Storage Performance Tuning for FAST! Virtual Machines

Fam Zheng
Senior Software Engineer

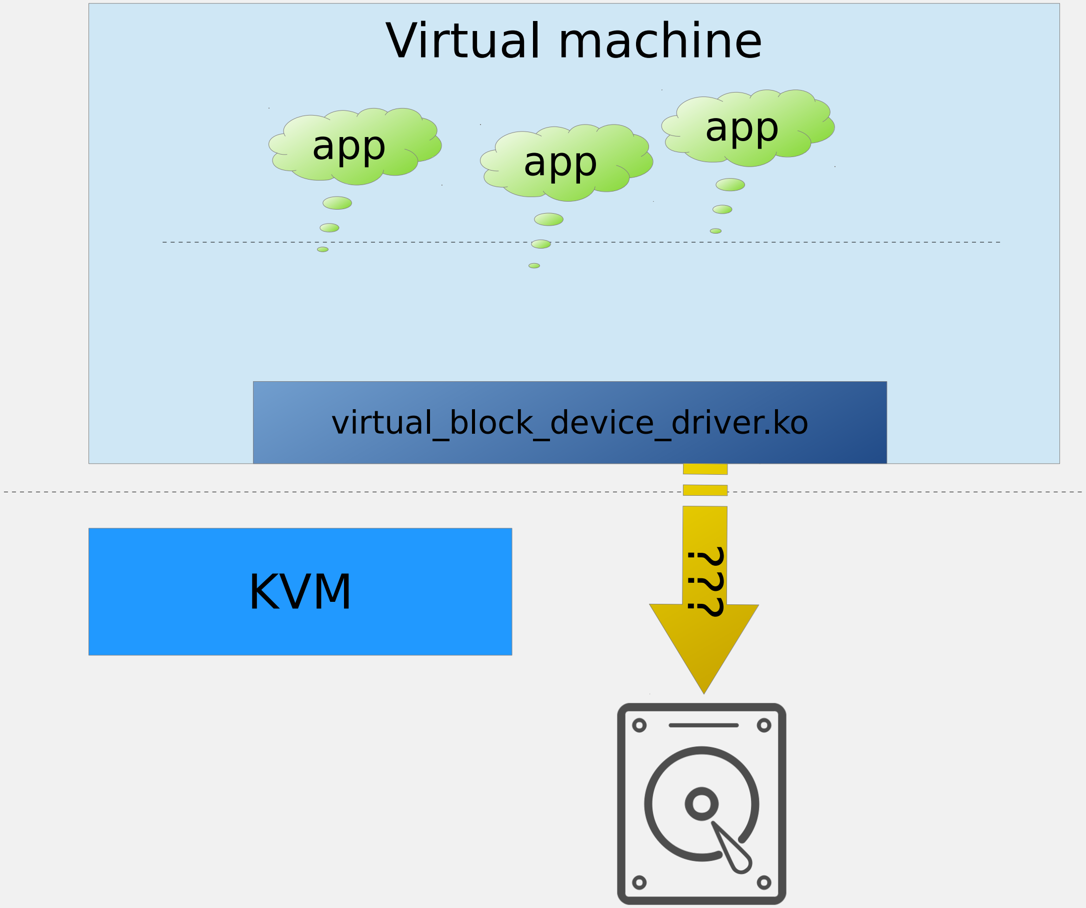
LC3-2018

Outline

- Virtual storage provisioning
- NUMA pinning
- VM configuration options
- Summary
- Appendix

Virtual storage provisioning

Provisioning virtual disks

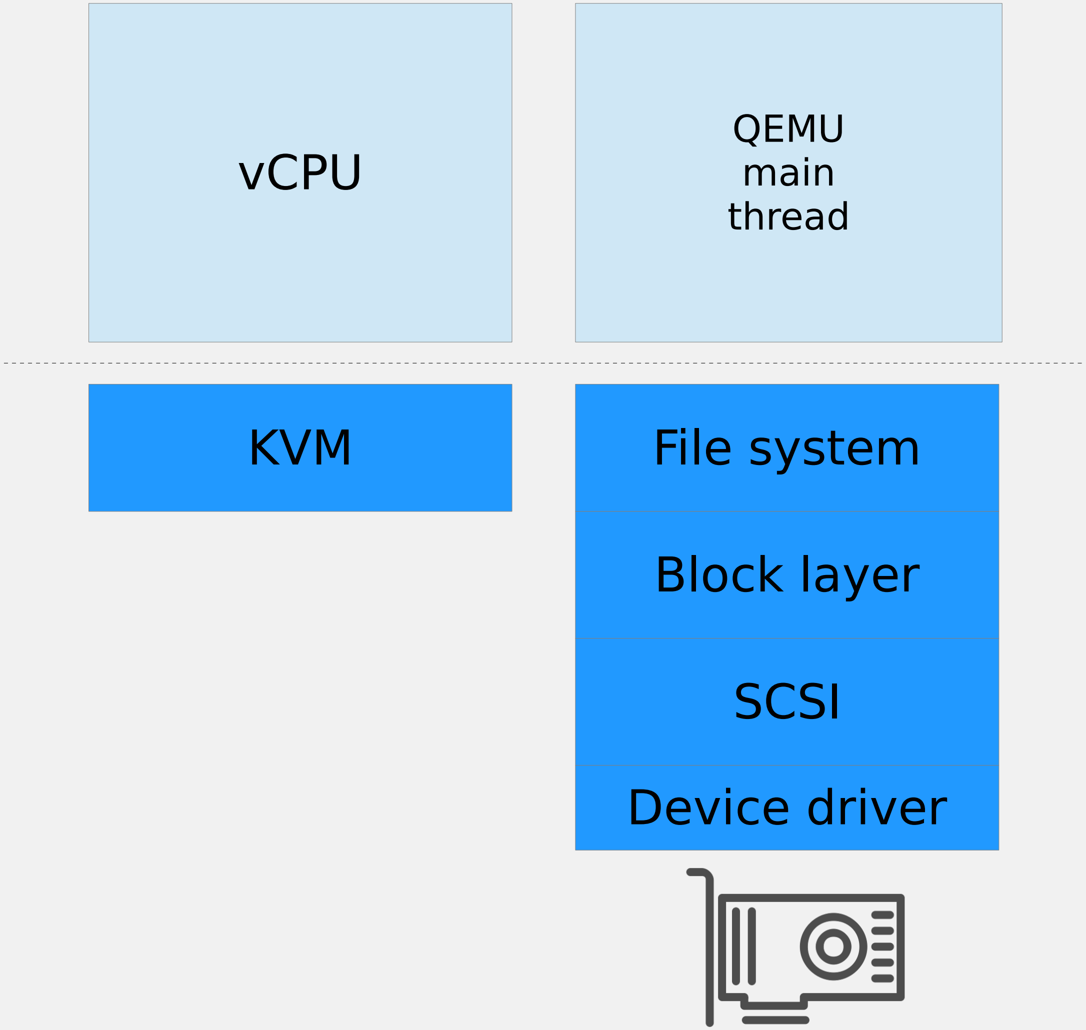


- Virtual storage provisioning is to expose host persistent storage to guest for applications’ use.
- A device of a certain type is presented on a system bus
- Guest uses a corresponding driver to do I/O
- The disk space is allocated from the storage available on the host.

QEMU emulated devices

- Device types: *virtio-blk, virtio-scsi, IDE, NVMe, ...*
- QEMU block features
  - *qcow2, live snapshot*
  - *throttling*
  - *block migration*
  - *incremental backup*
  - ...
- Easy and flexible backend configuration
  - Wide range of protocols: *local file, NBD, iSCSI, NFS, Gluster, Ceph, ...*
  - Image formats: qcow2, raw, LUKS, ...
- Pushed hard for performance
  - *IOThread polling; userspace driver; multiqueue block layer (WIP)*

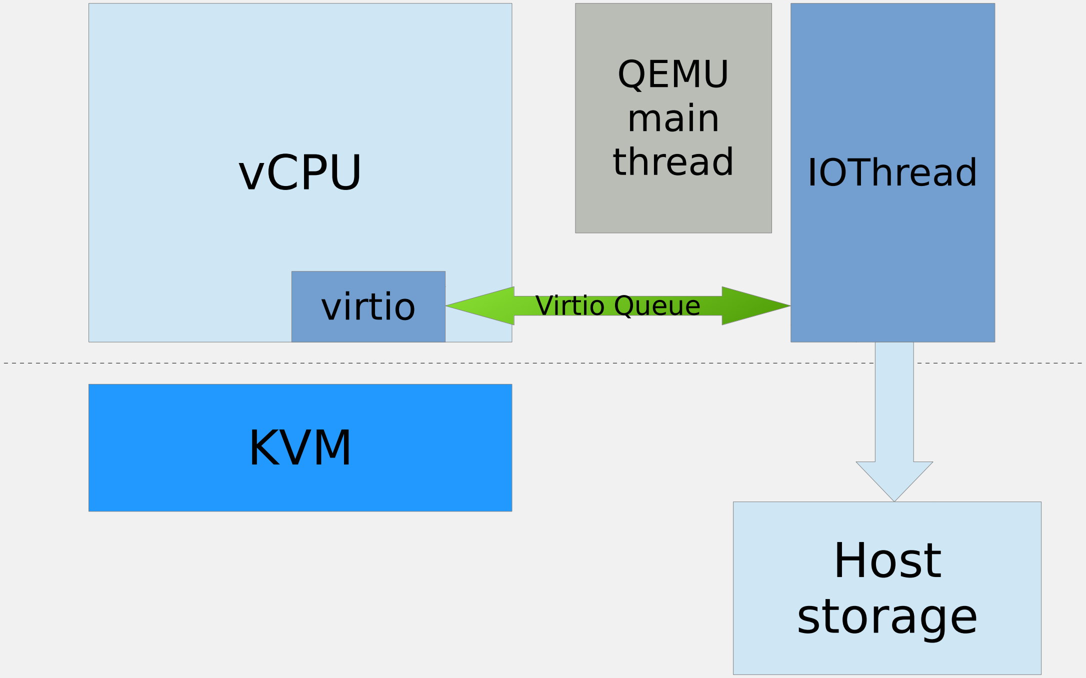
QEMU emulated device I/O (file backed)



