IO throttling in QEMU
&
Cache influence

Fu.Lin(river)
lfu@suse.com

11/2017

Pre Intro -- io controller

Cgroup blkio-controller:

- ➢ Proportional weight policy files
- ➢ Throttling/Upper limit policy files

cgroup subsys "blkio" implements the block io controller.
As listed above, one is by io weight, another by limiting iops or bps.

Virsh command for io controller:

- ➔ `blkiotune`: like mechanism of *Proportional weight policy files*
- ➔ `blkdeviotune`: like mechanism of *Throttling/Upper limit policy files*

According my investigation:
blkiotune's function is provided by "blkio" subsystem in host,
blkdeviotune mechanism named throttling is provided by QEMU.

Reference: Cgroup blk subsystem -- *https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt*

Intro

Evaluation Index:

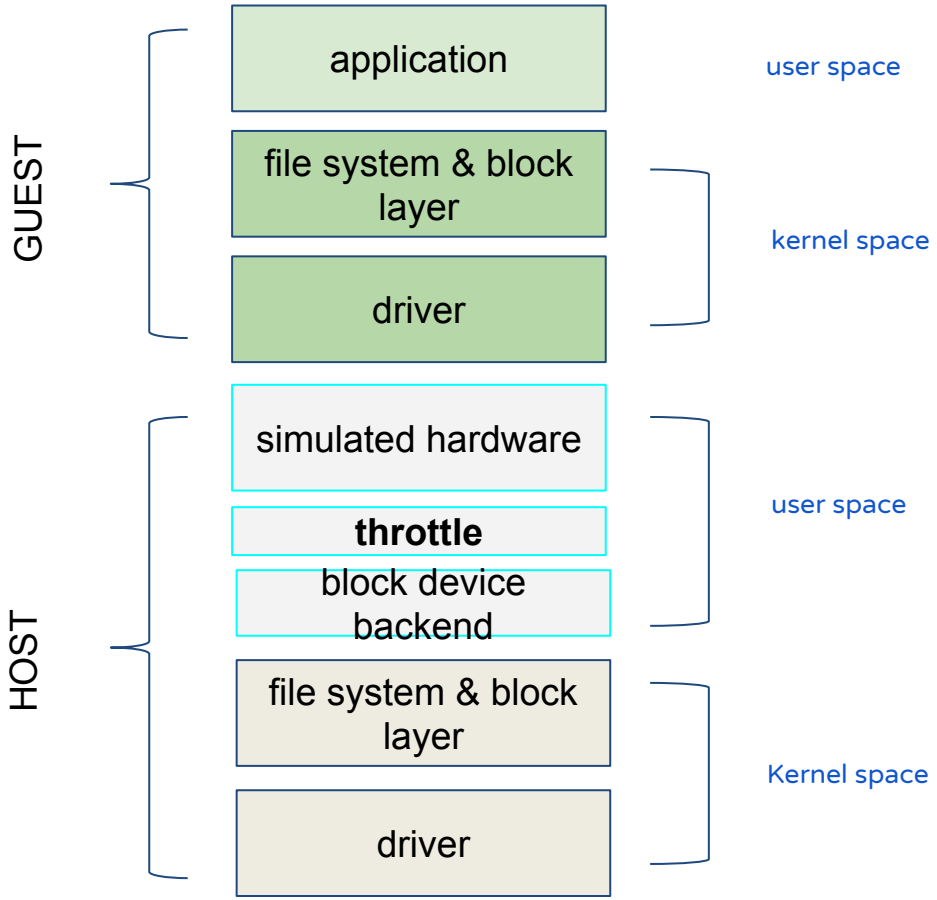
IOPS -- Input/Output Operations Per Second

BPS -- Bandwidth Per Second

Relationship:

BSP = IOPS * <block size>

How does the throttling work?

