libblkio

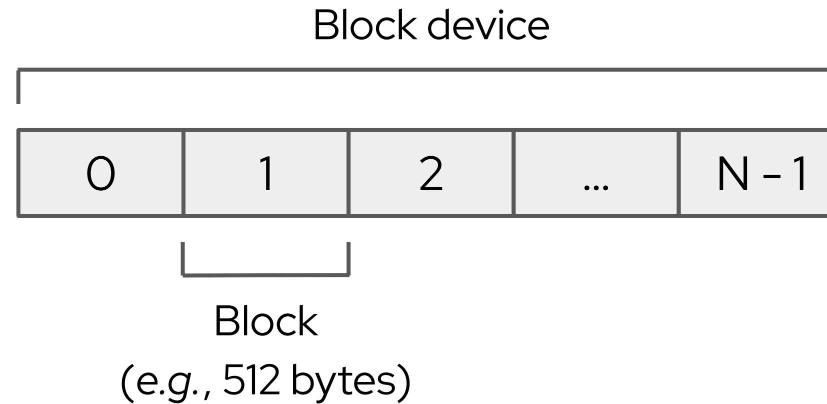
Introducing the libblkio High-performance Block I/O API

Stefan Hajnoczi
stefanha@redhat.com

Alberto Faria
afaria@redhat.com

Red Hat

Block devices



NVMe, SCSI, ATA, virtio-blk follow the block device model

Read & **Write** access data in units of blocks

Flush persists previously written data to permanent storage

Discard (TRIM) and **Write Zeroes** manage block allocation

Red Hat

Where block device are used

Databases

- MySQL & MariaDB

File Systems

- Ceph Bluestore
- XFS, btrfs, etc

Emulators & Hypervisors

- QEMU

I/O Frameworks & Software-defined Storage

- SPDK

Backup, Forensics, & Disk Imaging Tools

- casync
- qemu-img
- mkfs

Red Hat

How libblkio came about

QEMU accumulated non-trivial block drivers

- io_uring, NVMe userspace driver, ...

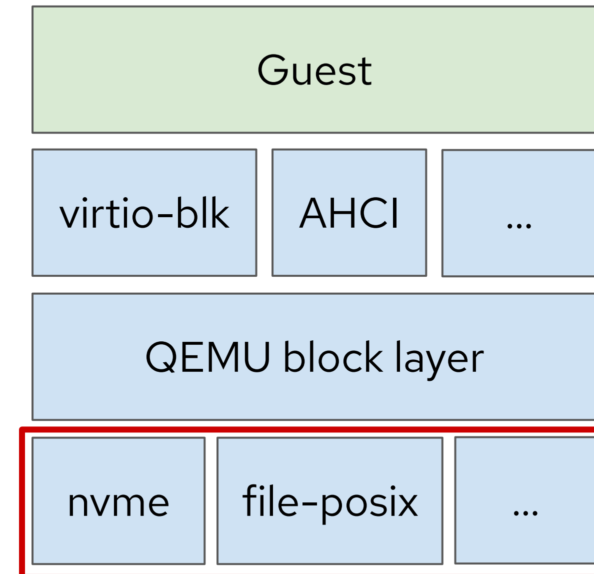
More drivers were needed:

- virtio-blk-vhost-vpda
- virtio-blk-vhost-user

QEMU block drivers are only usable within QEMU

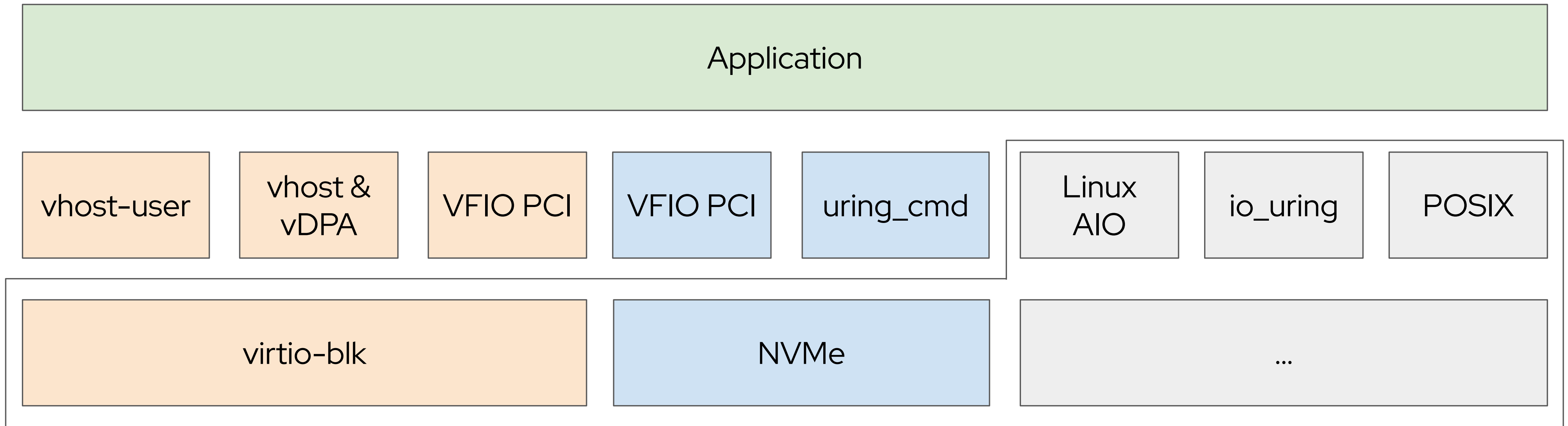
- Unable to reuse code in other programs

Decided to develop new drivers as a library instead



Red Hat

Block I/O interfaces have proliferated!

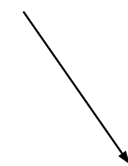


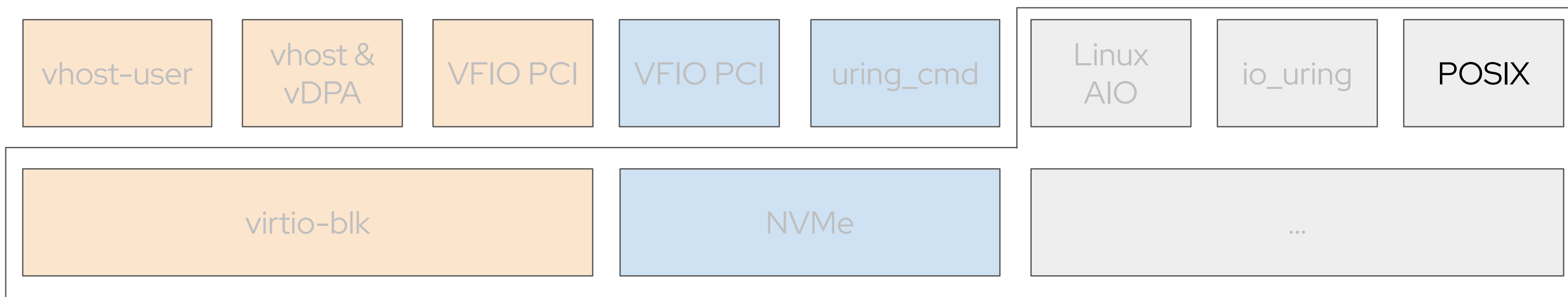
How many can your application support?