I/O Request Lifecycle

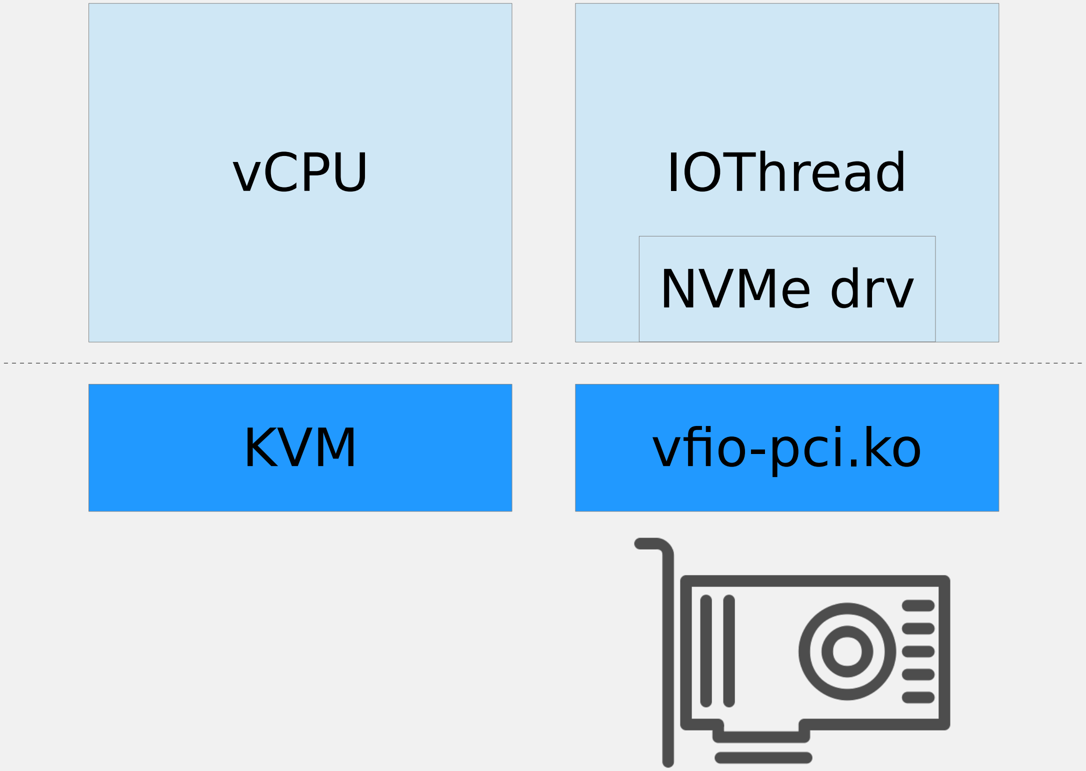
Guest virtio driver
↓
KVM ioeventfd
↓
vdev vring handler
↓
QEMU block layer
↓
LinuxAIO/POSIX syscall
↓
Host VFS/block/SCSI layer
↓
Host device driver
↓
Hardware

QEMU virtio IOThread



- A dedicated thread to handle virtio vrings
- Now fully support QEMU block layer features
  - (Previously known as *x-data-plane* of virtio-blk, limited to raw format, no block jobs)
- Currently one IOThread per device
  - Multi-queue support is being worked on
- Adaptive polling enabled
  - Optimizes away the event notifiers from critical path (Linux-aio, vring, ...)
  - Reduces up to 20% latency

QEMU userspace NVMe driver



With the help of VFIO, QEMU accesses host controller’s submission and completion queues without doing any syscall.

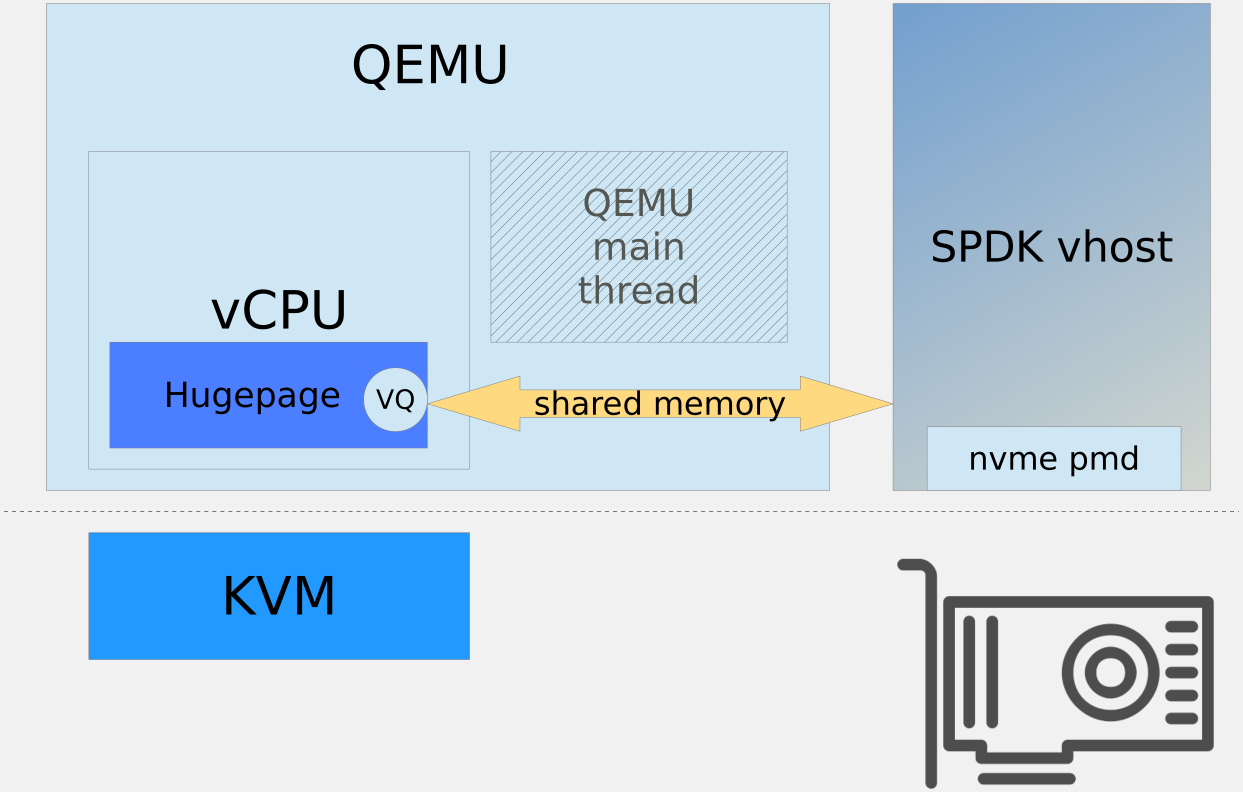
MSI/IRQ is delivered to IOThread with eventfd, if adaptive polling of completion queues doesn’t get result.

No host file system, block layer or SCSI. Data path is shortened.

QEMU process uses the controller exclusively.

(New in QEMU 2.12)

SPDK vhost-user

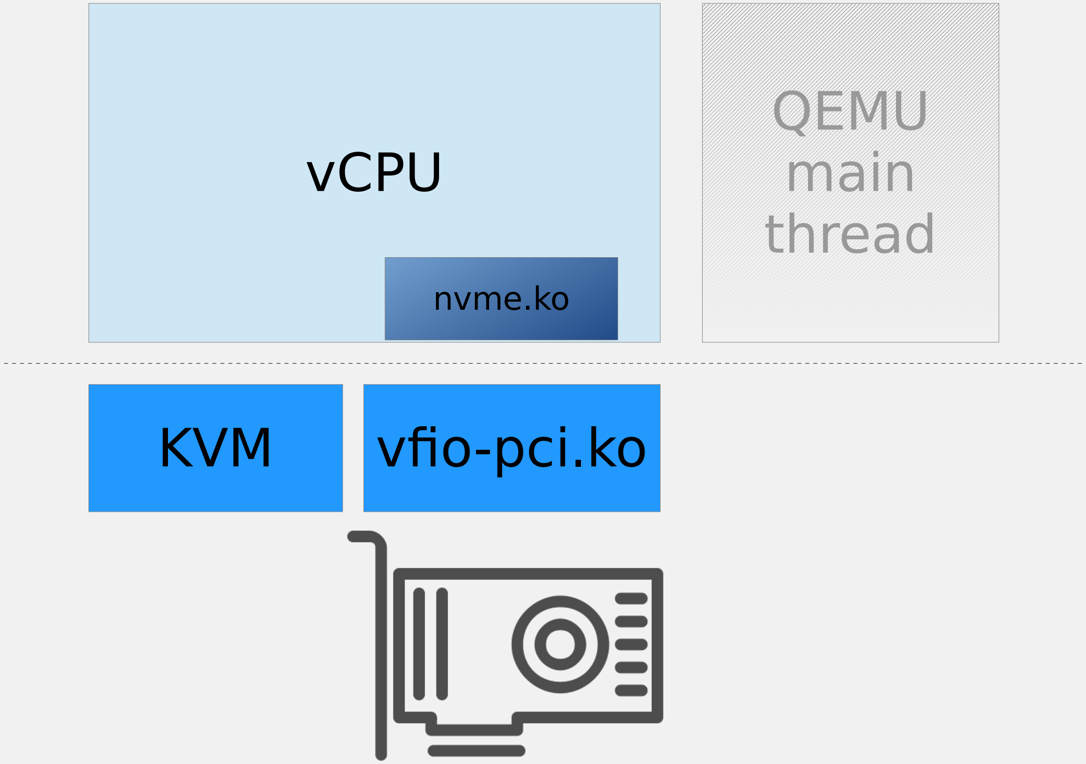


Virtio queues are handled by a separate process, SPDK vhost, which is built on top of DPDK and has a userspace poll mode NVMe driver.