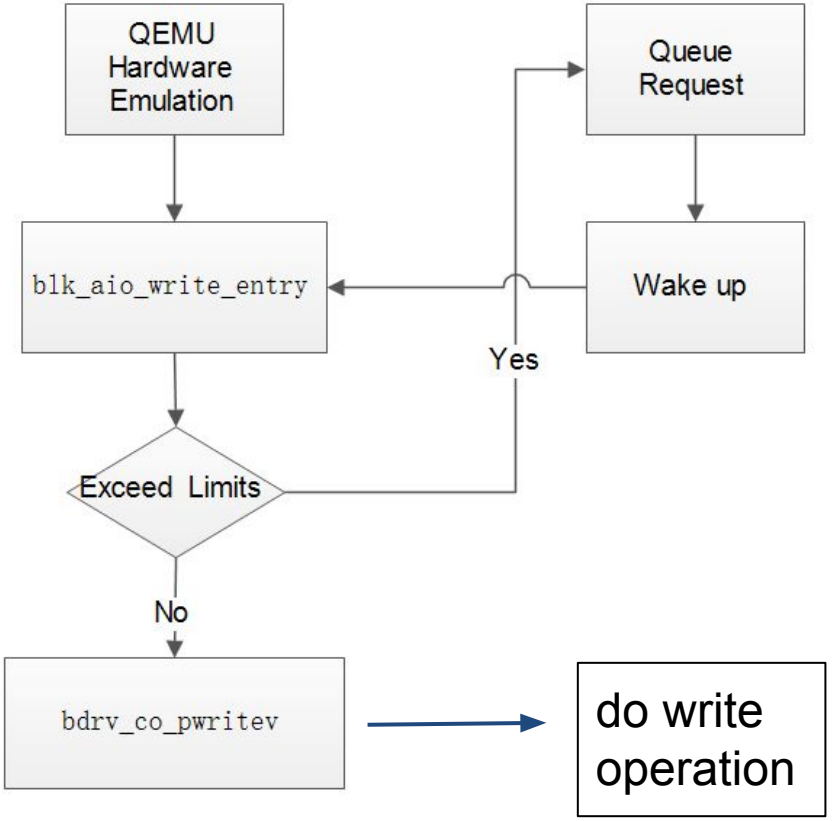


Rference: More Block Device Con guration, KVM Forum 2014

How does the throttling work?(cont.)

When IO comming…

example: write



Reference: [QEMU QoS特性及原理分析和Librbd QoS定义]

How does the throttling work?(cont.)

Initialization

```
// begin from vl.c
main
 -- drive_init_func
     -- drive_new
         -- <parse parameters about throttling>
            blockdev_init
             -- extract_common_blockdev_options(opts, &bdrv_flags, &throttling_group, &cfg,
                           &detect_zeroes, &error); // set throttling parameters' value
                /* disk I/O throttling */
                if (throttle_enabled(&cfg)) {
                    if (!throttling_group) { // if not belong to any group, then belong to itself
                        throttling_group = id;
                    }
                    blk_io_limits_enable(blk, throttling_group);
                     -- throttle_group_register_tgm
                         -- QLIST_INSERT_HEAD(&tg->head, tgm, round_robin); // initialize round-robin list
                         -- throttle_timers_init // initialize time related function and relations
                            qemu_co_queue_init   // initialize request queue
                    blk_set_io_limits(blk, &cfg);
                     -- throttle_group_config
                         -- throttle_config // initialize level / burst_level and previous_leak
                            throttle_group_restart_tgm // create coroutine, then enter
                             -- qemu_coroutine_create
                             -- aio_co_enter
                }
```

Note: *level*, *burst_level*, *previous_leak* are needed by the throttling running

Initialization

- parameters' value
  - total
  - total-max
  - total-max-length
  - iops-size
  - group
- throttle group list (a round-robin fashion)
  - `struct ThrottleTimers`
    - struct members
    - callbacks
      - `read_timer_cb`
      - `read_timer_cb`
- throttling running environment
  - `level`
  - `burst_level`
  - coroutine context

How does the throttling work?(cont.)

HOOK POINT

- reset operation account
  - `bkt->level`
  - `bkt->burst_level`
- increase operation account
  - `bkt->level`
  - `bkt->burst_level`
- wait
  - wait condition
  - wait time
  - handle wait

When io coming...

```
// block/block-backend.c, in <read | write> process
blk_aio_<read | write>_entry
 -- blk_co_<preadv | pwritev>
     -- /* throttling disk I/O */
        if (blk->public.throttle_group_member.throttle_state) {
            throttle_group_co_io_limits_intercept(&blk->public.throttle_group_member,
                    bytes, <false | true>);
         -- throttle_group_schedule_timer
            -- throttle_schedule_timer
                -- throttle_compute_timer // decide waitting or not
                    -- throttle_do_leak
                      -- throttle_leak_bucket // reset level, burst_level
                       throttle_compute_wait_for // waiting time
                        -- throttle_compute_wait // Leaky Bucket Algorithm implementation
            if (must_wait || tgm->pending_reqs[is_write]) {
                ...
                qemu_co_queue_wait // handle wait when need waitting
                 -- QSIMPLEQ_INSERT_TAIL // added in queue
                ...
            }
            throttle_account /* The I/O will be executed, so do the accounting:
                                io units including fragmentation and bandwidth */
            schedule_next_request
        }
        bdrv_co_<preadv | pwritev> // execute
```

How does the throttling work?(cont.)