Red Hat

Why use different interfaces?

POSIX is simple and device-agnostic

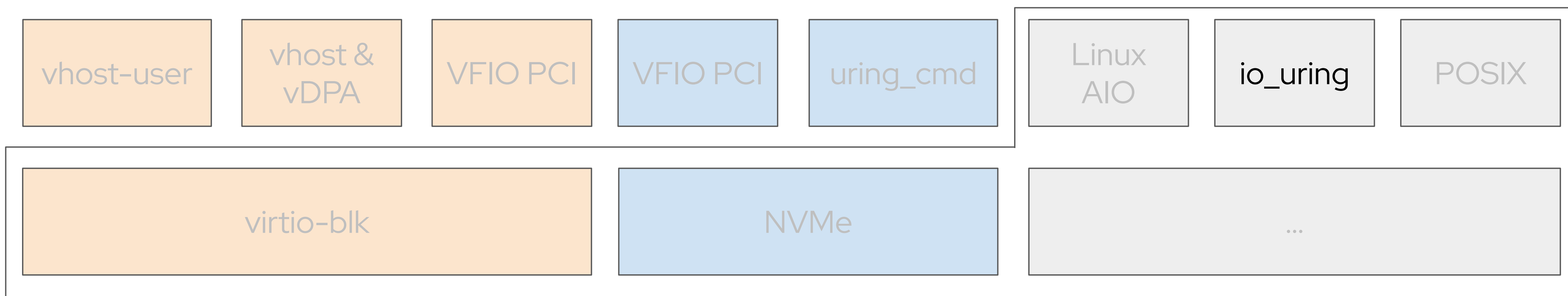




Red Hat

Why use different interfaces?

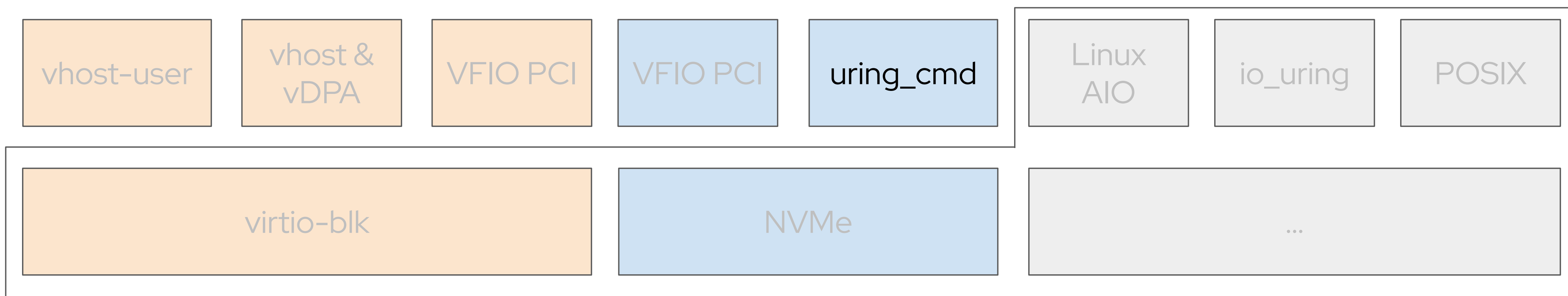
io_uring is asynchronous, less syscall overhead



Red Hat

Why use different interfaces?

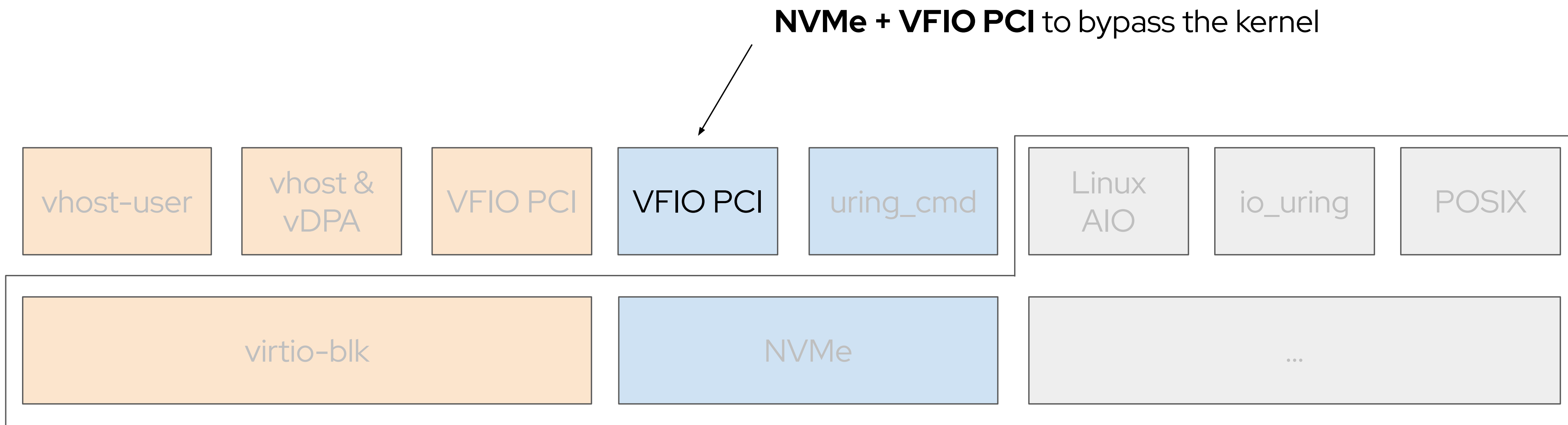
NVMe + uring_cmd to bypass the VFS



Red Hat

Why use different interfaces?

