Optimizing the QEMU Storage Stack

Stefan Hajnoczi – stefan.hajnoczi@uk.ibm.com
Open Virtualization
IBM Linux Technology Center

2010

Agenda

- The QEMU storage stack
- QEMU architecture
- Virtio-blk and the request lifecycle
- Performance challenges today
- Instrumenting the storage stack
- Out-of-line I/O emulation using ioeventfd
- Reducing lock contention with unlocked kick
- Prototyping a threaded device model

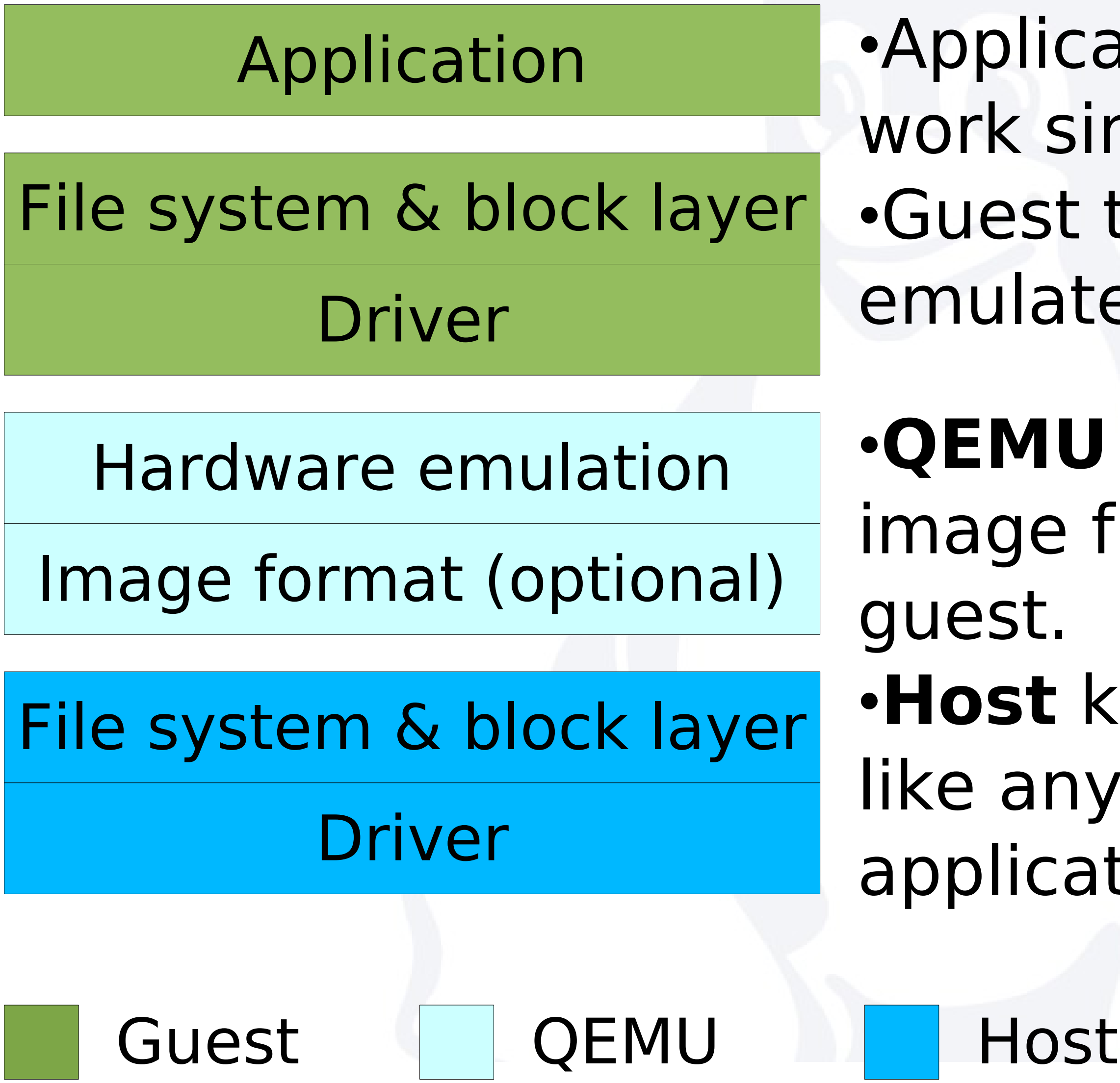
QEMU storage performance

- KVM and Xen have made Linux virtualization popular.
  - CPU vendors addressed performance challenges with hardware assist features.
  - Performance is good for CPU bound workloads, but I/O remains a challenge.
- Goal: Storage performance under virtualization should be **comparable to bare metal**.
  - Virtualization overhead must be minimized.
- Comparisons can be made by running benchmarks inside a virtual machine and directly on the host.
  - Need to be careful about fair apples-to-apples comparisons.