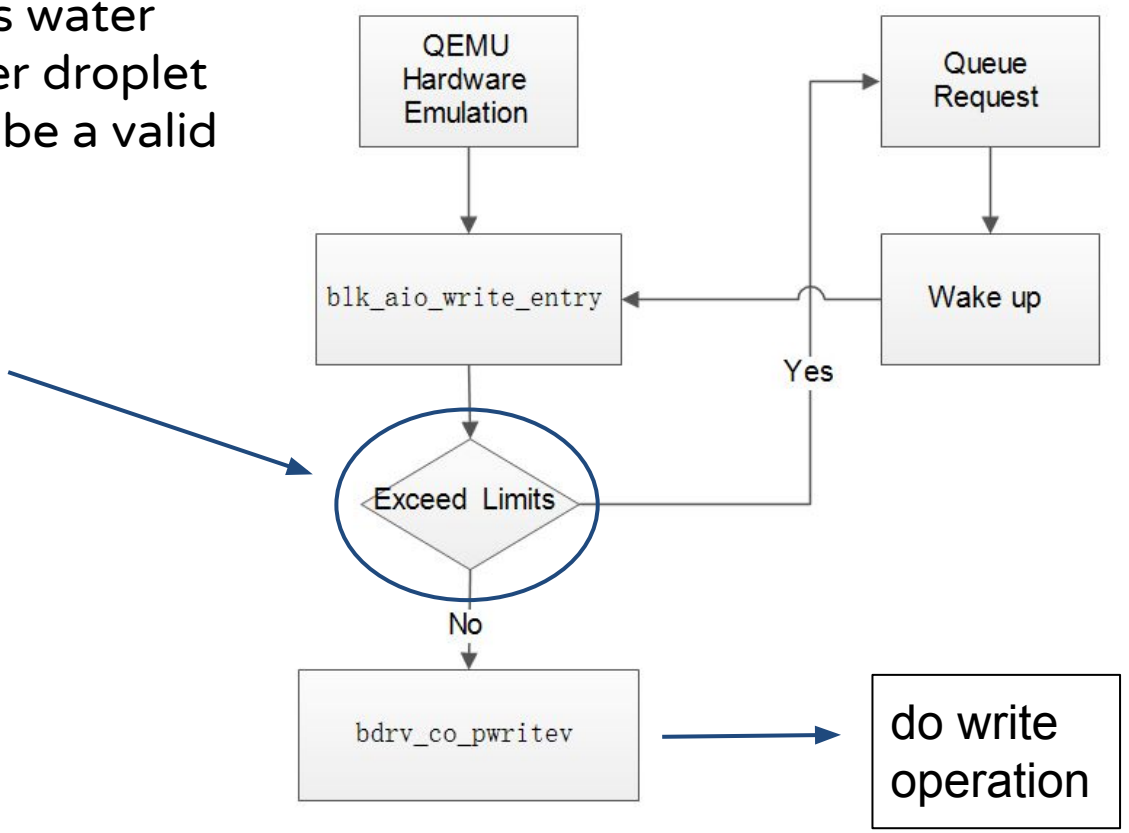
If io operation is described as water droplet, where does the water droplet flow, then it is considered to be a valid io operation by Guest OS?

In flow:

pass throttle hook point

In leaky bucket model:

in bucket

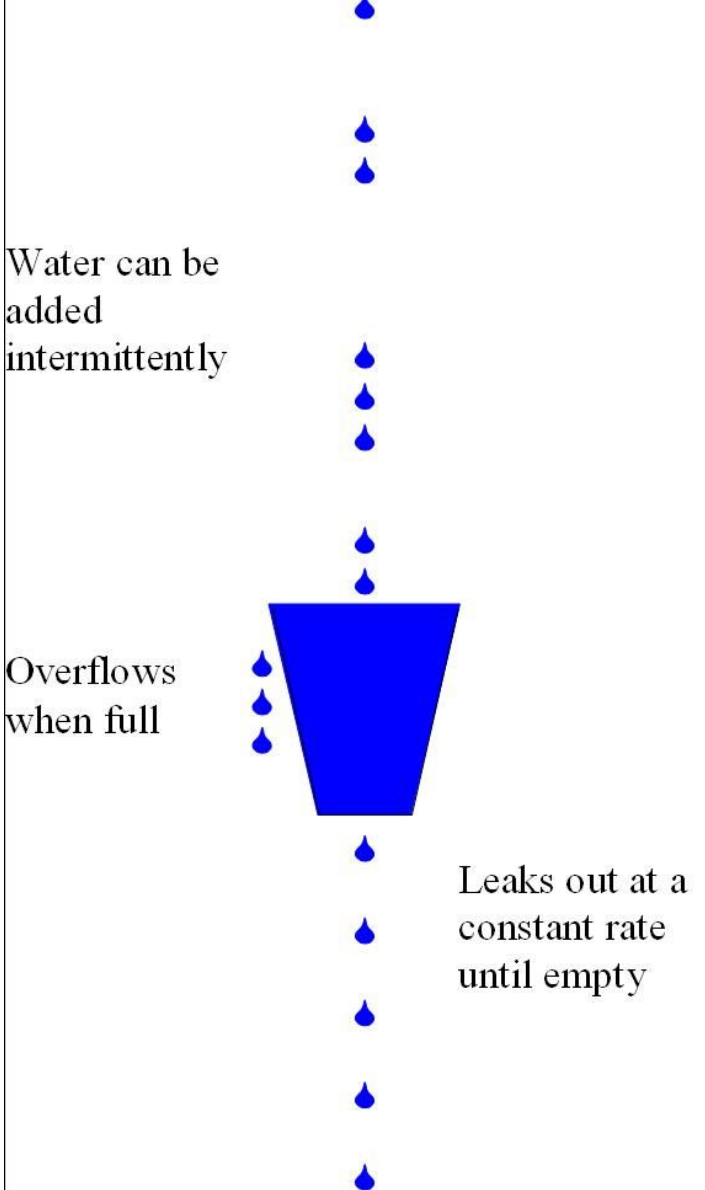


The Leaky Bucket algorithm

- Rake of water leaks from the bucket
- Max rate of water can be added in the bucket
- The max rate of max time

QEMU Parameters:

```
throttling.<iops | bps>-total
throttling.<iops | bps>-total-max
throttling.<iops | bps>-total-max-length

throttling.<iops | bps>-read
throttling.<iops | bps>-read-max
throttling.<iops | bps>-read-max-length

throttling.<iops | bps>-write
throttling.<iops | bps>-write-max
throttling.<iops | bps>-write-max-length

throttling.iops-size
```



The Leaky Bucket algorithm(cont.)

```
// util/throttle.c, int64_t throttle_compute_wait(LeakyBucket *bkt)

double extra; /* the number of extra units blocking the io */
double bucket_size;   /* I/O before throttling to bkt->avg */
double burst_bucket_size; /* Before throttling to bkt->max */
...
if (!bkt->max) {
      /* If bkt->max is 0 we still want to allow short bursts of I/O
       * from the guest, otherwise every other request will be throttled
       * and performance will suffer considerably. */
      bucket_size = (double) bkt->avg / 10;
      burst_bucket_size = 0;
} else {
      /* If we have a burst limit then we have to wait until all I/O
       * at burst rate has finished before throttling to bkt->avg */
      bucket_size = bkt->max * bkt->burst_length;
      burst_bucket_size = (double) bkt->max / 10;
}

/* If the main bucket is full then we have to wait */
extra = bkt->level - bucket_size;
if (extra > 0) {
    return throttle_do_compute_wait(bkt->avg, extra);
}
/* If the main bucket is not full yet we still have to check the
 * burst bucket in order to enforce the burst limit */
if (bkt->burst_length > 1) {
    assert(bkt->max > 0); /* see throttle_is_valid() */
    extra = bkt->burst_level - burst_bucket_size;
    if (extra > 0) {
        return throttle_do_compute_wait(bkt->max, extra);
    }
}
...
```

Glossary:

set in configuration

- ❏ `blk->avg -- *-total`
- ❏ `blk->max -- *-total-max`
- ❏ `blk->burst_length -- *-total-max-length`

vary in running environment

- ❏ `blk->level` -- account of operation
- ❏ `blk->burst_level` -- account of burst operation

Show:

- Bucket size
  - `bucket_size`
  - `burst_bucket_size`
- wait condition
  - when normal
  - when burst