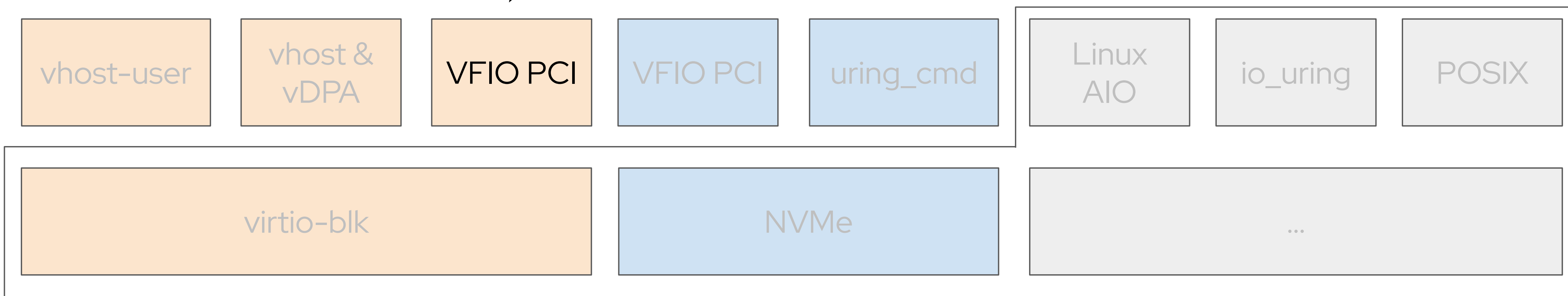
NVMe + VFIO PCI to bypass the kernel



Red Hat

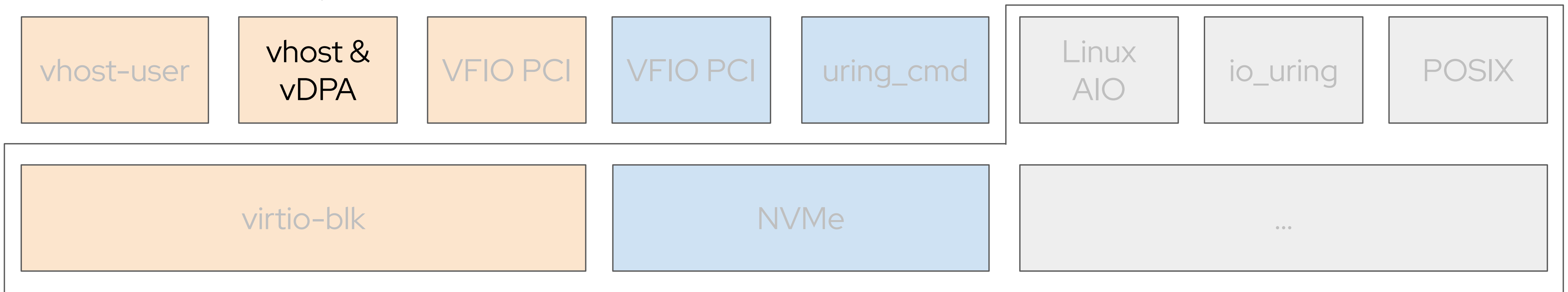
Why use different interfaces?

virtio-blk + VFIO PCI to bypass
the guest kernel's virtio-blk driver



Red Hat

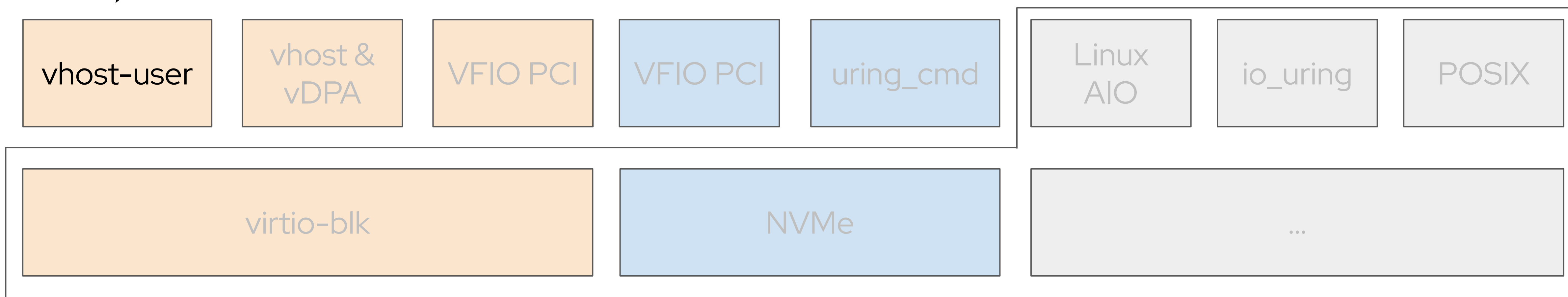
Why use different interfaces?



Red Hat

Why use different interfaces?

virtio-blk + vhost-user to communicate with a user-space storage server (*e.g.*, SPDK, qemu-storage-daemon)



Red Hat

Are block I/O interfaces similar?

Same:

- Read, write, flush, discard, write zeroes
- Requests
- Queues

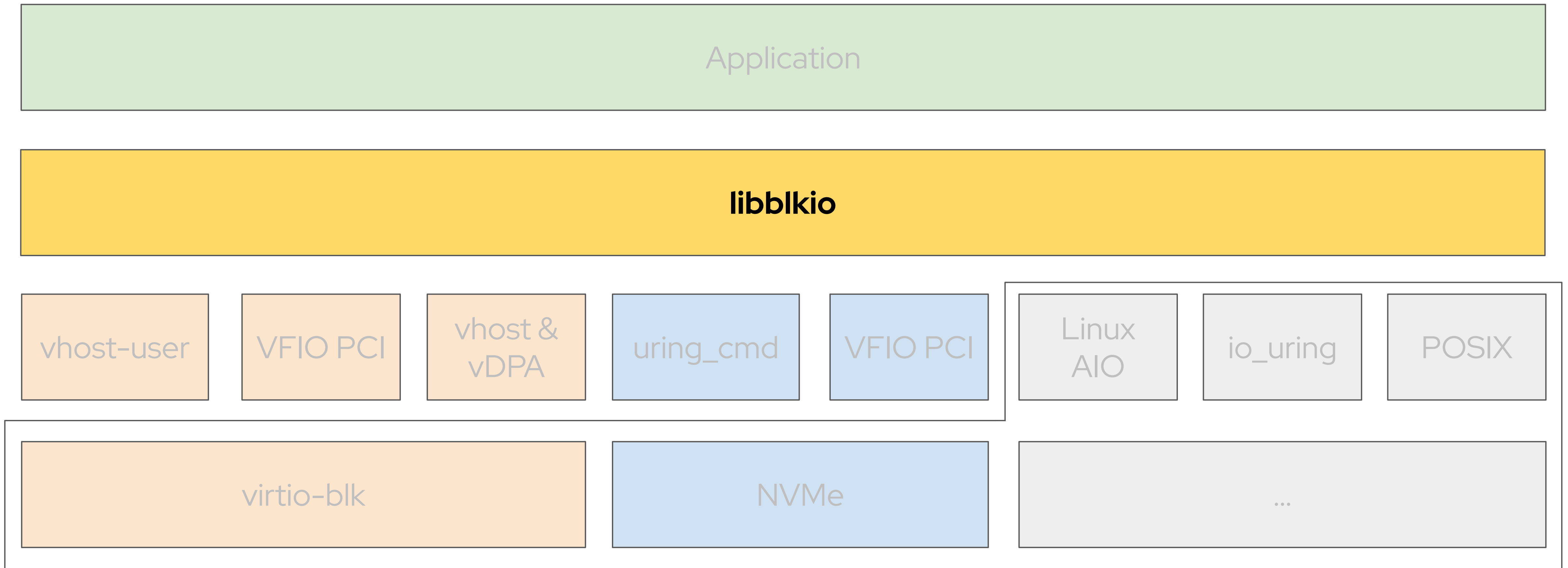
Different:

- Synchronous or asynchronous
- Polling support
- I/O buffer memory constraints
- Queue memory layout
- Exact semantics and API details
- ...

How much effort is it to integrate a new interface into your application?

Red Hat

libblkio provides a unified block I/O interface



Red Hat

Where **libblkio** can be used

Databases

- MySQL & MariaDB

File Systems

- Ceph Bluestore
- XFS, btrfs, etc

Emulators & Hypervisors

- QEMU

I/O Frameworks & Software-defined Storage

- SPDK

Backup, Forensics, & Disk Imaging Tools

- casync
- qemu-img
- mkfs

Red Hat

libblkio

- C API (implemented in Rust)
- Provides several “**drivers**”
  - Each driver talks to a different underlying block I/O interface
- All drivers provide the **same API**
  - No code changes necessary to use a different driver

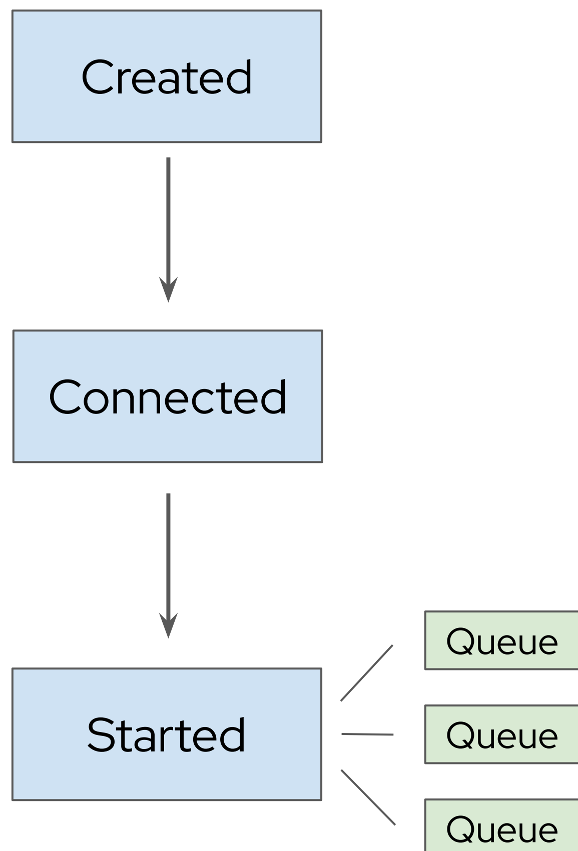
Red Hat

libblkio

- Includes drivers for all environments
  - "*io_uring*", useful on **bare metal**, in **containers**, in **VMs**
  - "*virtio-blk-vfio-pci*", useful to bypass guest kernel drivers in **VMs**
  - "*virtio-blk-vhost-vdpa*", useful to split a physical virtio-blk device into many virtualized devices, *e.g.*, for **containerized** environments
  - ...

Red Hat

Lifecycle



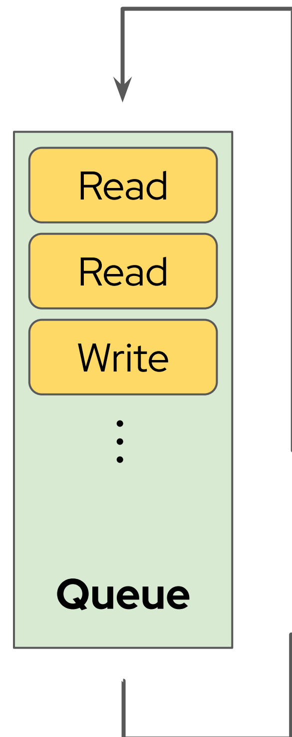
1. blkio_create(“io_uring”)

2. blkio_set_str(“path”, “disk.img”)
blkio_connect()

3. blkio_set_int(“num-queues”, 3)
blkio_start()

Red Hat

Queues



- Queues are independent of each other
    - Commonly assigned to a thread/core

- Workflow:

1. Enqueue requests
2. Submit requests and await completions
3. Process completions

Red Hat

Queues

- No request **ordering**
  - User must await completion before submitting next request to establish ordering
- Can **enqueue** any number of requests
  - But not all may be **in-flight** simultaneously
  - Drivers allow configuring this internal limit