Virtualized storage approach in QEMU

- In virtualization, the hypervisor needs to **manage resources** between virtual machines.
- Bare metal does not have to do this because there are no shared resources.
- Two approaches:
  - Multiplexing resources (e.g. emulation). Control is in hypervisor, flexible, slow.
  - Passthrough or hardware assist (e.g. PCI device assignment). Hypervisor is involved less, less flexible, fast.
- Today's users mostly rely on multiplexed storage resources.
- Let's look at the QEMU **storage stack** to understand how storage is emulated.

The QEMU storage stack



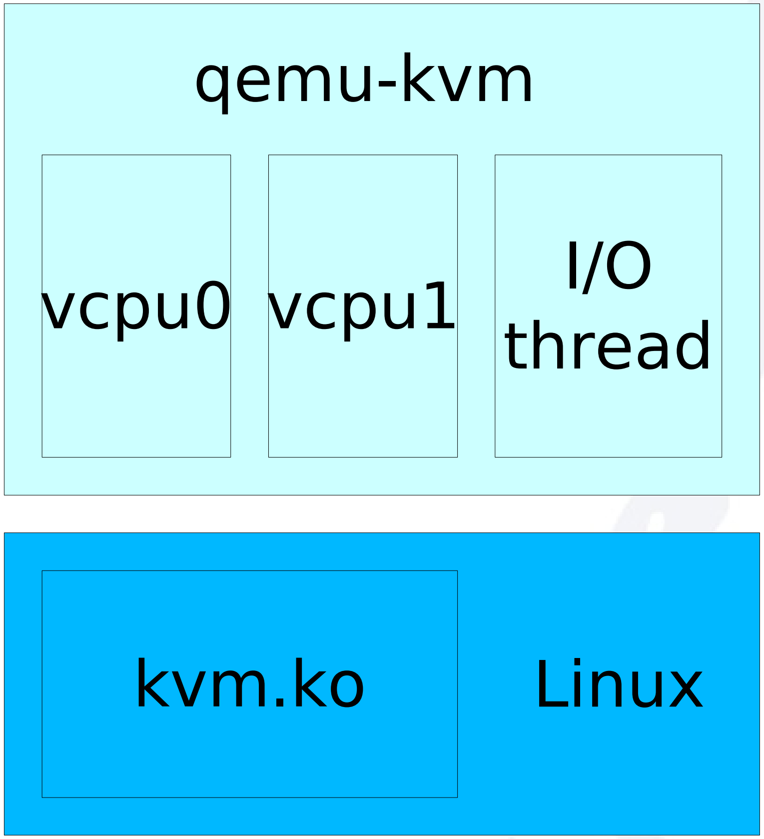
- Application and **guest** kernel work similar to bare metal.
- Guest talks to QEMU via emulated hardware.

- **QEMU** performs I/O to an image file on behalf of the guest.
- **Host** kernel treats guest I/O like any userspace application.

Seeing double

- There may be two **file systems**. The guest file system and the host file system (which holds the image file).
- There may be two **volume managers**. The guest and host can both use LVM and md independently.
- There are two **page caches**. Both guest and host can buffer pages from a file.
- There are two **I/O schedulers**. The guest will reorder or delay I/O but the host will too.
- Configuring either the guest or the host to bypass these layers typically leads to best performance.