QEMU IOThread and host kernel is out of data path. Latency is greatly reduced by busy polling.

No QEMU block features. No migration (w/ NVMe pmd).

vfio-pci device assignment



vCPU

nvme.ko

QEMU main thread

KVM

vfio-pci.ko

Highly efficient. Guest driver accesses device queues directly without VMEXIT.

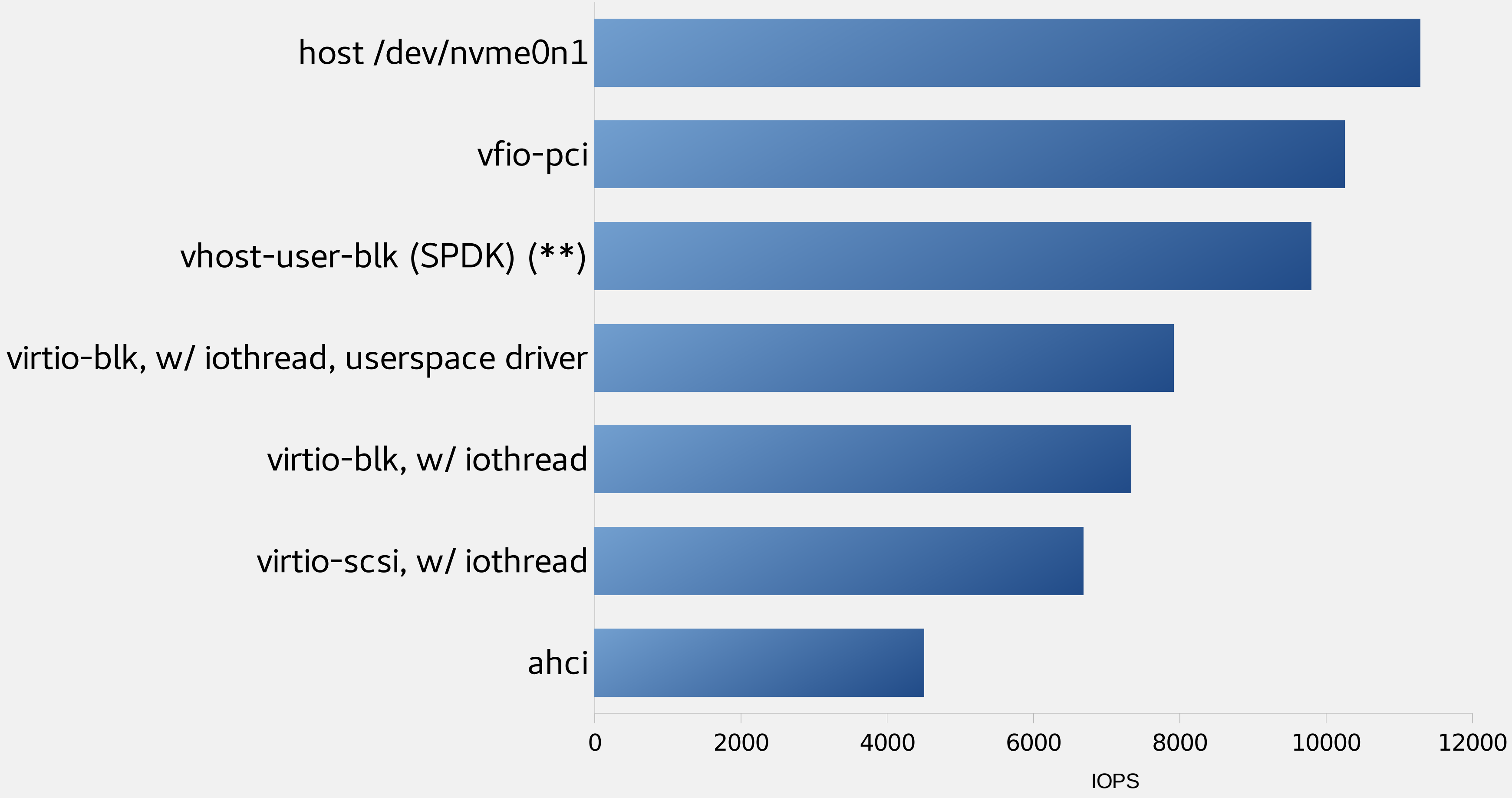
No block features of host system or QEMU. Cannot do migration.

Provisioning virtual disks

Type	Configuration	QEMU block features	Migration	Special requirements	Supported in current RHEL/RHV
QEMU emulated	IDE	✓	✓		✓
	NVMe	✓	✓		✗
	virtio-blk, virtio-scsi	✓	✓		✓
vhost	vhost-scsi	✗	✗		✗
	SPDK vhost-user	✗	✓	Hugepages	✗
Device assignment	vfio-pci	✗	✗	Exclusive device assignment	✓

Sometimes higher performance means less flexibility

fio randread bs=4k iodepth=1 numjobs=1

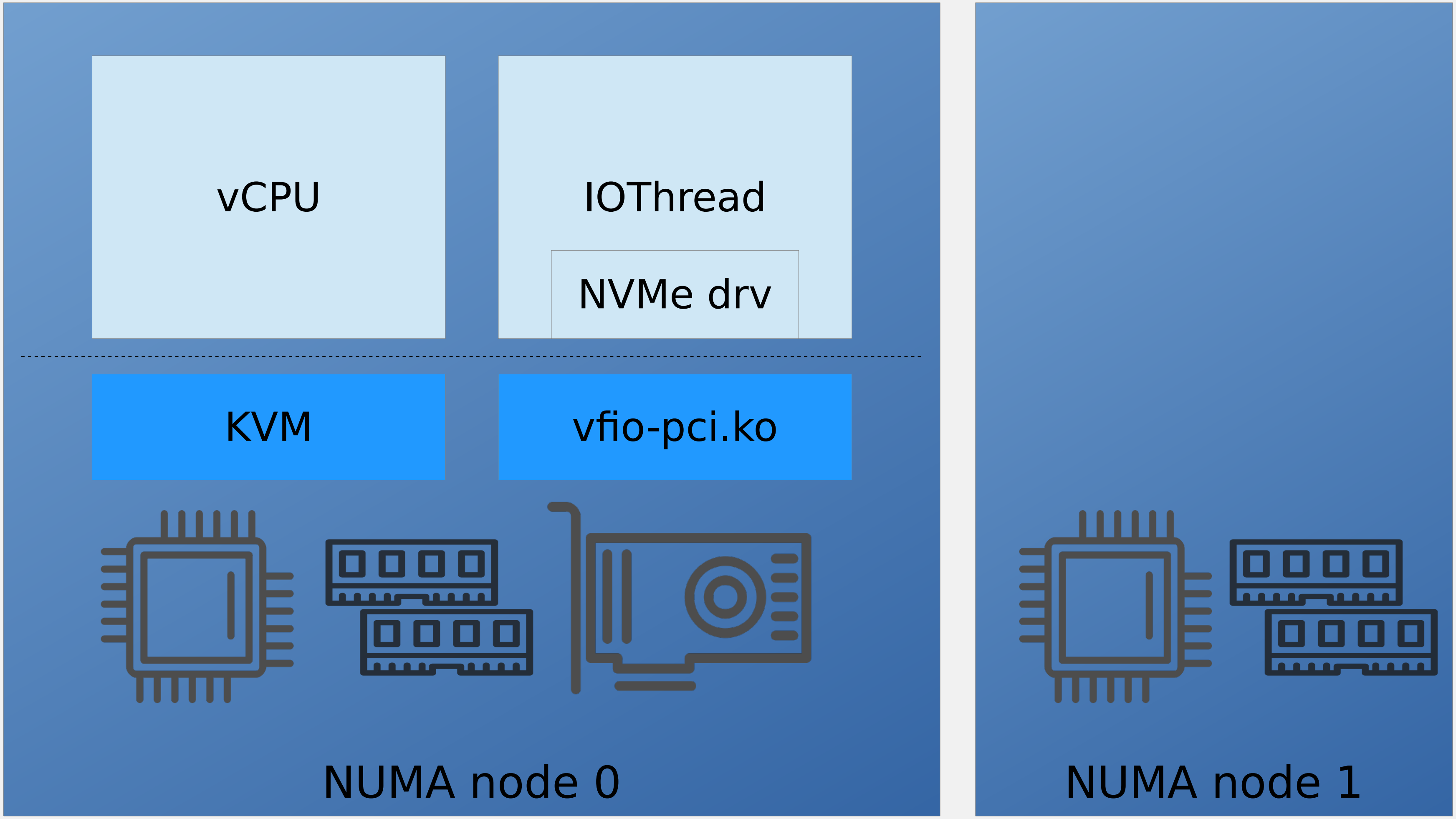


Backend: NVMe, Intel® SSD DC P3700 Series 400G
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, Fedora 28
Guest: Q35, 1 vCPU, Fedora 28
QEMU: 8e36d27c5a
(**): SPDK poll mode driver threads take 100% host CPU cores, dedicatedly