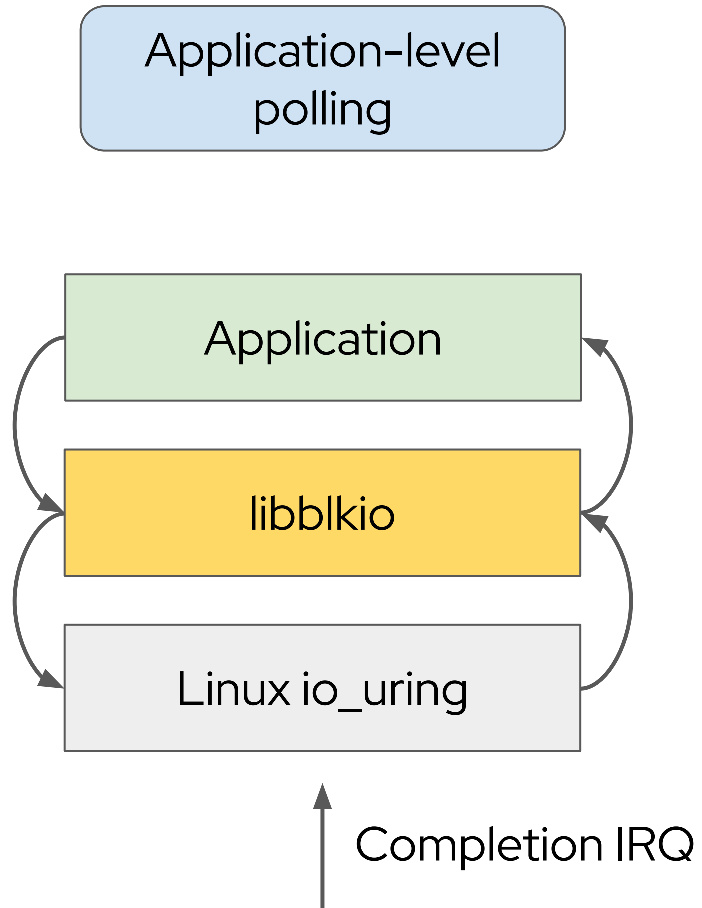
Red Hat

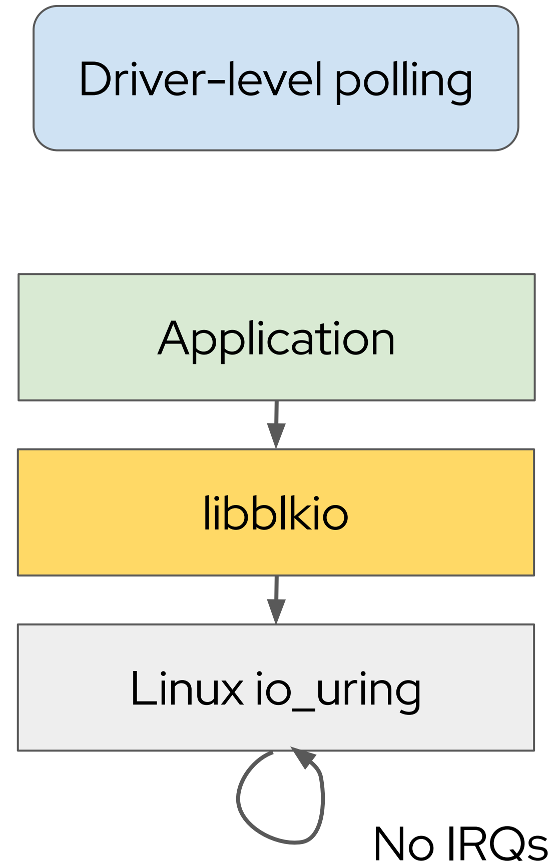
I/O modes

Blocking I/O	Event-driven I/O	Polled I/O
1. Enqueue & submit requests	1. Enqueue & submit requests	1. Enqueue & submit requests
2. **Block** waiting for completions	2. Read/poll **eventfd**	2. **Loop** checking for completions
3. Process completions	3. Process completions	3. Process completions

Red Hat

Polling modes





Red Hat

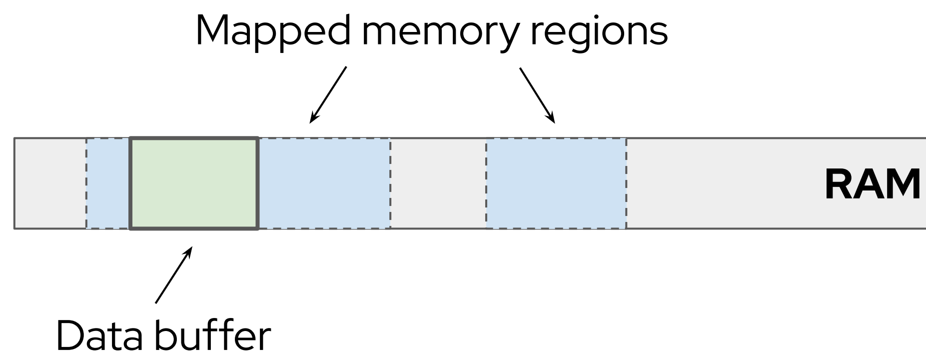
Block limits and properties

- Devices/drivers may impose restrictions on requests
- Named ***properties*** expose this information
  - “max-transfer” – Maximum read/write request size
  - “buf-alignment” – Mandatory data buffer offset/size alignment
  - …
- *Writable* properties are used for configuration
  - “path” – Device path for several drivers
  - “num-queues” – Number of queues to create
  - …

Red Hat

Memory regions

- Some drivers require **pre-registering** memory for data buffers
  - *e.g.*, for establishing IOMMU mappings
- libblkio provides a unified ***memory region*** abstraction for this
  - Can be mapped/unmapped dynamically
  - Drivers may require data buffers to belong to mapped regions



Red Hat

Memory regions

- There may be a limit on **concurrently** mapped regions
  - Mapping/unmapping might not be cheap
  - Ideally would map once and use many times
- Drivers may impose further restrictions on data buffers
  - Memory alignment, file descriptor-backed memory, …
- Utilities for **allocating** suitable memory are provided

Red Hat

Feature summary

- Unified, multi-queue block I/O API
- Blocking I/O, event-driven I/O, polled I/O
  - Fits your app’s I/O model
- Properties
- Memory regions

- Drivers are modular
  - Low integration effort
  - Can be contributed to https://gitlab.com/libblkio/libblkio

Red Hat

Case study & evaluation

Red Hat

Case study: libblkio in QEMU

- QEMU (https://www.qemu.org/) is an emulator with a full block layer
- New QEMU block driver adds:
  - `-blockdev` **`io_uring`**`,filename=test.img,...`
  - `-blockdev` **`virtio-blk-vhost-user`**`,path=vhost-user-blk.sock,...`
  - `-blockdev` **`virtio-blk-vhost-vdpa`**`,path=/dev/vhost-vdpa-0,...`
- ~700 source lines of code (SLOC)
  - Applications typically need less code
- Expected in QEMU 7.2 release

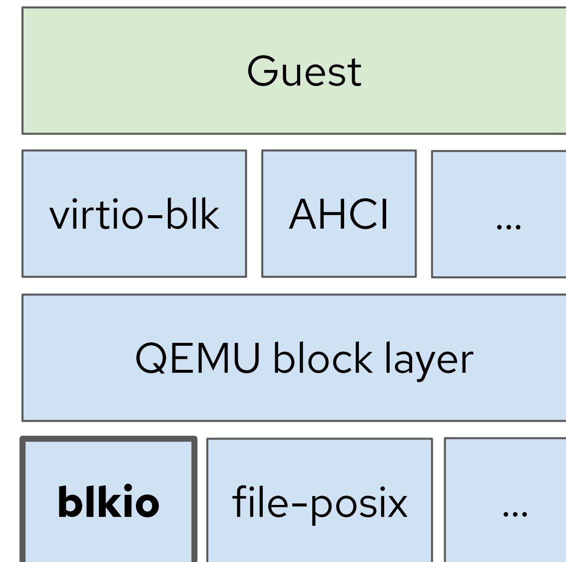
Red Hat

QEMU block drivers

Guests (VMs) submit I/O requests to *emulated storage controllers*.

