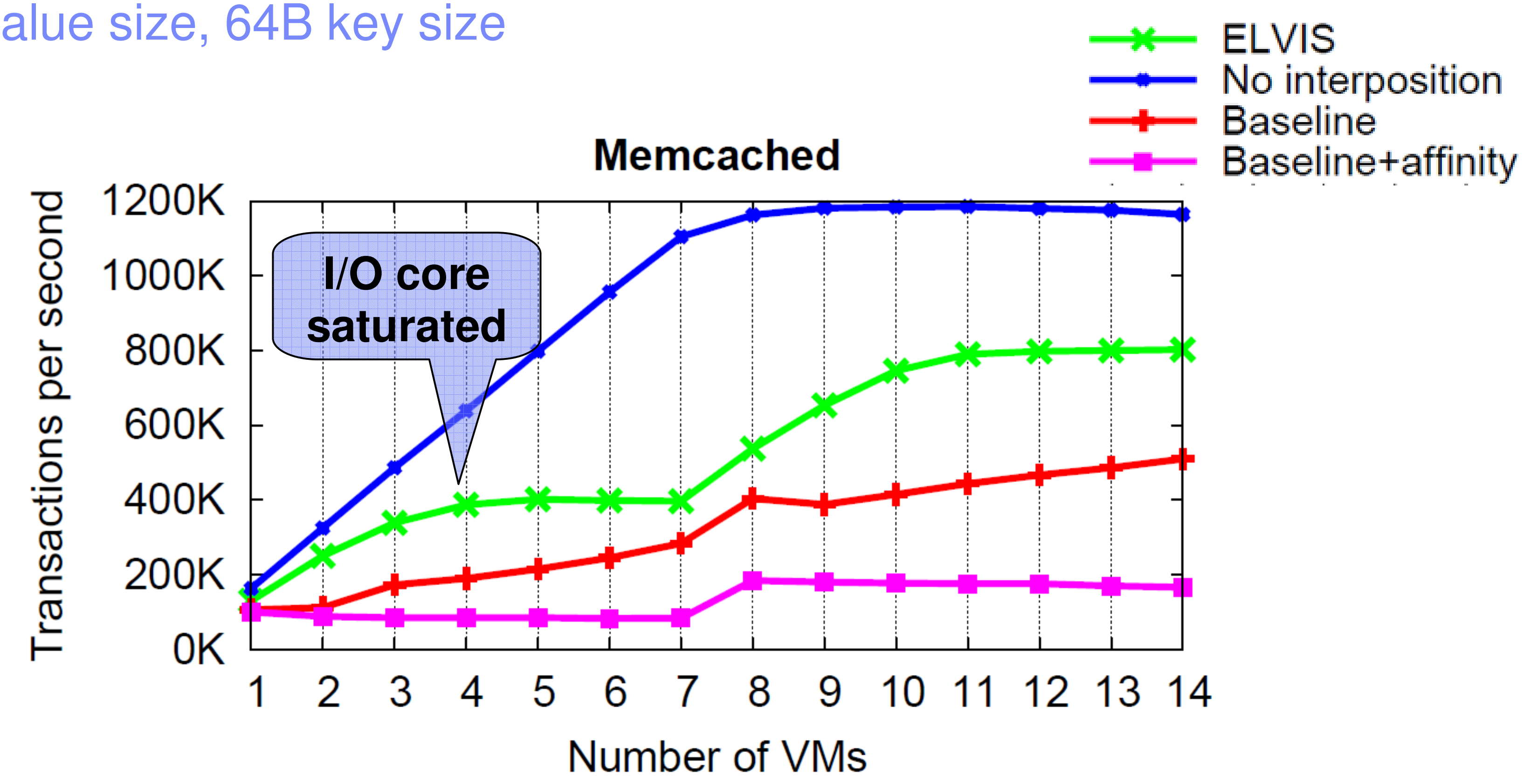
2x10Gb port
ELVIS: 2 cores dedicated for I/O and 1 dedicated core per VM (N+2 total)
Baseline: N+2 cores (to handle I/O and to run the VMs)
No Interposition: N cores to run the VMs

Netperf – UDP Request Response (latency sensitive)