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

NUMA Pinning

NUMA (Non-uniform memory access)



Goal: put vCPU, IOThread and virtual memory on the same NUMA node with the host device that undertakes I/O

Automatic NUMA balancing

- Kernel feature to achieve good NUMA locality
  - Periodic NUMA unmapping of process memory
  - NUMA hinting fault
  - Migrate on fault - moves memory to where the program using it runs
  - Task NUMA placement - moves running programs closer to their memory
- Enabled by default in RHEL:
  cat /proc/sys/kernel/numa_balancing
  1
- Decent performance in most cases
- Disable it if using manual pinning

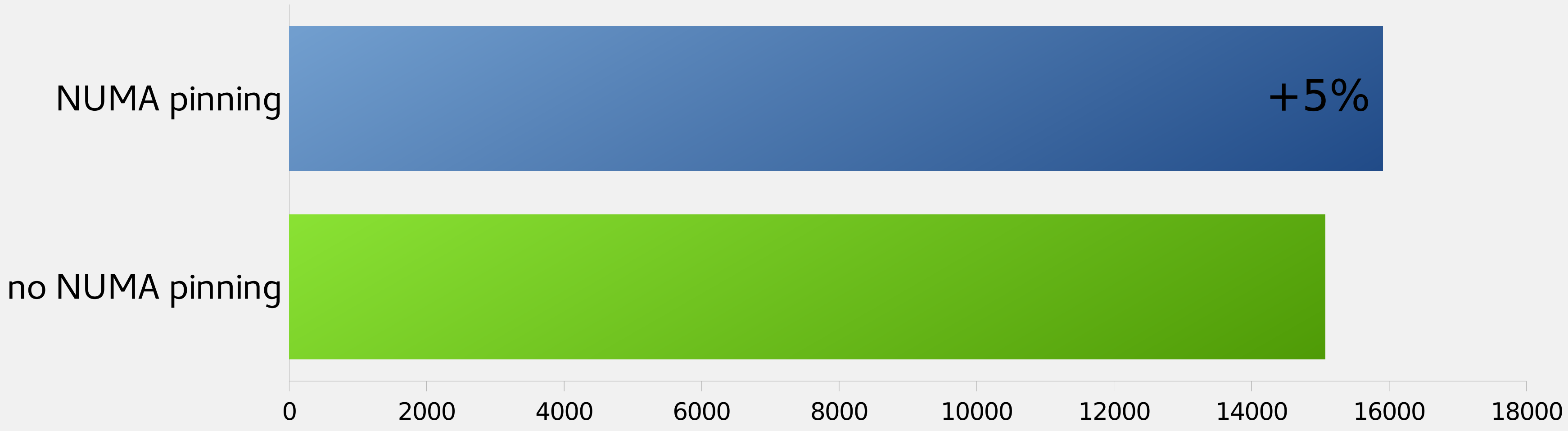
Manual NUMA pinning

- Option 1: Allocate all vCPUs and virtual memory on the optimal NUMA node
  $ numactl -N 1 -m 1 qemu-system-x86_64 …
- Or use Libvirt (*)
- Restrictive on resource allocation:
  - Cannot use all host cores
  - NUMA-local memory is limited

- Option 2: Create a guest NUMA topology matching the host, pin IOThread to host storage controller’s NUMA node
- Libvirt is your friend! (*)
- Relies on the guest to do the right NUMA tuning

* See appendix for Libvirt XML examples

fio randread bs=4k iodepth=1 numjobs=1



Backend: Intel® SSD DC P3700 Series
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA balancing disabled. Virtual device: virtio-blk w/ IOThread
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

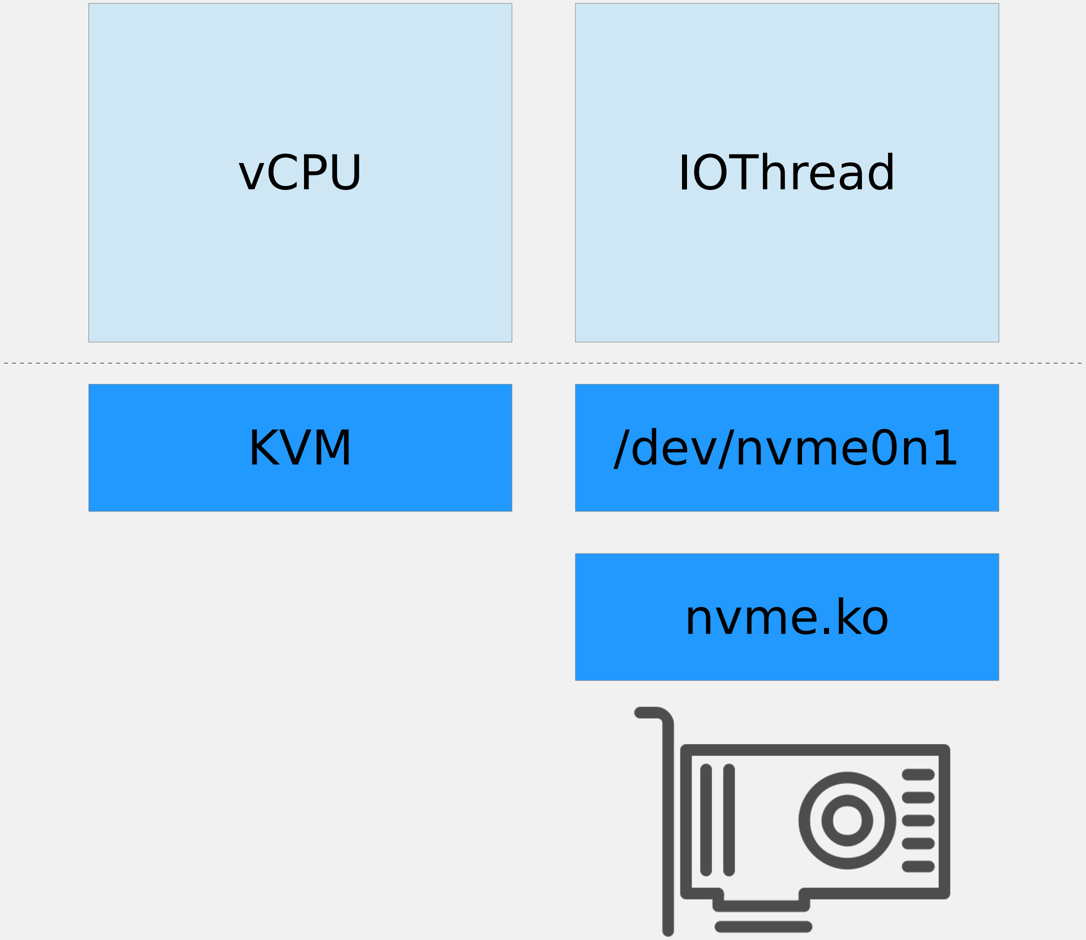
VM Configuration Options

Raw block device vs image file

- Image file is more flexible, but slower
- Raw block device has better performance, but harder to manage
- Note: snapshot is supported with raw block device. E.g:

```
$ qemu-img create -f qcow2 -b /path/to/base/image.qcow2 \
    /dev/sdc
```

QEMU emulated device I/O (block device backed)



vCPU

IOThread

KVM

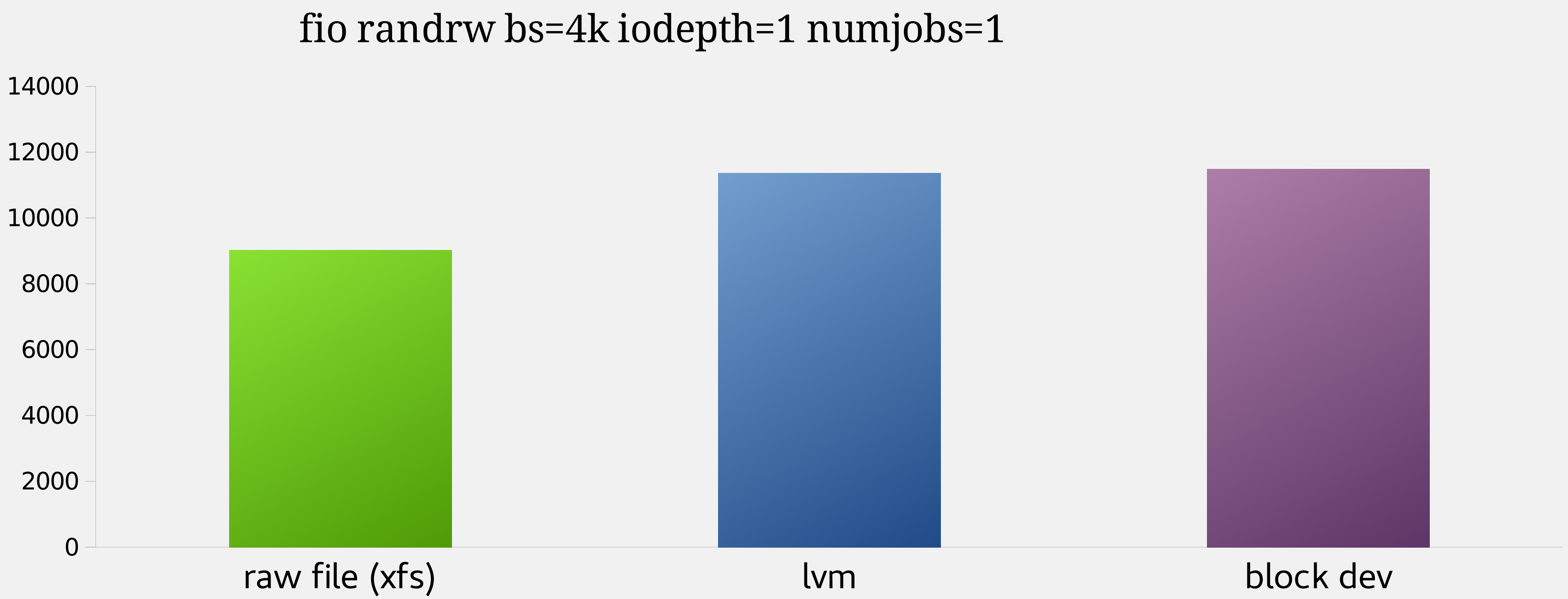
/dev/nvme0n1

nvme.ko

Using raw block device may improve performance: no file system in host.