The block layer hands requests to drivers.

Block driver integrates libblkio with QEMU.



See Jeff Cody's talk for info on developing QEMU block drivers:
http://bugnik.us/kvm2013/

Red Hat

I/O buffer memory in QEMU

Guest RAM

Guest RAM is long-lived with occasional hotplug.

Aligned heap

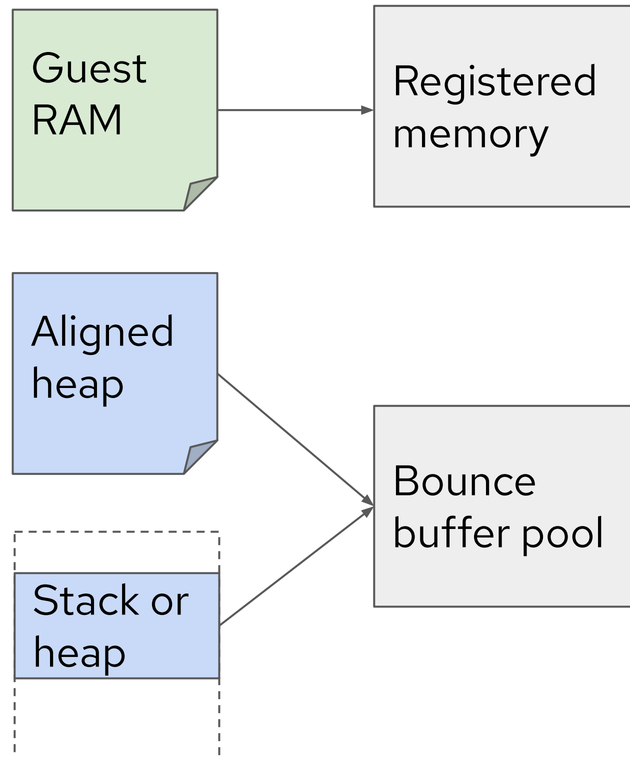
Block drivers like crypto or qcow2 allocate internal I/O buffers.

Stack or heap

Small I/O from within QEMU may be arbitrary stack or heap memory.

Red Hat

Mapping QEMU I/O buffers



Guest RAM is permanently mapped to libblkio.

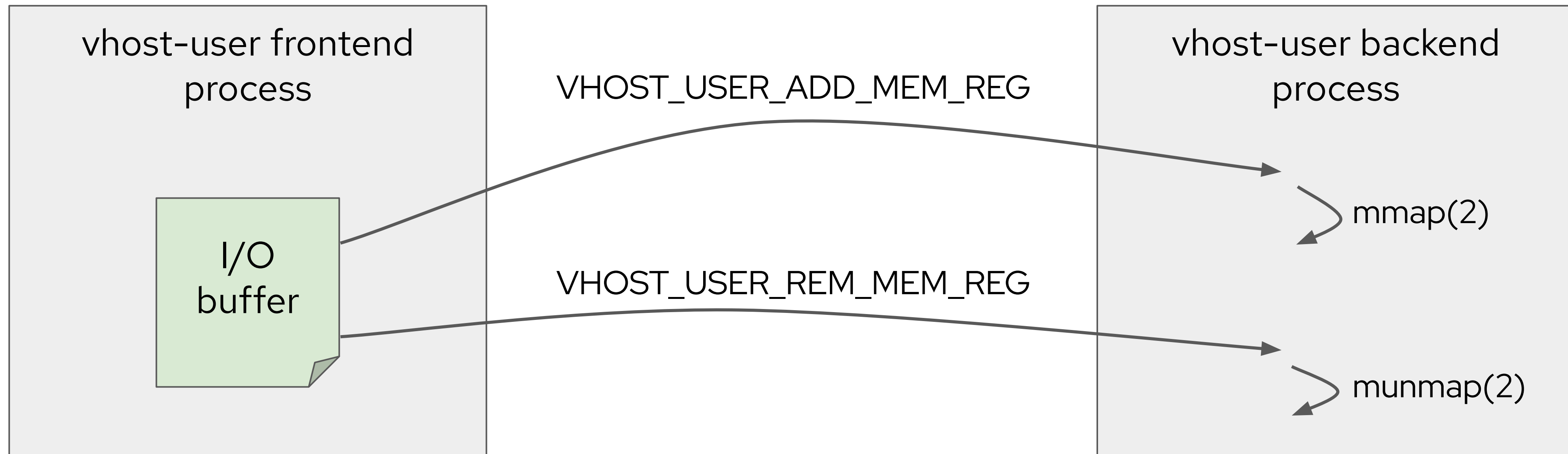
Bounce buffer pool is permanently mapped to libblkio

- Incurs copy overhead
- Alternative: temporary mappings?
- Alternative: intercept heap buffer allocation?

Existing applications need to make similar decisions about mapping I/O buffer memory.

Red Hat

vhost-user map/unmap operation



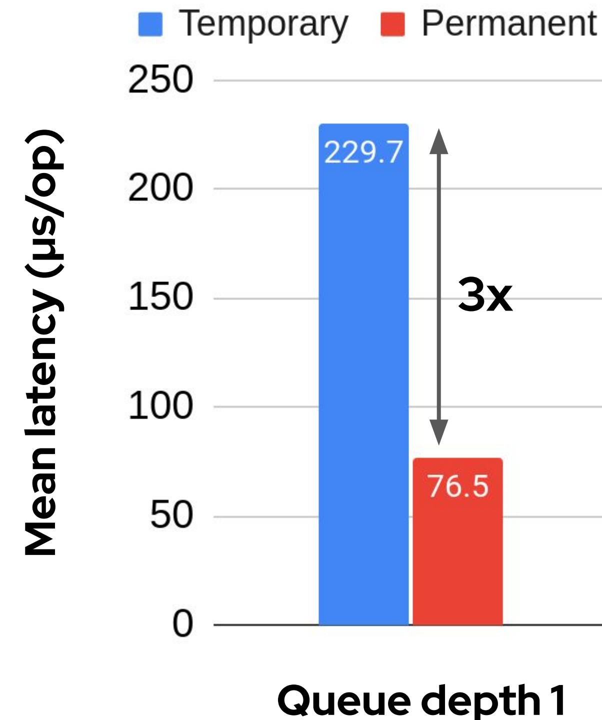
Mapping involves file descriptor passing over an AF_UNIX socket.