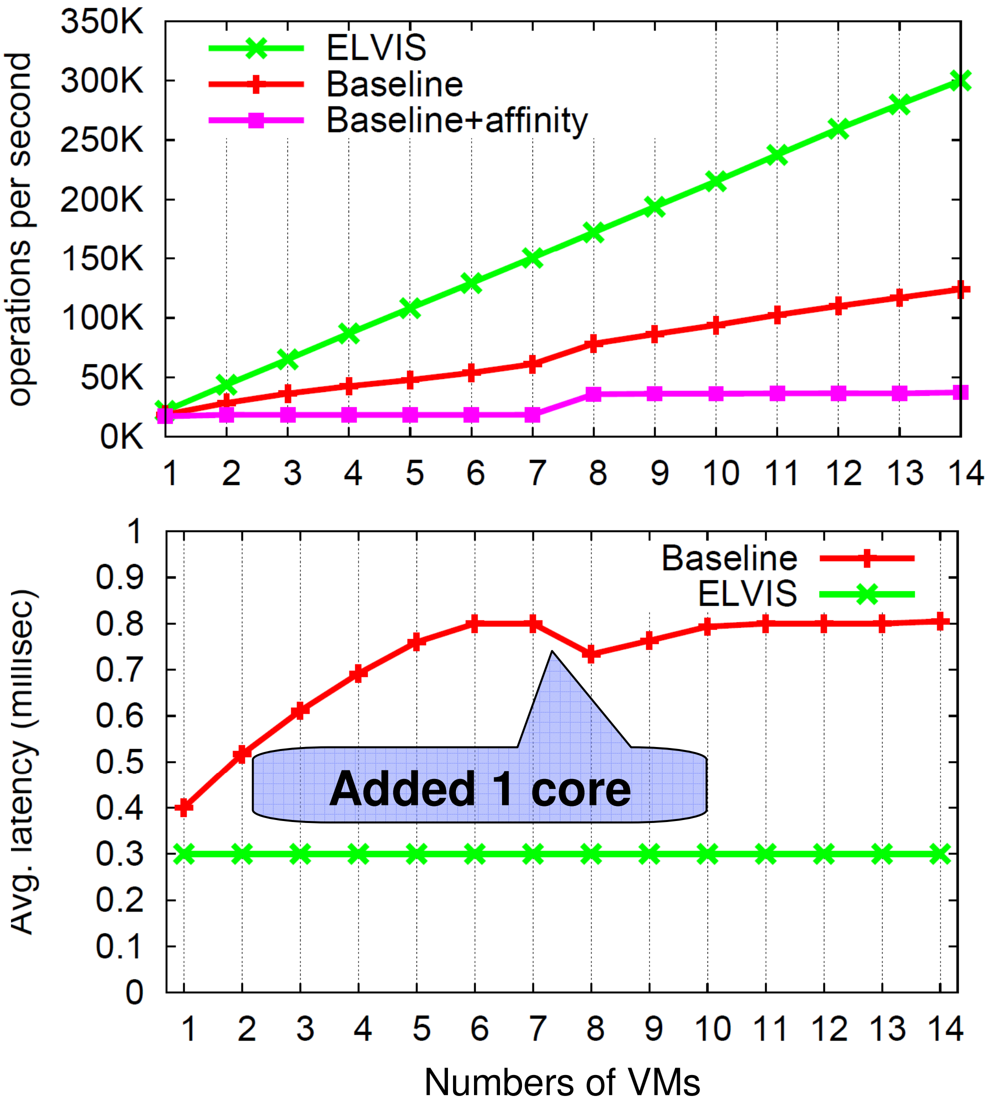
- Latency slightly increased with more VMs
- Better than No Interposition in some cases because enabling SR-IOV in the NIC increases latency by 22% (ELVIS disables SR-IOV)

Memcached - 90% get, 10% set, 32 concurrent requests per VM
1KB value size, 64B key size