Middle ground: use LVM

LVM is much more flexible and easier to manage than raw block or partitions, and has good performance



Backend: Intel® SSD DC P3700 Series
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA pinning. Virtual device: virtio-blk w/ IOThread
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

Using QEMU VirtIO IOThread

- When using virtio, it’s recommended to enabled IOThread:

```
qemu-system-x86_64 … \
-object iothread,id=iothread0 \
-device virtio-blk-pci,iothread=iothread0,id=… \
-device virtio-scsi-pci,iothread=iothread0,id=…
```

- Or in Libvirt...

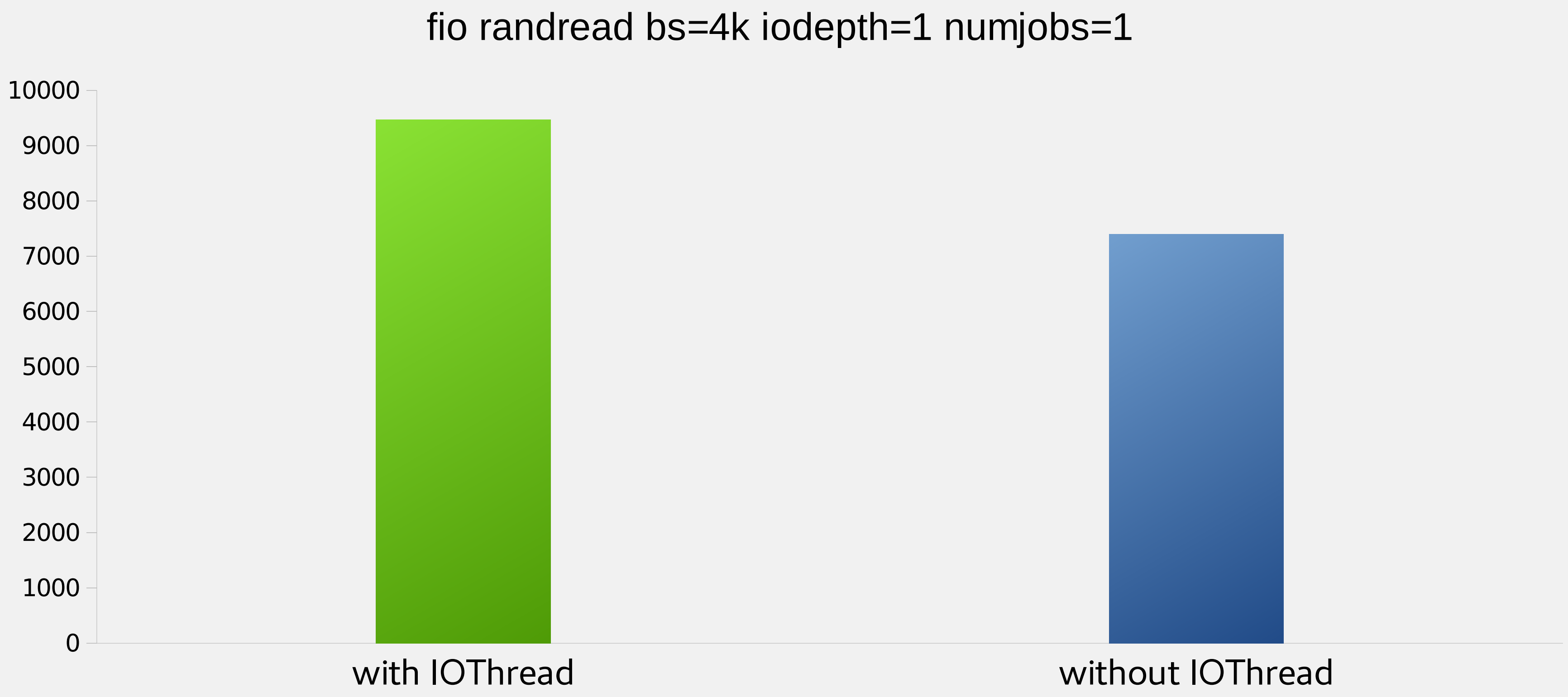
Using QEMU VirtIO IOThread (Libvirt)

```
<domain>
  ...
  <iothreads>1</iothreads>

  <disk type='file' device='disk'>
    <driver name='qemu' type='raw' cache='none' iothread='1'/>
    <target dev='vda' bus='virtio'/>
    ...
  </disk>

  <devices>
    <controller type='scsi' index='0' model='virtio-scsi'>
      <driver iothread='1'/>
      ...
    </controller>
  </devices>
</domain>
```

virtio-blk with and without enabling IOThread

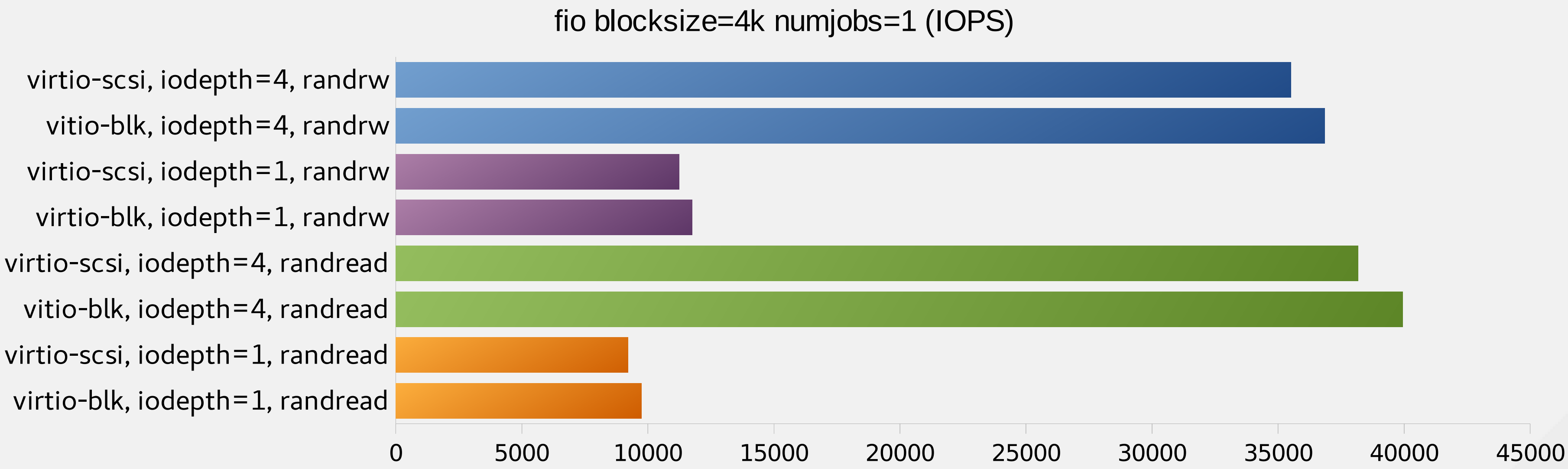


Backend: Intel® SSD DC P3700 Series
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA pinning. Virtual device: virtio-blk
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

virtio-blk vs virtio-scsi

- Use virtio-scsi for many disks, or for full SCSI support (e.g. unmap, write same, SCSI pass-through)
  - virtio-blk DISCARD and WRITE ZEROES are being worked on
- Use virtio-blk for best performance



Backend: Intel® SSD DC P3700 Series; QEMU userspace driver (nvme://)
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA pinning. IOThread enabled.
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