The Leaky Bucket algorithm(cont.)

```
// util/throttle.c

static int64_t throttle_compute_wait_for(ThrottleState *ts,
                                         bool is_write)
{
    BucketType to_check[2][4] = { {THROTTLE_BPS_TOTAL,
                                   THROTTLE_OPS_TOTAL,
                                   THROTTLE_BPS_READ,
                                   THROTTLE_OPS_READ},
                                  {THROTTLE_BPS_TOTAL,
                                   THROTTLE_OPS_TOTAL,
                                   THROTTLE_BPS_WRITE,
                                   THROTTLE_OPS_WRITE}, };
    int64_t wait, max_wait = 0;
    int i;

    for (i = 0; i < 4; i++) {
        BucketType index = to_check[is_write][i];
        wait = throttle_compute_wait(&ts->cfg.buckets[index]);
        if (wait > max_wait) {
            max_wait = wait;
        }
    }

    return max_wait;
}
```

Show:

- wait time

wake up condition???

The Leaky Bucket algorithm(cont.)

```
blk_co_pwritev
 -- throttle_group_co_io_limits_intercept
     -- throttle_group_schedule_timer
        -- throttle_schedule_timer
             -- throttle_compute_timer
                 -- throttle_do_leak
                     -- throttle_leak_bucket
                 -- throttle_compute_wait_for
```

```
// util/throttle.c

void throttle_leak_bucket(LeakyBucket *bkt, int64_t delta_ns)
{
    double leak;

    /* compute how much to leak */
    leak = (bkt->avg * (double) delta_ns) / NANOSECONDS_PER_SECOND;

    /* make the bucket leak */
    bkt->level = MAX(bkt->level - leak, 0);

    /* if we allow bursts for more than one second we also need to
     * keep track of bkt->burst_level so the bkt->max goal per second
     * is attained */
    if (bkt->burst_length > 1) {
        leak = (bkt->max * (double) delta_ns) / NANOSECONDS_PER_SECOND;
        bkt->burst_level = MAX(bkt->burst_level - leak, 0);
    }
}
```

Note:

- ❏ blk->avg -- total
- ❏ blk->max -- total-max
- ❏ blk->burst_length -- total-max-length
- ❏ blk->level -- account of operation
- ❏ blk->burst_level -- account of burst operation

Show:

- reset operation account used judging

Default value & Note

	Default value	meaning
`*-<total | read | write>`	0	unlimited
`*-max`	0	bursts not allowed
`*-max-length`	1	burst time keep 1 second
`iops-size`	0	not fragmented

Note:

- `total, read/write` cannot be used at the same time
- `throttling.iops-size` requires an iops value to be set
- `*-max` requires `*-<total | read | write>`, `*-max-length` requires `*-max`
- More information at *`throttle_is_valid`* function in *util/throttle.c*

Huge IO request

Question:
The user can take advantage of huge I/O request instead of several smaller ones to circumvent the limits.
This means: iops is not met the limit, but bps has already met.

solution: `throttling.iops-size`

Huge IO request(cont.)

how does the `iops-size` work?

```
// util/throttle.c

void throttle_account(ThrottleState *ts, bool is_write, uint64_t size)
{
    ...
    const BucketType bucket_types_units[2][2] = {
        { THROTTLE_OPS_TOTAL, THROTTLE_OPS_READ },
        { THROTTLE_OPS_TOTAL, THROTTLE_OPS_WRITE }
    };
    double units = 1.0;
    ...
    /* if cfg.op_size is defined and smaller than size we compute unit count */
    if (ts->cfg.op_size && size > ts->cfg.op_size) {
        units = (double) size / ts->cfg.op_size;
    }

    for (i = 0; i < 2; i++) {
        LeakyBucket *bkt;
        ...

        bkt = &ts->cfg.buckets[bucket_types_units[is_write][i]]; // for IOPS
        bkt->level += units;
        if (bkt->burst_length > 1) {
            bkt->burst_level += units;
        }
    }
}
```

When `iops-size` were setted, large than <iops-size> unit would be fragemented.