QEMU Architecture



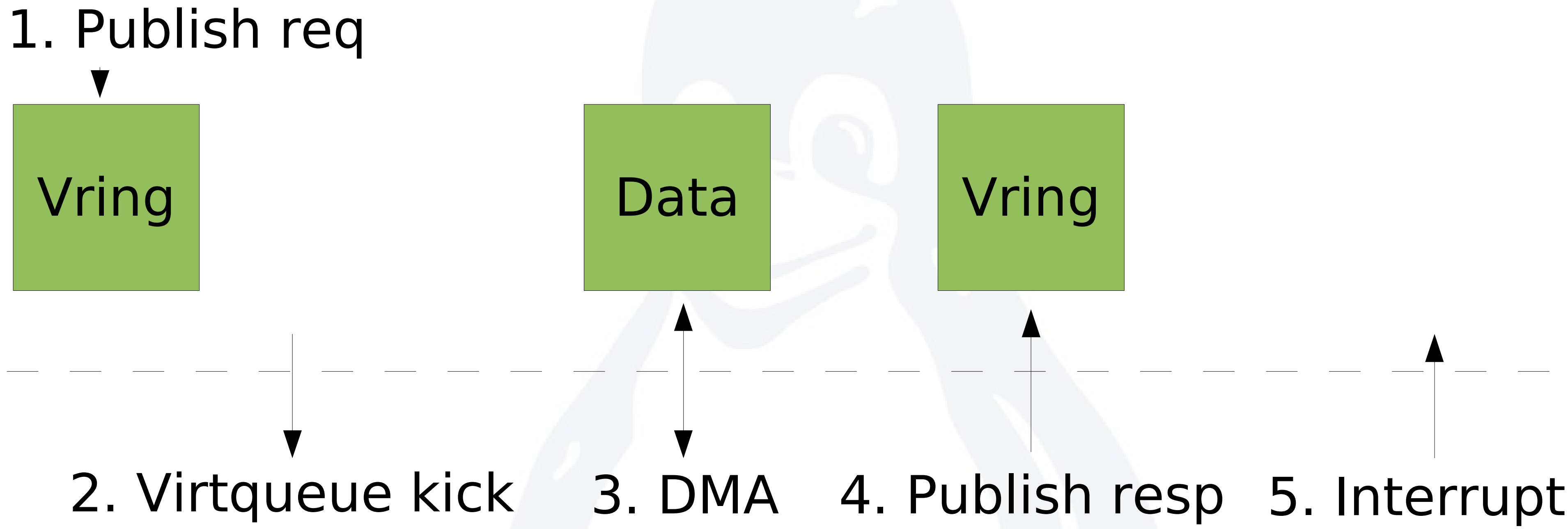
- Each guest CPU has a dedicated **vcpu thread** that uses the kvm.ko module to execute guest code.
- There is an **I/O thread** that runs a select(2) loop to handle events.

- Only one thread may be executing QEMU code at any given time. This excludes guest code and blocking in select(2).

Emulated storage

- QEMU presents emulated storage interfaces to the guest.
- **Virtio** is a paravirtualized storage interface, delivers the best performance, and is extensible for the future.
  - One virtio-blk PCI adapter per block device
- **IDE** emulation is used for CD-ROMs and is also available for disks.
  - Good guest compatibility but low performance
- **SCSI** emulation can be used for special applications but note virtio can do SCSI passthrough.

Virtio-blk request lifecycle



- Request/response data and metadata live in guest memory.
- Virtqueue kick is a pio write to a virtio PCI hardware register.
- Completion is signaled by virtio PCI interrupt.