Raw vs qcow2

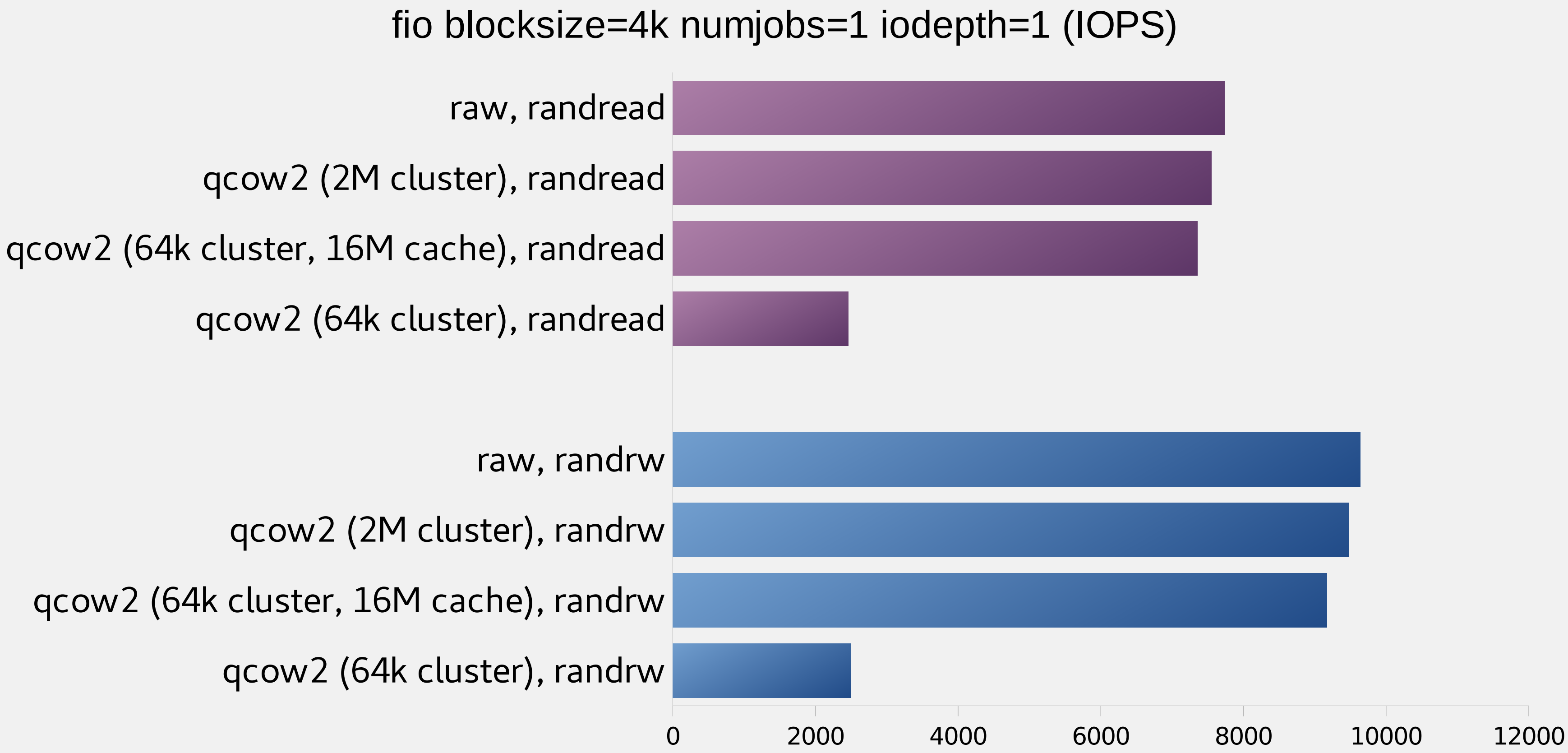
- Don’t like the trade-off between features and performance?
  - Try increasing qcow2 run-time cache size

```
qemu-system-x86_64 … \
-drive \
file=my.qcow2,if=none,id=drive0,aio=native,cache=none,\
cache-size=16M \
...
```

  - Or increase the cluster_size when creating qcow2 images

```
qemu-img create -f qcow2 -o cluster_size=2M my.qcow2 100G
```

Raw vs qcow2

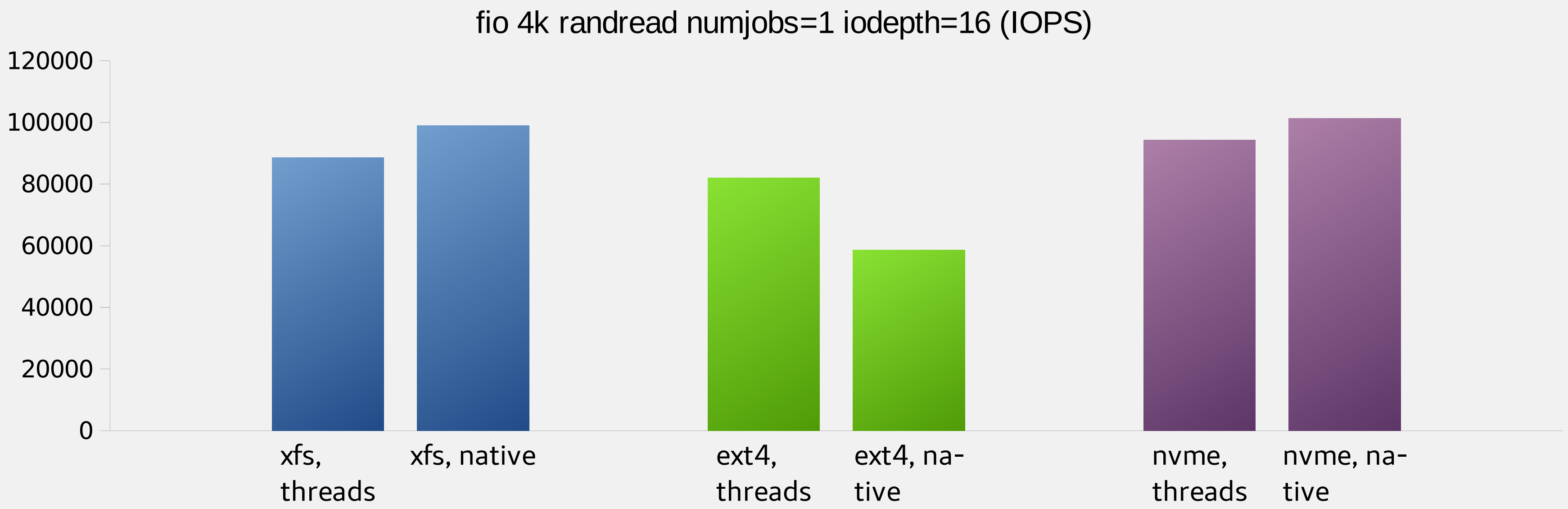


Backend: Intel® SSD DC P3700 Series, formatted as xfs; Virtual disk size: 100G; Preallocation: full
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA pinning. Virtual device: virtio-blk w/ IOThread
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

AIO: native vs threads

- aio=native is usually better than aio=threads
- May depend on file system and workload
  - ext4 native is slower because io_submit is not implemented async



Backend: Intel® SSD DC P3700 Series
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA pinning. Virtual device: virtio-blk w/ IOThread
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

Image preallocation

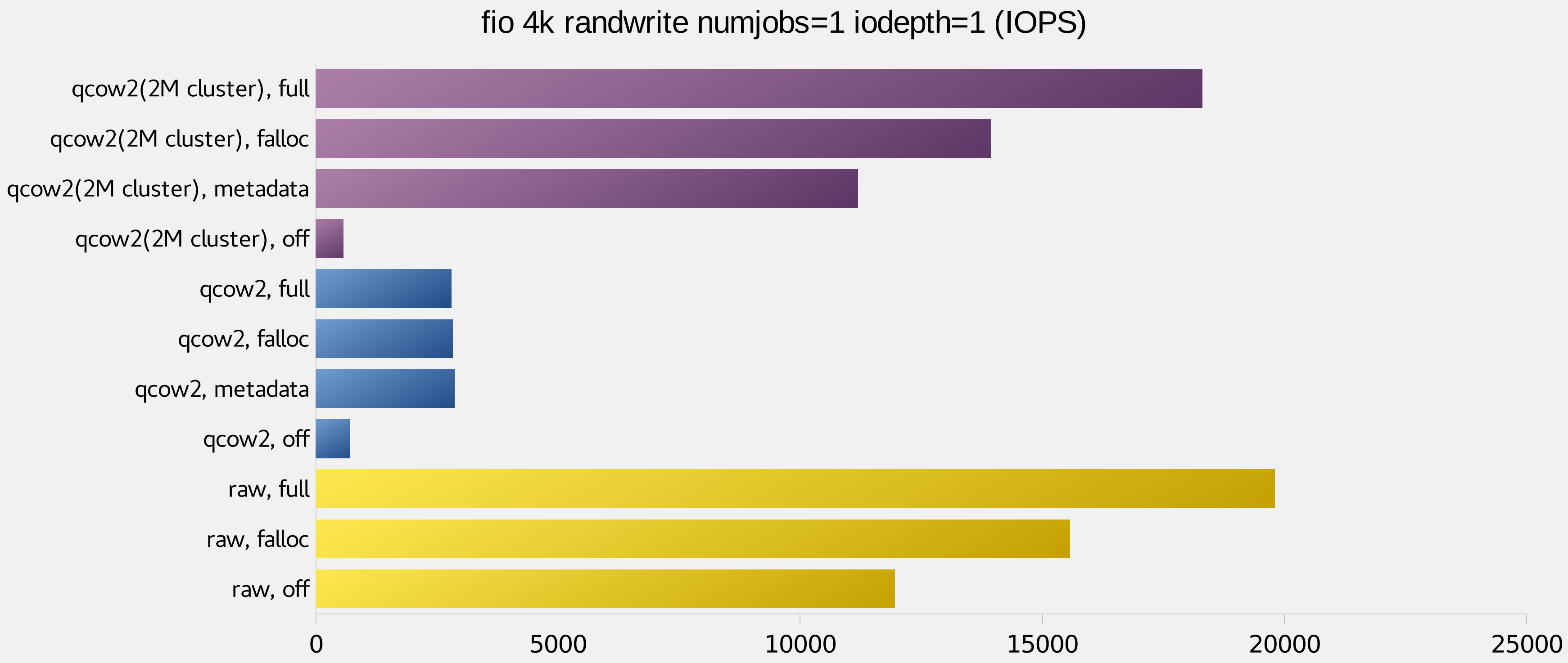
- Reserve space on file system for user data or metadata:

  ```
  $ qemu-img create -f $fmt -o preallocation=$mode test.img 100G
  ```