Red Hat

Measuring vhost-user map/unmap overhead

- Permanent mapping ***vs*** temporary mapping
  - 1 vCPU, 512-byte random reads
  - Mean latency (**lower is better**)
- qemu-storage-daemon vhost-user-blk export
- Out-of-the-box config without guest & vhost-user-blk polling

The I/O buffer mapping strategy is important!



Workload: 512-byte random reads, direct I/O, mean of 5 runs, 60 seconds per run
Hardware: Intel® Xeon® Silver 4214 CPU, Intel® Optane™ SSD DC P4800X Series (375 GB)
Software: Fedora 36 w/ Linux 5.19.1, qemu-storage-daemon vhost-user-blk export, libblkio 1.0.0, QEMU ba58ccbef6 (approx 7.1.0-rc3) + blkio driver patches
Ansible playbook & results: https://github.com/stefanha/qemu-perf/commit/33196329916b66f1cdbf54b2cbc3898a9d83fb7e

Red Hat

Real-world performance: QEMU with libblkio

How does libblkio's io_uring driver compare against QEMU's io_uring implementation?

QEMU

- `-blockdev host_device,filename=/dev/nvme0n1,aio=io_uring,...`

vs

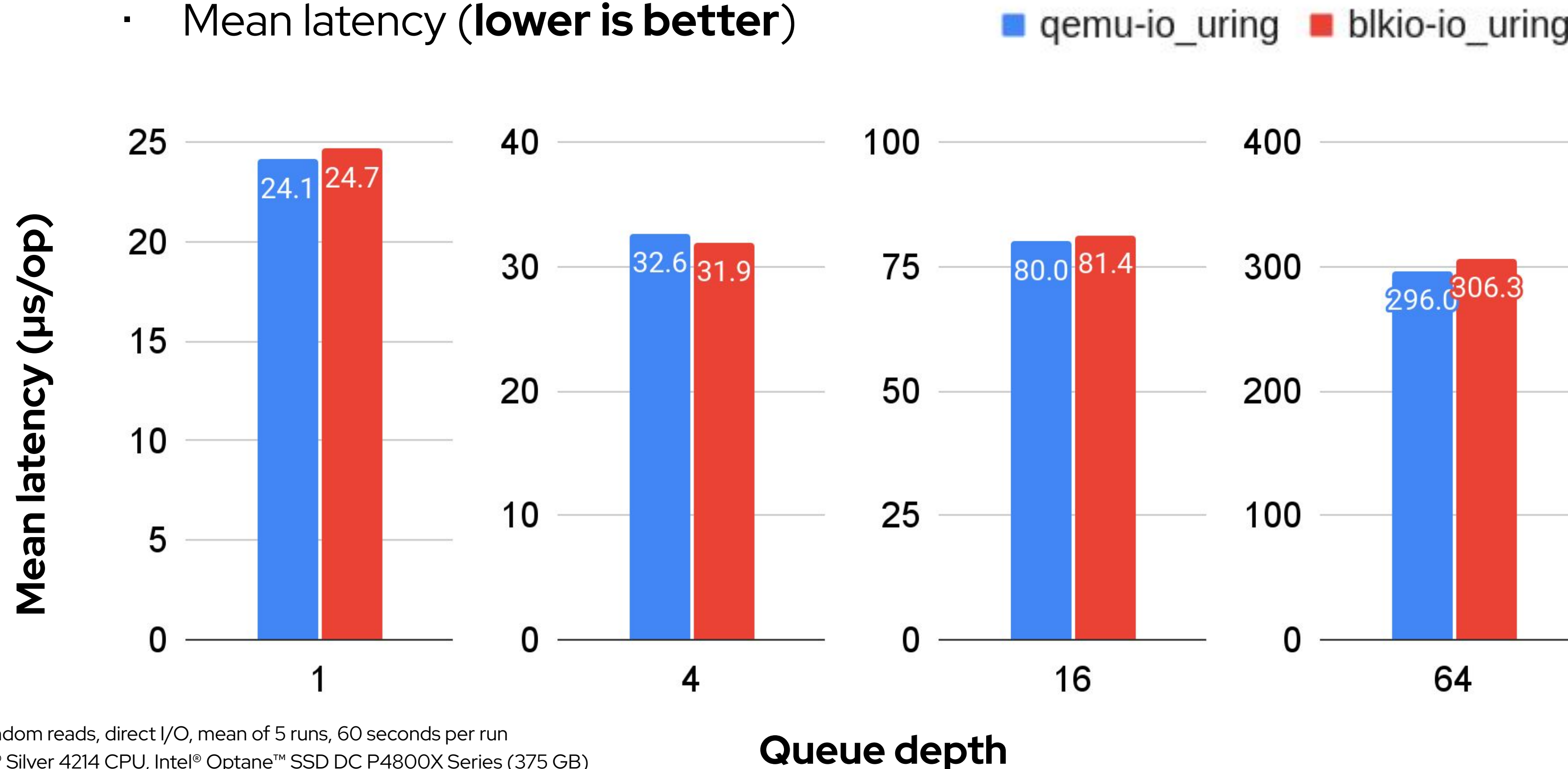
libblkio

- `-blockdev io_uring,filename=/dev/nvme0n1,...`

Red Hat

QEMU io_uring randread

- Native io_uring ***vs*** libblkio io_uring QEMU block drivers
  - 1 vCPU, 512-byte random reads
  - Mean latency (**lower is better**)