Meaning

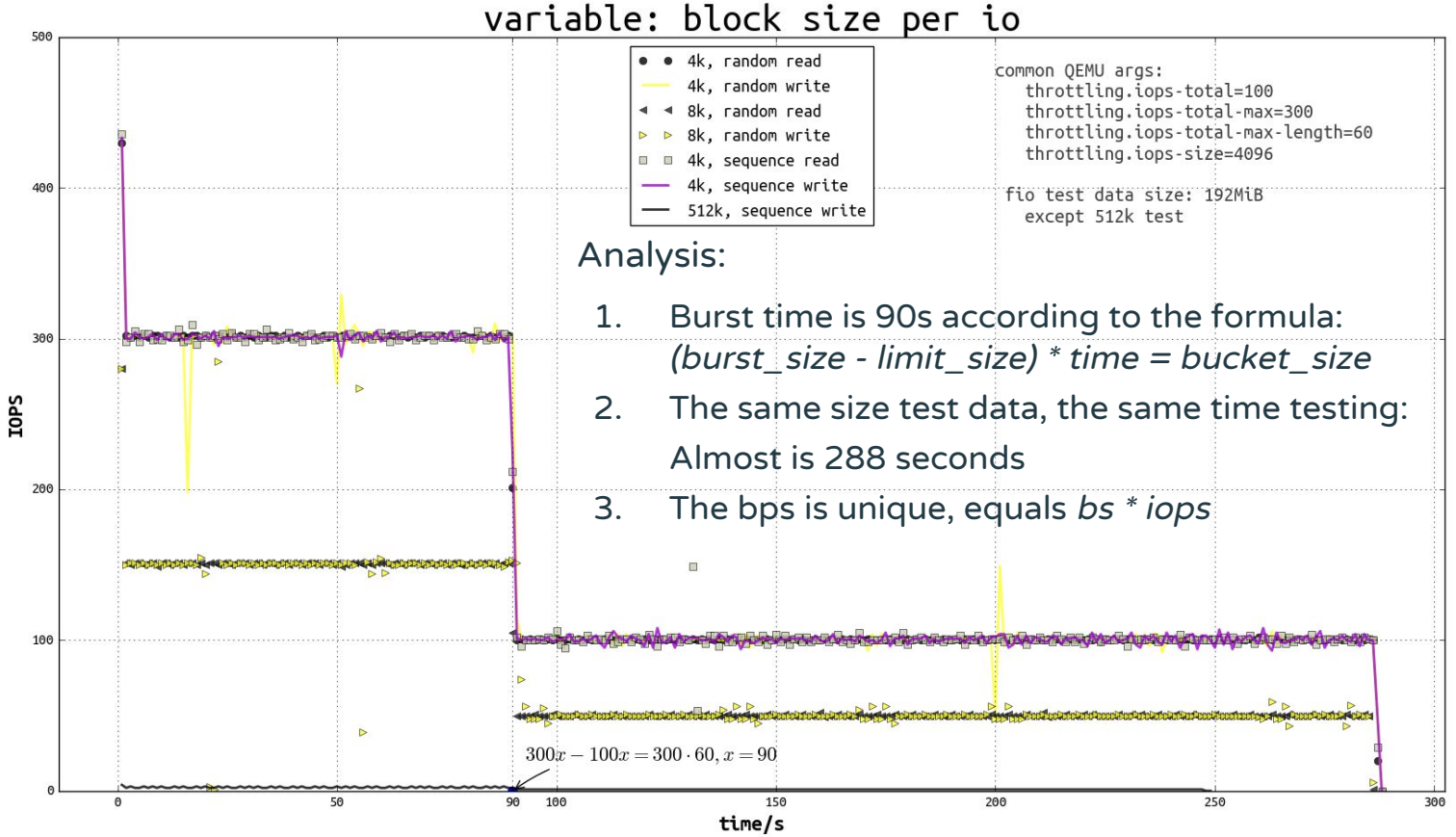
For example:

```
-drive file=hd0.qcow2,
  throttling.iops-total=100
  throttling.iops-total-max=2000
  throttling.iops-total-max-length=60
  throttling.iops-size=4096
```

Perform I/O on hd0.qcow2 at a rate of 2000 IOPS for 1 minute before it's throttled down to 100 IOPS.
Any larger than 4KiB request will be counted, for example, 8KiB request will be counted as two, 6KiB request will be counted as one and a half.

Test

proof the parameters is effective



Analysis:

1. Burst time is 90s according to the formula: *(burst_size - limit_size) * time = bucket_size*
2. The same size test data, the same time testing: Almost is 288 seconds
3. The bps is unique, equals *bs * iops*

Disk Group

Example:

```
-drive file=hd1.qcow2,throttling.iops-total=6000,throttling.group=foo
-drive file=hd2.qcow2,throttling.iops-total=6000,throttling.group=foo
-drive file=hd3.qcow2,throttling.iops-total=3000,throttling.group=bar
-drive file=hd4.qcow2,throttling.iops-total=6000,throttling.group=foo
-drive file=hd5.qcow2,throttling.iops-total=3000,throttling.group=bar
-drive file=hd6.qcow2,throttling.iops-total=5000
```

Means:

Hd1, hd2 and hd4 are all members of a group named ‘foo’; hd3 and hd5 are members of ‘bar’, hd6 is left alone.