Symptoms of poor performance

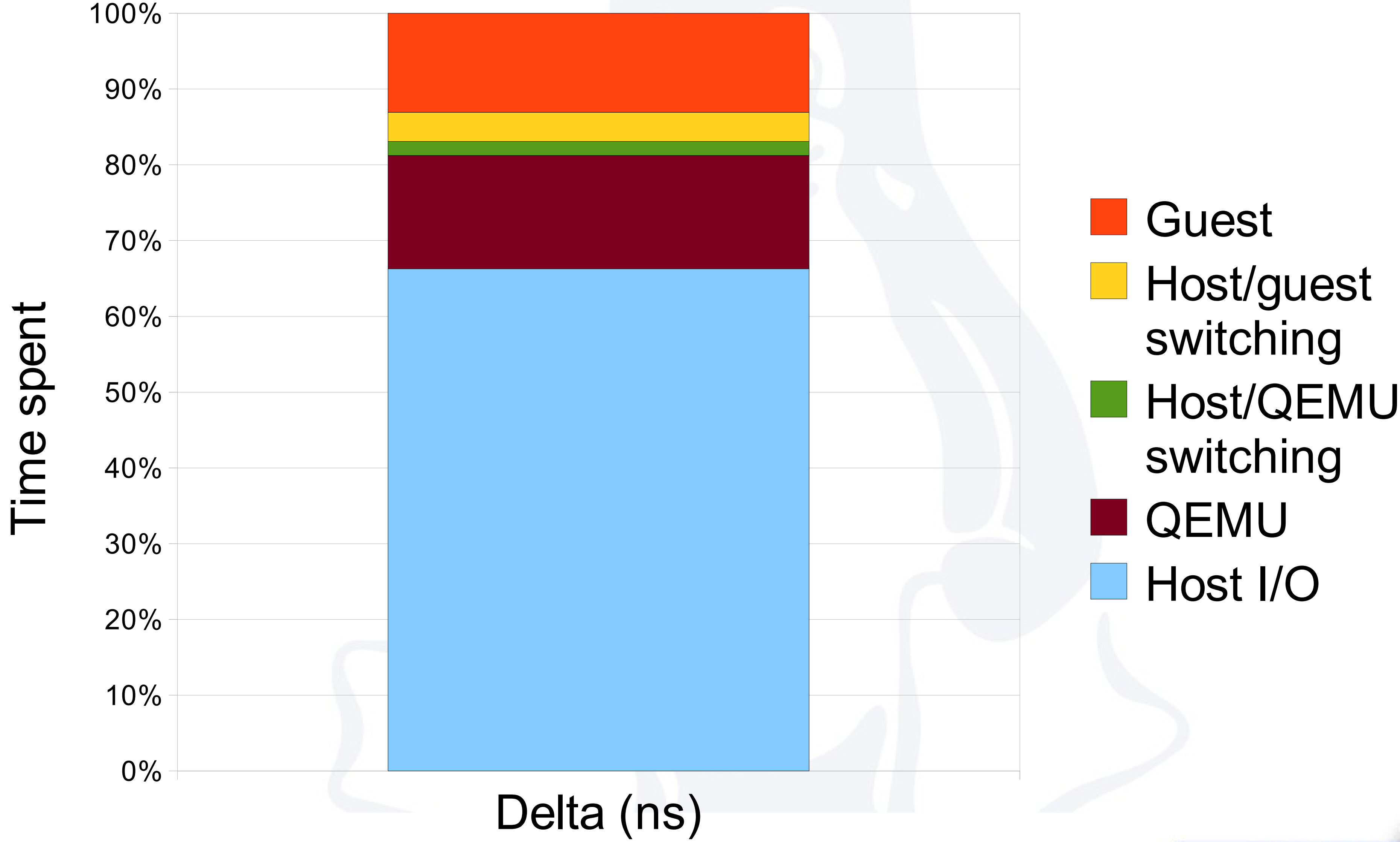
- Low **throughput** compared to bare metal.
  - <40% of bare metal: **fix your configuration**
  - 40-75%: legitimate configuration that needs optimizations in QEMU and Linux
- High guest **CPU utilization** due to disk I/O.
- **High latency** compared to bare metal.
  - Matters most for synchronous applications.
- Investigate by **instrumenting the stack**.