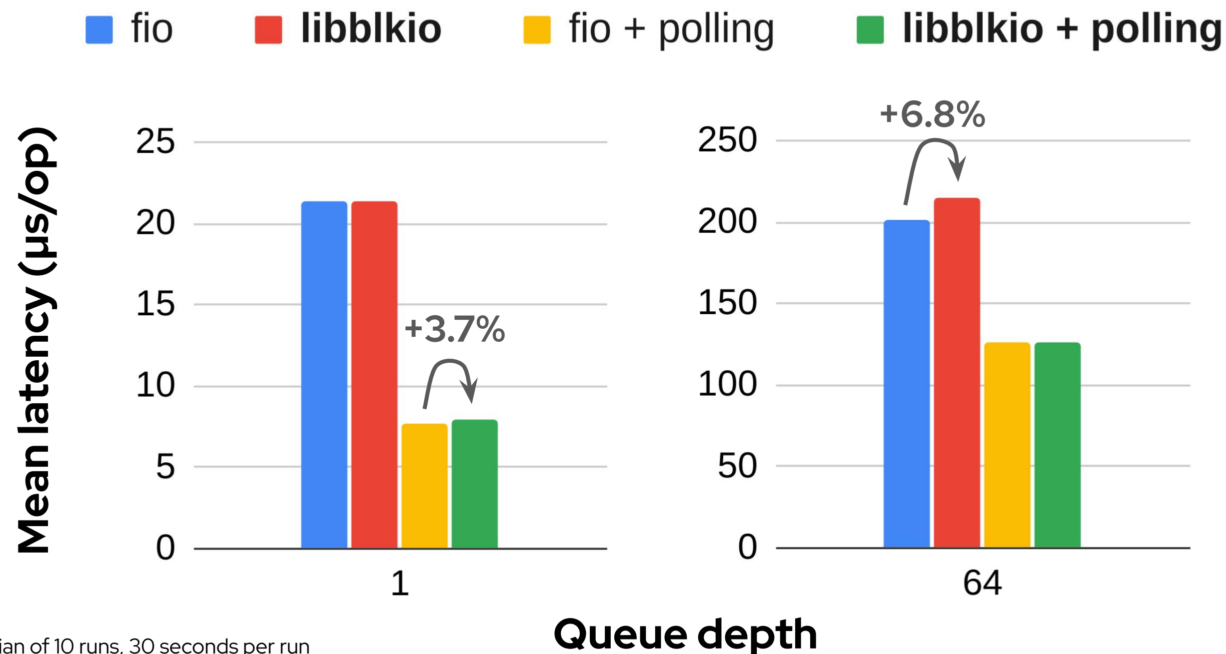


Workload: 512-byte random reads, direct I/O, mean of 5 runs, 60 seconds per run
Hardware: Intel® Xeon® Silver 4214 CPU, Intel® Optane™ SSD DC P4800X Series (375 GB)
Software: Fedora 36 w/ Linux 5.19.1, libblkio 1.0.0, QEMU ba58ccbef6 (approx 7.1.0-rc3) + blkio driver patches
Ansible playbook & results: https://github.com/stefanha/qemu-perf/commit/630d3b4907bb50bc6a3421b51d5926cbdcab5a82

Red Hat

fio micro-benchmarks

- fio's io_uring engine ***vs*** libblkio fio engine using "io_uring" driver
  - 1 core, 512-byte random reads, w/o and w/ driver-level polling (*i.e.*, poll queues)
  - Mean latency (**lower is better**)



Workload: 512-byte random reads, direct I/O, median of 10 runs, 30 seconds per run
Hardware: Intel® Xeon® Silver 4214 CPU, Intel® Optane™ SSD DC P4800X Series (375 GB)
Software: Fedora 36 w/ Linux 5.19.1, libblkio 1.0.0, fio 3.31 + libblkio engine patches

Red Hat

Future work

Red Hat

Future work: Stable blkio Rust crate

Native Rust API should be idiomatic and safe.

Current API exposes MemoryRegion and raw iovec pointers

- Better abstractions needed for safety

How to manage request lifetime across do_io() loop?

- Caller is trusted to keep the iovecs alive for the duration of the request

Hanna Reitz has been experimenting with the blkio crate in Rust applications.

Future work: Network storage

NVMe over TCP, NBD, iSCSI, etc could be added.

Data path APIs are asynchronous but control path APIs are synchronous.

May need async control path API to avoid hangs.

Red Hat

Future work: Queue passthrough

Emulators like QEMU present a storage device like virtio-blk to the guest.

They emulate the storage controller and invoke corresponding libblkio APIs.

When the underlying device is the same type as the guest device, passing through the queues bypasses the emulator and libblkio for better performance.

Currently in development for virtio-blk devices by Stefano Garzarella.

Red Hat

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

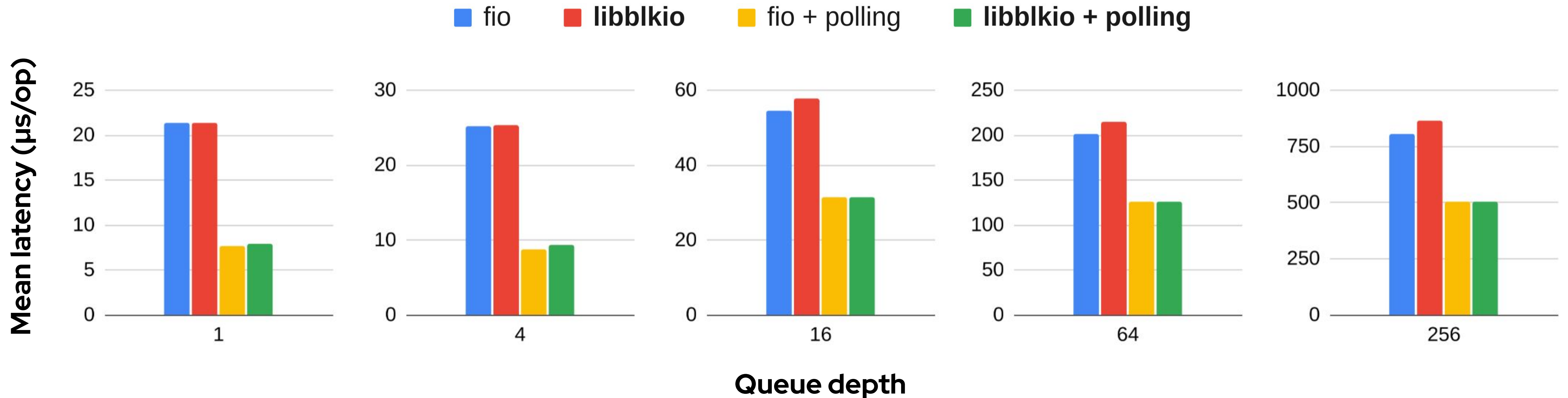
youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Red Hat

Extended fio micro-benchmark results



Workload: 512-byte random reads, direct I/O, median of 10 runs, 30 seconds per run
Hardware: Intel® Xeon® Silver 4214 CPU, Intel® Optane™ SSD DC P4800X Series (375 GB)
Software: Fedora 36 w/ Linux 5.19.1, libblkio 1.0.0, fio 3.31 + libblkio engine patches