Note:

1. **grouping drives all share the same limits**
2. Unlimit means no group
3. Every limited disk has its default group
4. The same group, the same settings
5. Multiple groups or added new group, the last wins

Usage

QEMU:

```
qemu-system-x86_64 \
  ...
  -device virtio-blk-pci,drive=test1 \
  -drive if=none,file=test-1.qcow2,format=qcow2,id=test1,throttling.iops-total=1000 \
  ...
```

Virsh:

```
blkdeviotune <domain> <image absolute path> [--total-bytes-sec <number>]
[--read-bytes-sec <number>] [--write-bytes-sec <number>] [--total-iops-sec <number>]
[--read-iops-sec <number>] [--write-iops-sec <number>] [--total-bytes-sec-max <number>]
[--read-bytes-sec-max <number>] [--write-bytes-sec-max <number>] [--total-iops-sec-max <number>]
[--read-iops-sec-max <number>] [--write-iops-sec-max <number>] [--size-iops-sec <number>]
[--group-name <string>] [--total-bytes-sec-max-length <number>] [--read-bytes-sec-max-length
<number>] [--write-bytes-sec-max-length <number>] [--total-iops-sec-max-length <number>]
[--read-iops-sec-max-length <number>] [--write-iops-sec-max-length <number>] [--config] [--live]
[--current]
```

Usage(cont.)

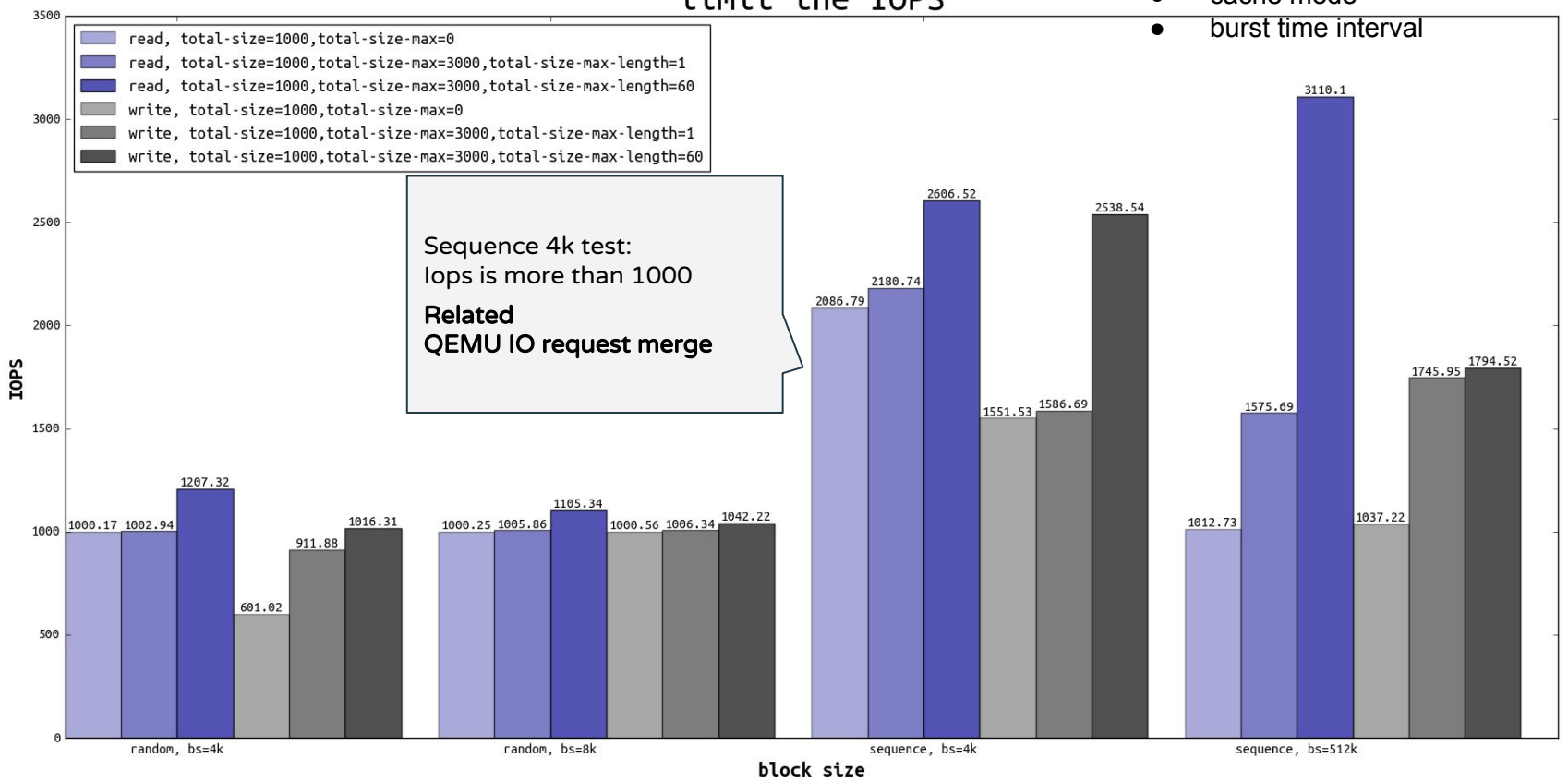
```
<domain>
  ...
  <blkiotune>
    <weight>800</weight>
    <device>
      <path>/dev/sda</path>
      <weight>1000</weight>
    </device>
    <device>
      <path>/dev/sdb</path>
      <weight>500</weight>
      <read_bytes_sec>10000</read_bytes_sec>
      <write_bytes_sec>10000</write_bytes_sec>
      <read_iops_sec>20000</read_iops_sec>
      <write_iops_sec>20000</write_iops_sec>
    </device>
  </blkiotune>
  ...
</domain>
```