- Common modes for raw and qcow2:
  - off: no preallocation
  - falloc: use posix_fallocate() to reserve space
  - full: reserve by writing zeros
- qcow2 specific mode:
  - metadata: fully create L1/L2/refcnt tables and pre-calculate cluster offsets, but don’t allocate space for clusters
- Consider enabling preallocation when disk space is not a concern (it may defeat the purpose of thin provisioning)

Image preallocation

- Mainly affect the first pass of write performance after creating VM



Backend: Intel® SSD DC P3700 Series; File system: xfs
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA pinning. Virtual device: virtio-blk w/ IOThread
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

Cache modes

- Cache modes consist of three separate semantics

	Disk write cache	Host page cache bypassing (O_DIRECT)	Ignore flush (dangerous!)
writeback (default)	Y	N	N
none	Y	Y	N
writethrough	N	N	N
directsync	N	Y	N
unsafe	Y	N	Y

- Usually cache=none is the optimal value
  - To avoid redundant page cache in both host kernel and guest kernel with O_DIRECT
- But feel free to experiment with writeback/directsync as well
- unsafe can be useful for throwaway VMs or guest installation

NVMe userspace driver in QEMU

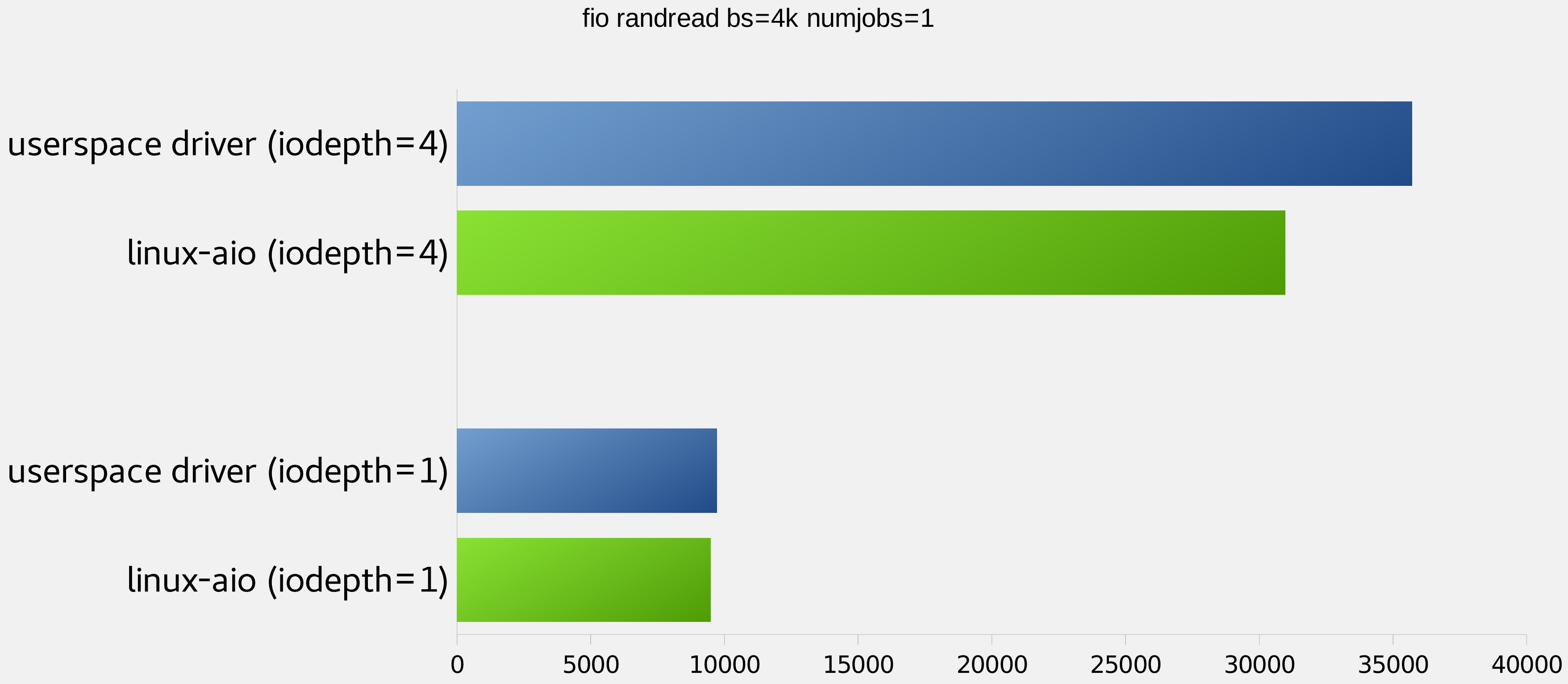
- Usage
  - Bind the device to vfio-pci.ko

    ```
    # modprobe vfio
    # modprobe vfio-pci
    # echo 0000:44:00.0 > /sys/bus/pci/devices/0000:44:00.0/driver/unbind
    # echo 8086 0953 > /sys/bus/pci/drivers/vfio-pci/new_id
    ```

  - Use nvme:// protocol for the disk backend

    ```
    qemu-system-x86_64 … \
    -drive file=nvme://0000:44:00.0/1,if=none,id=drive0 \
    -device virtio-blk,drive=drive0,id=vblk0,iothread=…
    ```

userspace NVMe driver vs linux-aio

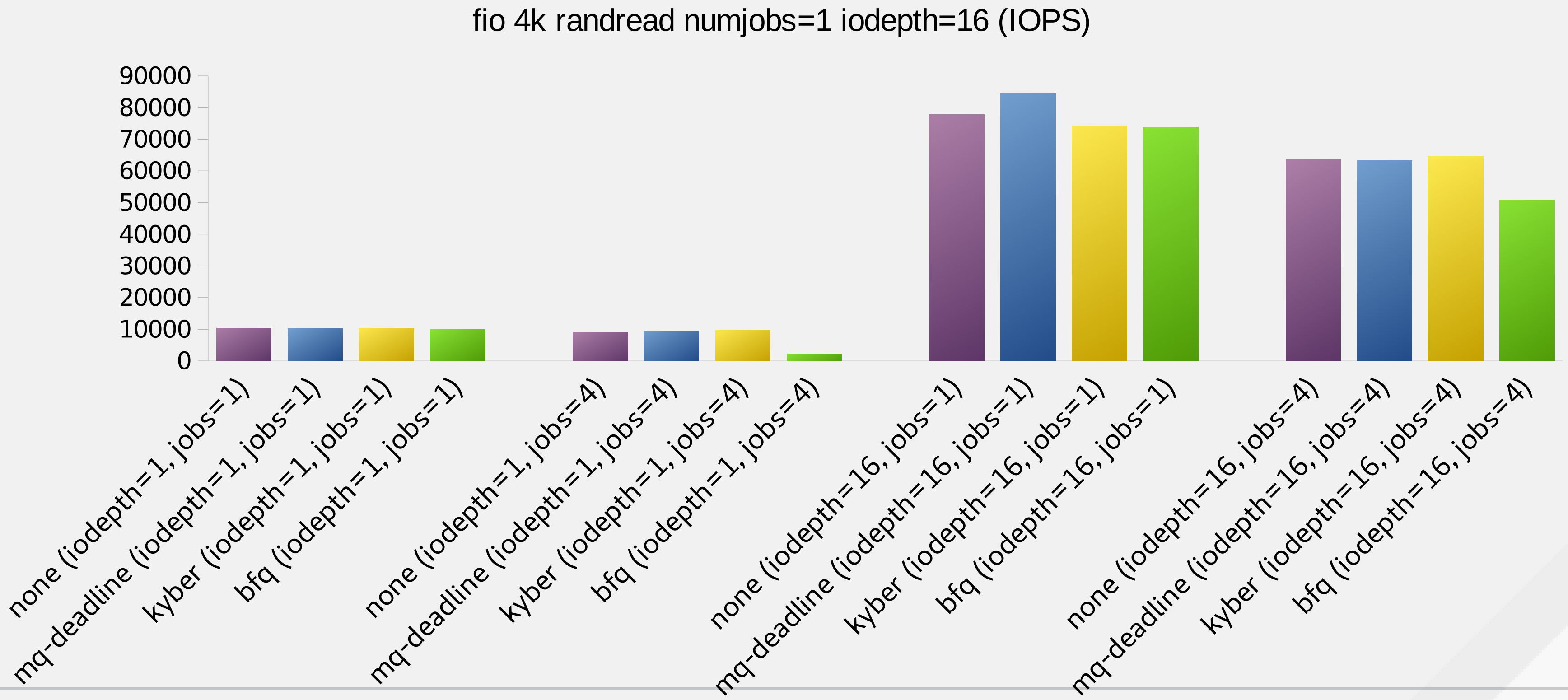


Backend: Intel® SSD DC P3700 Series
Host: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 2 sockets w/ NUMA, Fedora 28
Guest: Q35, 6 vCPU, 1 socket, Fedora 28, NUMA pinning. Virtual device: virtio-blk w/ IOThread
QEMU: 8e36d27c5a

[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

IO scheduler

- blk-mq has been enabled on virtio-blk and virtio-scsi
- Available schedulers: none, mq-deadline, kyber, bfq
- Select one of *none*, *mq-deadline* and *kyber* depending on your workload, if using SSD