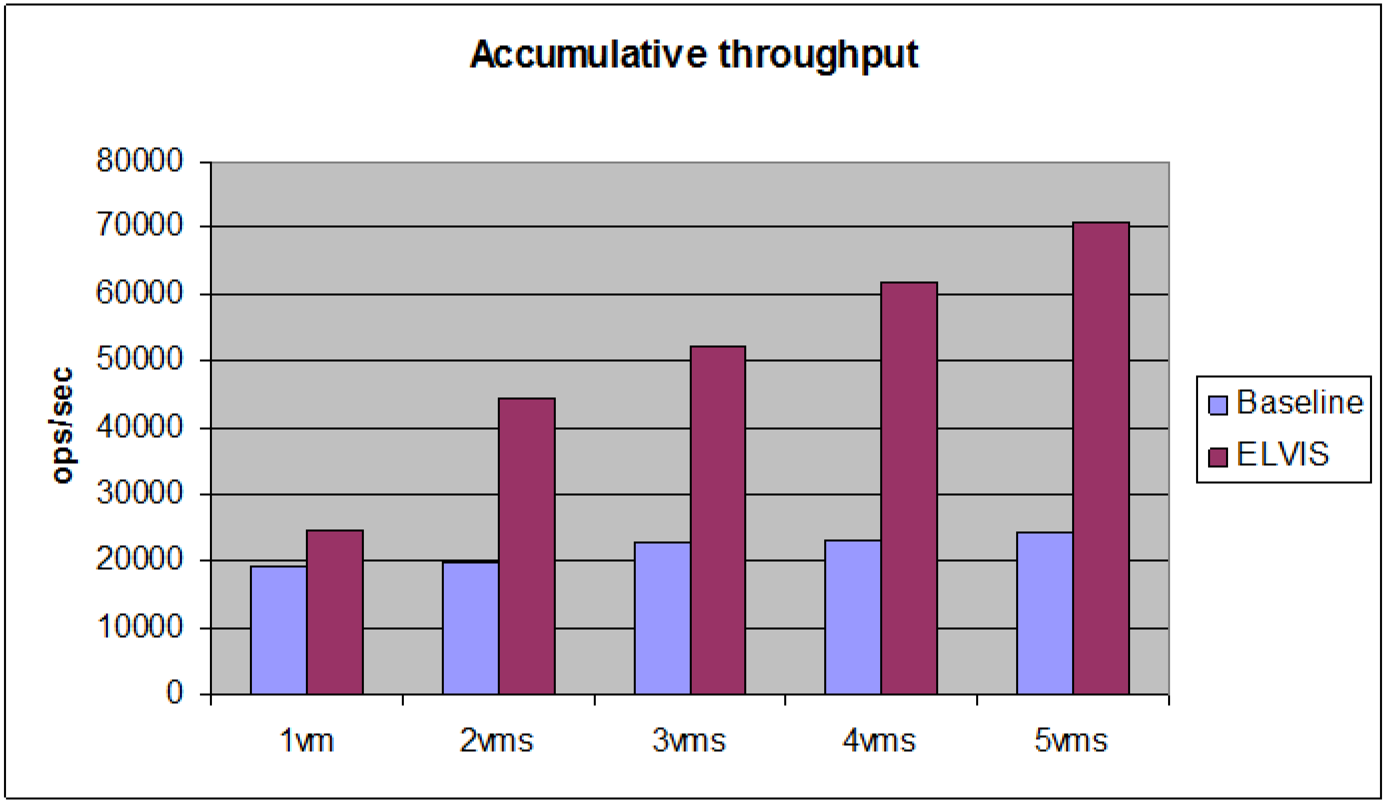
- I/O core saturated after 3 VMs
- ELVIS was up to 30% slower than No interposition when the I/O core was not saturated, but was always 30%-115% better than Baseline

Filebench – block I/O interposition based on host RAM disk 4x4KB random writes, 4x4KB random reads per VM



- Latency remains constant
- Throughput increases linearly

Filebench (*) - 4 threads performing 8KB random reads per VM using fusion-io (PCIe flash) as a block device for the VMs



Number of cores used = number of VMs + 1

(*) Evaluation performed by Razya Ladelsky <razyal@il.ibm.com> using a different machine setup, Kernel 3.9, QEMU 1.3, and vhost-block back-end shared by Asias He <asias@redhat.com>

Conclusions and Future Work

- Current trend towards multi-core systems, towards faster networks and block devices makes Virtio **inefficient** and **not scalable**

- ELVIS presents a new **efficient and scalable** model based on vhost

- Future Work
  - Mechanism to dynamically allocate or release I/O cores and map Virtio queues to I/O cores
  - Policy to monitor the system load, decide how many I/O cores are required and map queues to I/O cores

Questions ?