Instrumenting the storage stack

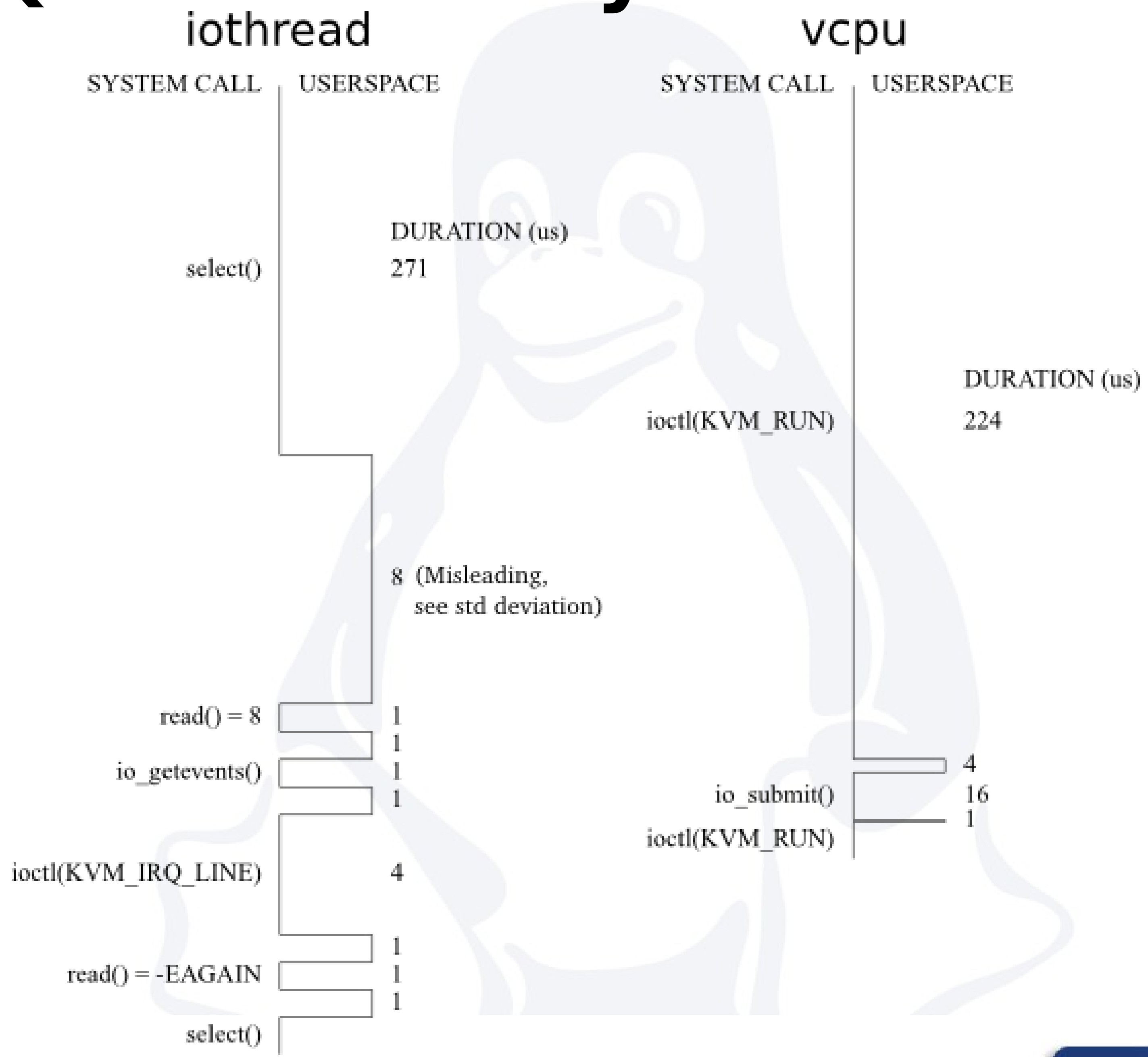
- Goal: Identify latency overheads imposed by the QEMU storage stack.
- Linux and QEMU tracing mechanisms allow **lightweight logging**.
- **Timestamps** reveal how much time was spent in each layer of the stack.
- Challenges:
  - Combining traces from different sources (guest, QEMU, host kernel).
  - Reliable timing across host/guest boundary.
- For details and git branch: http://www.linux-kvm.org/page/Virtio/Block/Latency

Sequential 4k read latency

1 vcpu, 4 GB RAM, x2apic, virtio-blk cache=none guest
2x4-core, 8 GB RAM, 12 LVM striped LUNs over FC
kvm.git host kernel, qemu-kvm.git 0.12.4