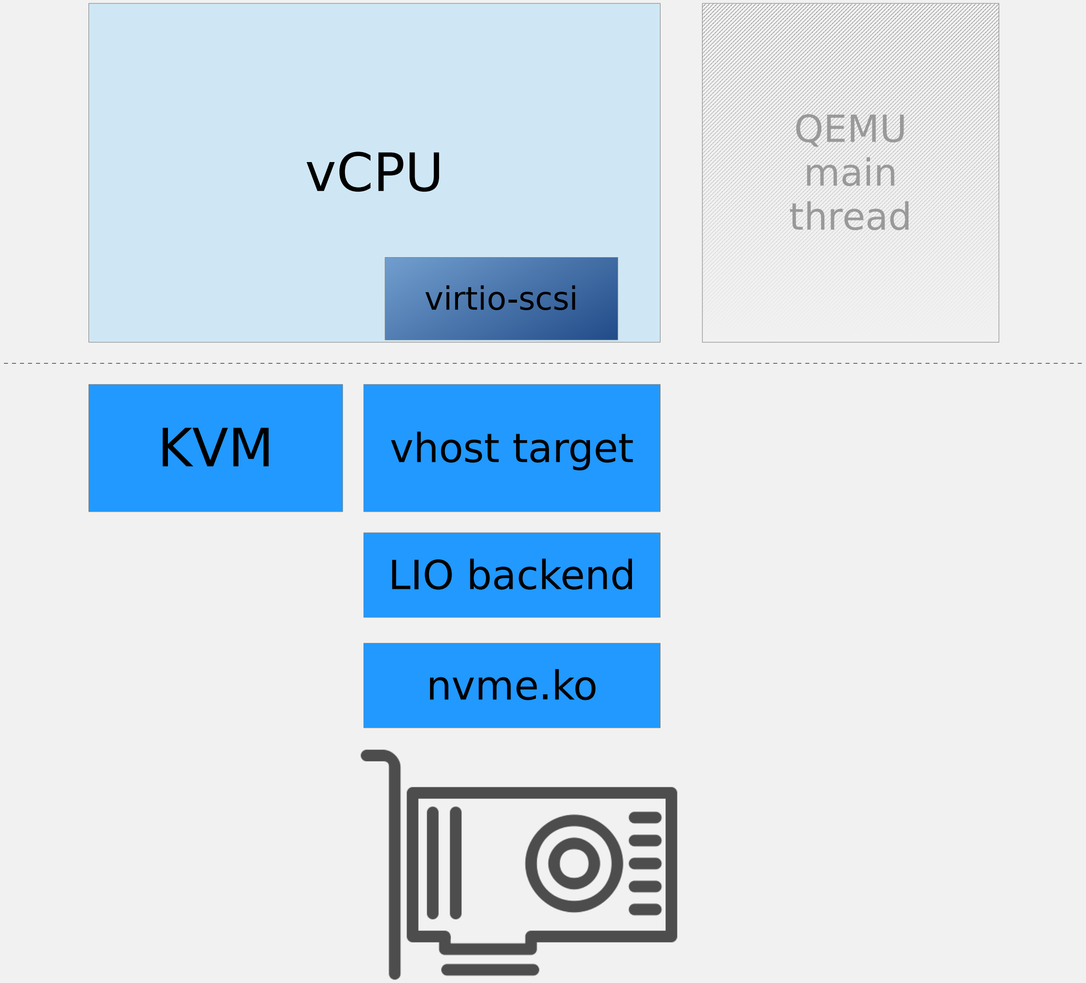
[*]: numbers are collected for relative comparison, not representative as a formal benchmarking result

Summary

- Plan out your virtual machines based on your own constraints: *Live migration, IO throttling, live snapshot, incremental backup, hardware availability*, ...
- Workload characteristics must be accounted for
- Upgrade your QEMU and Kernel, play with the new features!
- Don’t presume about performance, benchmark it!
- NVMe performance heavily depends on preconditioning, take the numbers with a grain of salt
- Have fun tuning for your FAST! virtual machines :-)

THANK YOU

Appendix: vhost-scsi

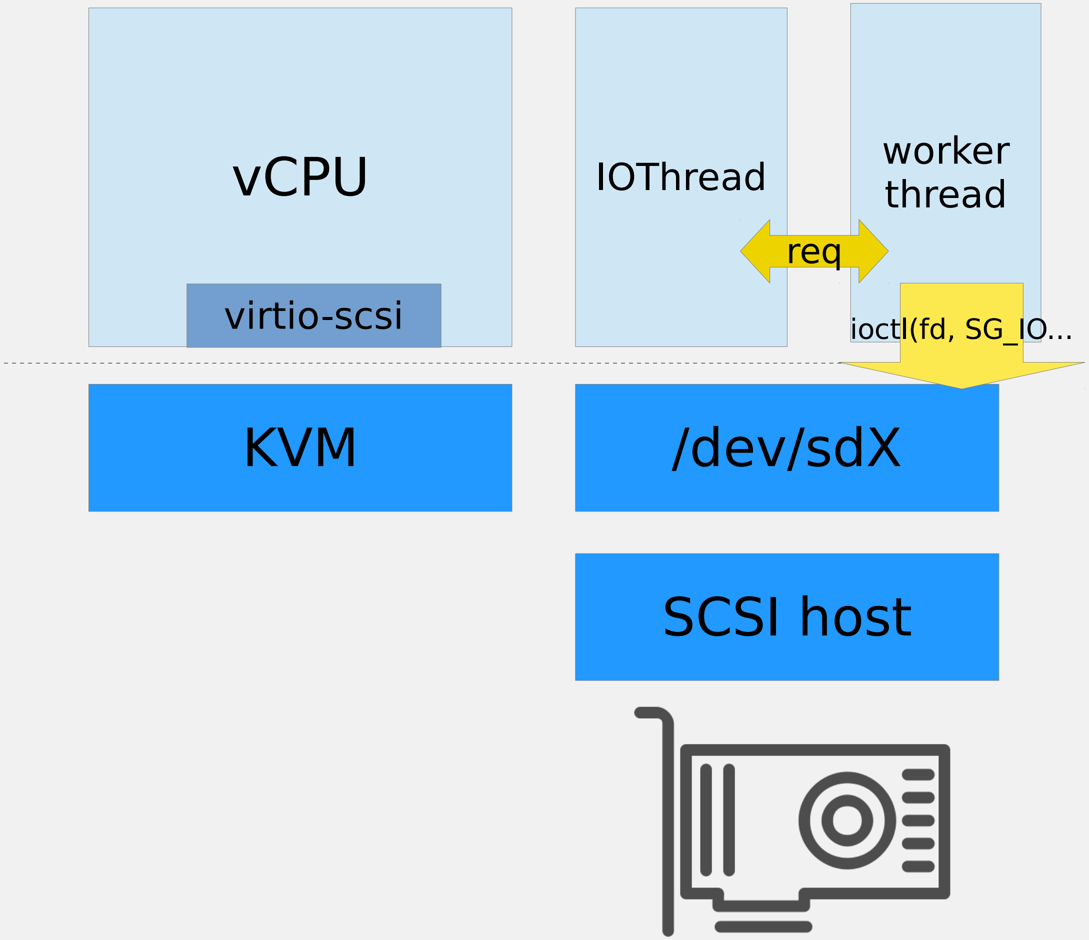


I/O requests on the virtio queue are handled by host kernel vhost LIO target.

Data path is efficient: no ctx switch to userspace is needed (IOThread is out of data path). Backend configuration with LIO is relatively flexible.

Not widely used. No migration support. No QEMU block layer features.

Appendix: QEMU SCSI pass-through



SCSI commands are passed from guest SCSI subsystem (or userspace SG_IO) to device.

Convenient to expose host device's SCSI functions to guest.

No asynchronous SG_IO interface available. aio=native has no effect.

Appendix: NUMA - Libvirt xml syntax (1)

- vCPU pinning:

```
<vcpu cpuset='0-7'>8</vcpu>
<cputune>
  <vcpupin vcpu='0' cpuset='0'/>
  <vcpupin vcpu='1' cpuset='1'/>
  <vcpupin vcpu='2' cpuset='2'/>
  <vcpupin vcpu='3' cpuset='3'/>
  <vcpupin vcpu='4' cpuset='4'/>
  <vcpupin vcpu='5' cpuset='5'/>
  <vcpupin vcpu='6' cpuset='6'/>
  <vcpupin vcpu='7' cpuset='7'/>
</cputune>
```

Appendix: NUMA - Libvirt xml syntax (2)

- Memory allocation policy

```
<numatune>
  <memory mode='strict' nodeset='0'>
</numatune>
<cpu>
  <numa>
    <cell id="0" cpus="0-1" memory="3" unit="GiB"/>
    <cell id="1" cpus="2-3" memory="3" unit="GiB"/>
  </numa>
</cpu>
```

Appendix: NUMA - Libvirt xml syntax (3)

- Pinning the whole emulator

```
<cputune>
  <emulatorpin cpuset="1-3"/>
</cputune>
```

Appendix: NUMA - Libvirt xml syntax (4)

- Creating guest NUMA topology: Use pcie-expander-bus and pcie-root-port to associate device to virtual NUMA node

```
<controller type='pci' index='3' model='pcie-expander-bus'>
  <target busNr='180'>
    <node>1</node>
  </target>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x02'function='0x0'/>
</controller>
<controller type='pci' index='6' model='pcie-root-port'>
  <model name='ioh3420'/>
  <target chassis='6' port='0x0'/>
  <address type='pci' domain='0x0000' bus='0x03' slot='0x00' function='0x0'/>
</controller>
```