QEMU latency timeline

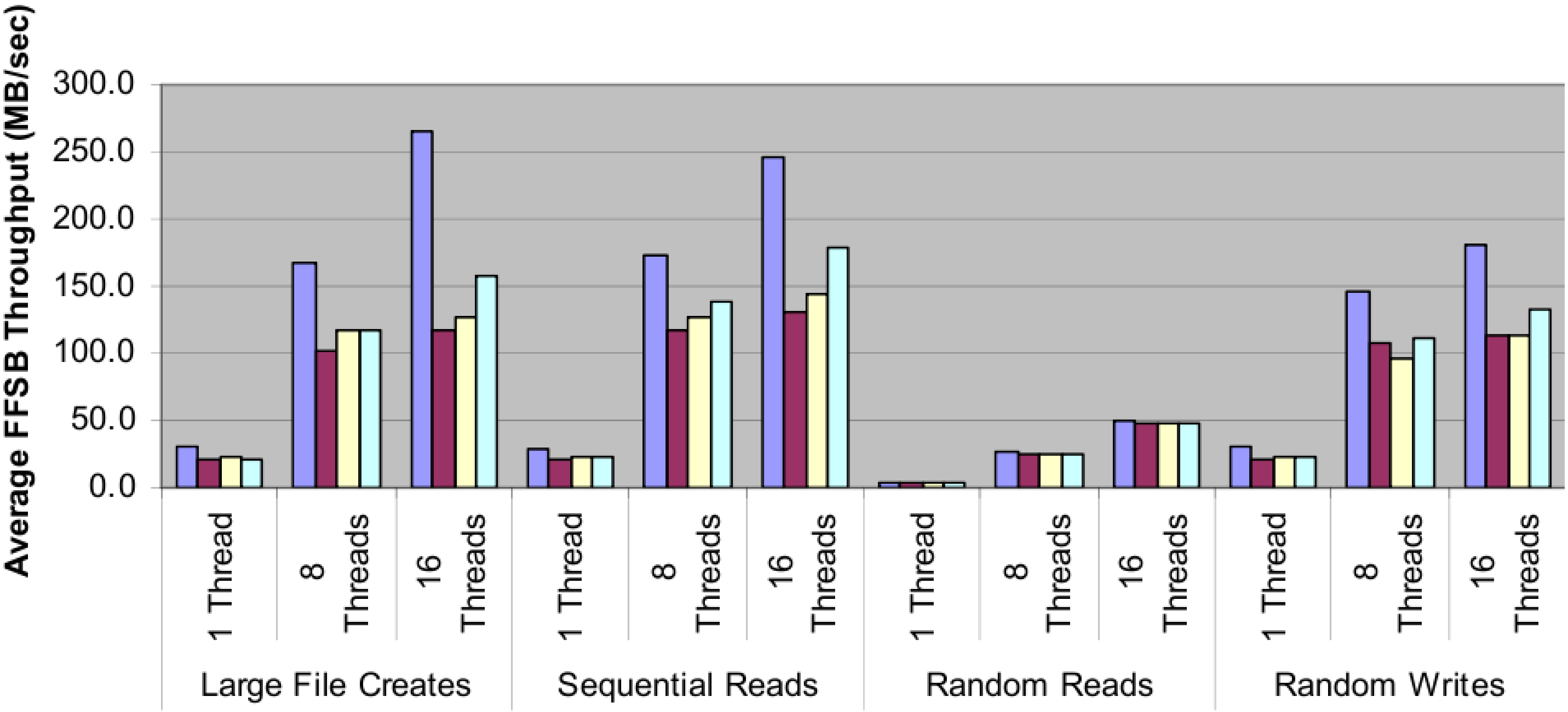


virtio-ioeventfd

- Vcpu thread should be running guest code and not QEMU code!
- **Stealing time** from the guest has consequences:
  - High guest system time (symptom)
  - **Lock contention** for SMP guests
- Use ioeventfd to decouple for virtqueue kicks from vcpu thread execution.
- This is the model used by vhost-net for in-kernel virtio-net emulation.
- Takes advantage of spare cycles on host.
- Potentially has overhead on a fully loaded host.
  - Needs to be looked at with vhost-net too.

Virtio-ioeventfd results

- cache=none, x2apic, aio=native, 2.6.32, raw on ext4. Blue=bare metal, yellow=unmodified KVM, green=virtio-ioeventfd



unlocked-kick

- For SMP guests, the virtio-blk spinlock is at the top of CPU profiles and lockstat.
- 2 vcpu guest, virtio-blk on 16 core host:

```
# Events: 456K cycles #
# Overhead Command Shared Object Symbol #
........ ............... ......................... ...... #
51.94% qemu-kvm [guest.kernel.kallsyms] [g] .text.lock.spinlock
2.59% qemu-kvm 3b698bb8d2 [u] 0x00003b698bb8d2
1.13% qemu-kvm [guest.kernel.kallsyms] [g] __blockdev_direct_IO
1.10% qemu-kvm [guest.kernel.kallsyms] [g] __find_get_block
1.03% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_free
1.03% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_alloc
0.83% qemu-kvm [ext3] [g] __ext3_get_inode_loc
0.82% qemu-kvm [jbd] [g] do_get_write_access
0.74% qemu-kvm [jbd] [g] journal_add_journal_head
0.73% qemu-kvm [guest.kernel.kallsyms] [g] __make_request
0.58% qemu-kvm [ext3] [g] ext3_mark_iloc_dirty
0.58% qemu-kvm [guest.kernel.kallsyms] [g] _spin_lock
0.57% qemu-kvm [guest.kernel.kallsyms] [g] ioread8
0.56% qemu-kvm [guest.kernel.kallsyms] [g] schedule
0.56% qemu-kvm [guest.kernel.kallsyms] [g] radix_tree_lookup
0.54% qemu-kvm [guest.kernel.kallsyms] [g] kfree
0.54% qemu-kvm [guest.kernel.kallsyms] [g] bit_waitqueue
```

unlocked-kick

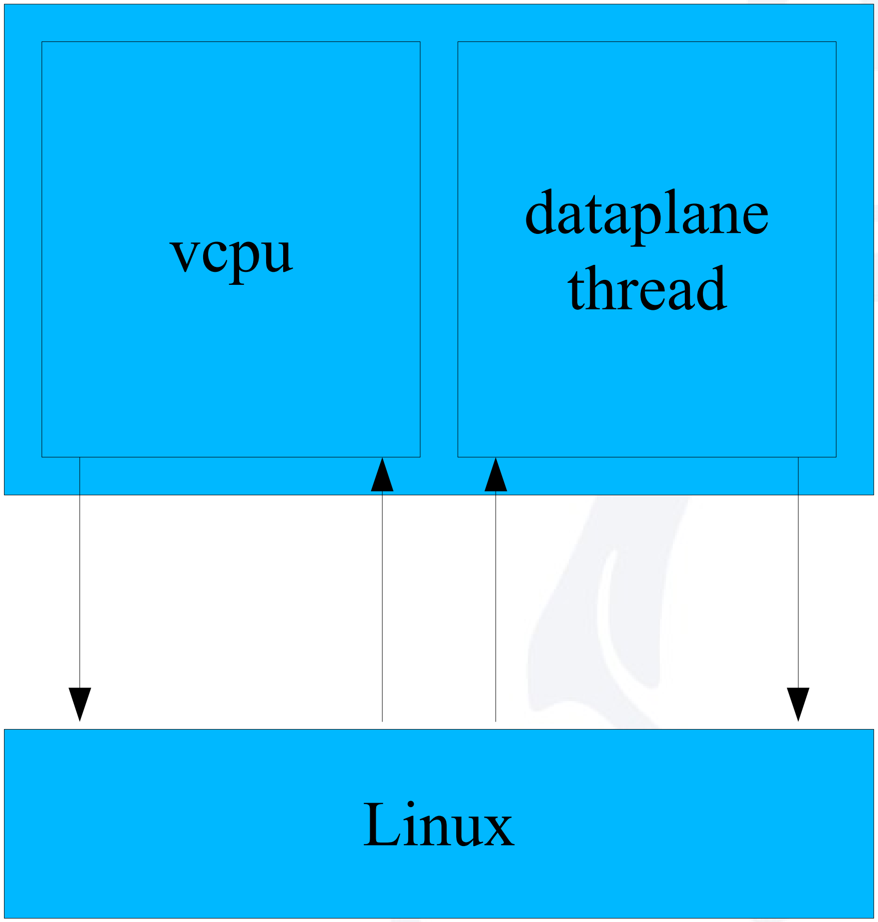
- The block layer allows driver to release block queue lock in its request processing function.
- This avoids spinning other vcpus:

```
# Events: 293K cycles #
# Overhead Command Shared Object Symbol
# ........ ............... ......................... ...... #
5.65% qemu-kvm 3b6787aaa9 [u] 0x00003b6787aaa9
3.73% qemu-kvm [guest.kernel.kallsyms] [g] .text.lock.spinlock
2.19% qemu-kvm [guest.kernel.kallsyms] [g] __blockdev_direct_IO
2.14% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_free
2.13% qemu-kvm [guest.kernel.kallsyms] [g] __find_get_block
2.00% qemu-kvm [guest.kernel.kallsyms] [g] kmem_cache_alloc
1.63% qemu-kvm [ext3] [g] __ext3_get_inode_loc
1.62% qemu-kvm [jbd] [g] do_get_write_access
1.57% qemu-kvm [jbd] [g] journal_add_journal_head
1.46% qemu-kvm [guest.kernel.kallsyms] [g] _spin_lock
1.17% qemu-kvm [guest.kernel.kallsyms] [g] schedule
1.17% qemu-kvm [ext3] [g] ext3_mark_iloc_dirty
1.09% qemu-kvm [guest.kernel.kallsyms] [g] iowrite16
1.08% qemu-kvm [virtio_ring] [g] vring_kick
1.06% qemu-kvm [guest.kernel.kallsyms] [g] radix_tree_lookup
1.06% qemu-kvm [guest.kernel.kallsyms] [g] bit_waitqueue
1.06% qemu-kvm [guest.kernel.kallsyms] [g] kfree
```

Virtio-blk dataplane

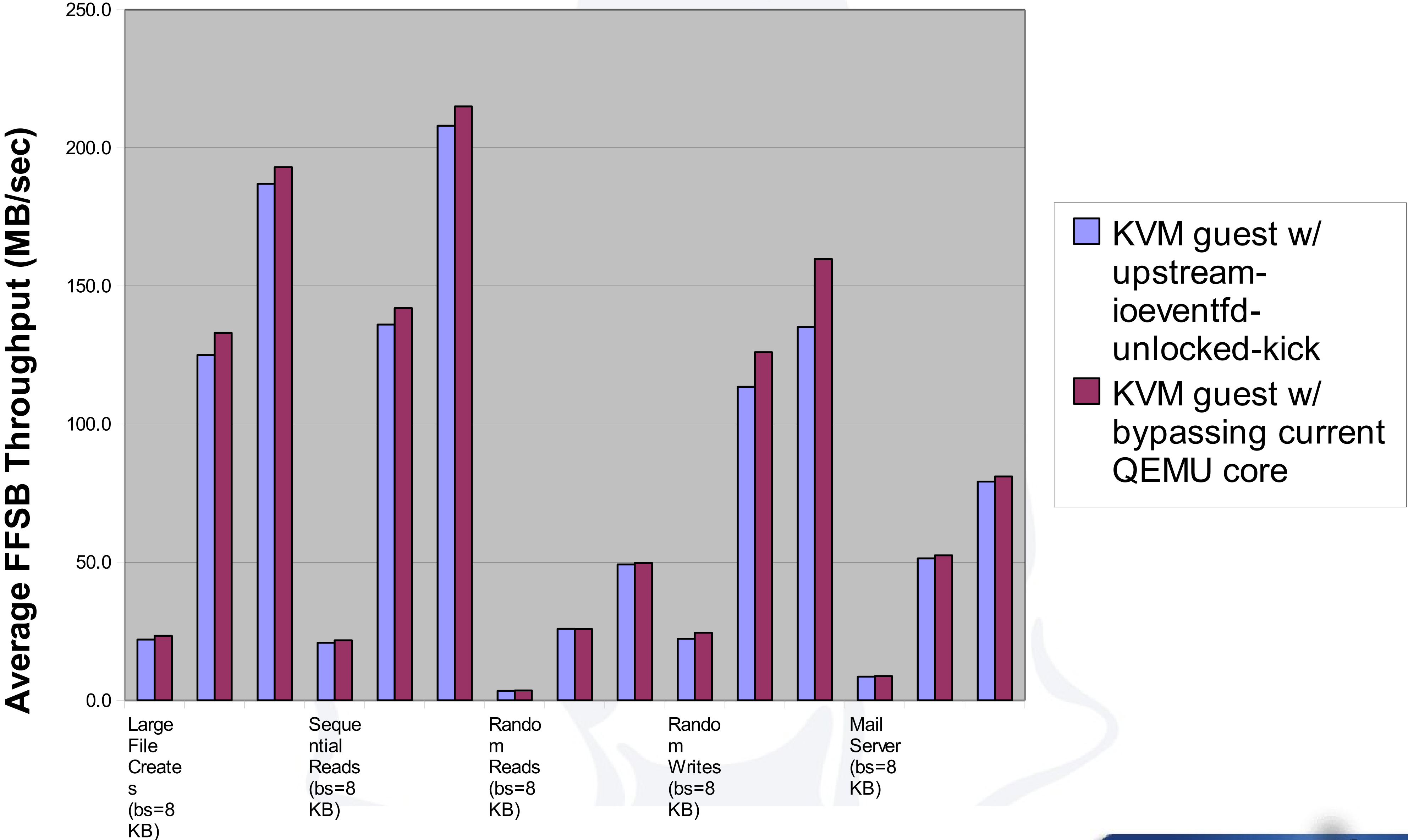
- Experiment to build a **dedicated thread per virtio-blk device** outside of QEMU's global mutex.
  - No global mutex, better scalability
  - Proof of concept for a threaded device model
- Rewrite virtio-blk emulation without dependencies on QEMU core code (not thread-safe).
- Only supports raw image format because other formats have state and dependencies on QEMU core.
- Git branch: http://repo.or.cz/w/qemu/stefanha.git/shortlog/refs/heads/virtio-blk-data-plane

Dataplane model



- Virtqueue kicks from guest are delivered to dataplane thread using ioeventfd.
- Completion interrupt injected from dataplane thread using ioctl to kvm.ko.

Dataplane FFSB Results
KVM Guest = 2 vcpus, 4GB; KVM Host = 16 cpus, 12GB; Linux 2.6.32
KVM configuration = virtio-blk, no cache, aio=native
DS3400 Storage w/ 8 x 24-disk RAID10 Arrays, ext4 (no barrier)



Thank you