From https://libvirt.org/formatdomain.html#elementsBlockTuning

IO Test Exception

not consider, main:

- IO merge
- QCOW2 l2-cache
  - here use QCOW2 format
- cache mode
- burst time interval

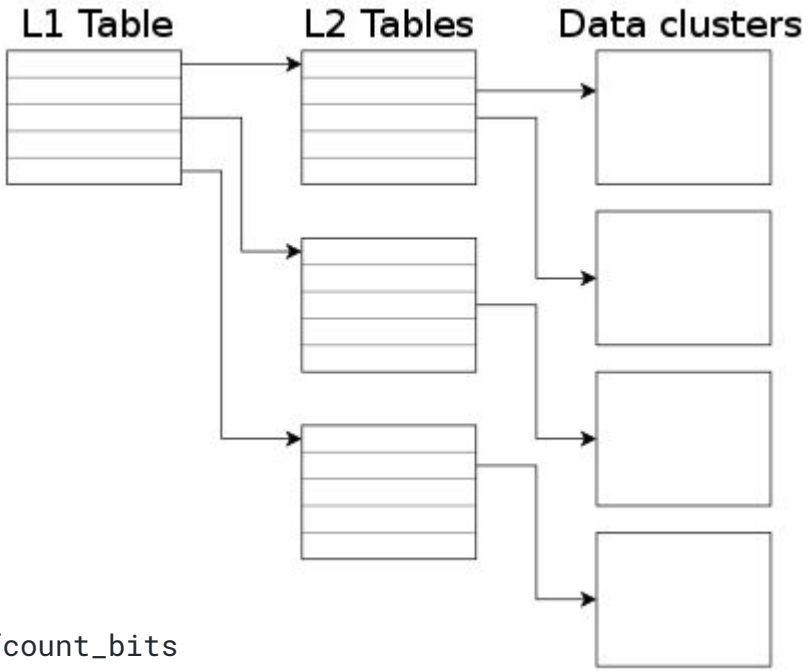


IO Test Exception -- merge

```
/*  hw/block/virtio-blk.c */

static void virtio_blk_submit_multireq(BlockBackend *blk, MultiReqBuffer *mrb)
{
    ...
    max_transfer = blk_get_max_transfer(mrb->reqs[0]->dev->blk);

    qsort(mrb->reqs, mrb->num_reqs, sizeof(*mrb->reqs), // there is a merge operation from here
          &multireq_compare);

    for (i = 0; i < mrb->num_reqs; i++) {
        VirtIOBlockReq *req = mrb->reqs[i];
        if (num_reqs > 0) {
            /*
             * NOTE: We cannot merge the requests in below situations:
             * 1. requests are not sequential
             * 2. merge would exceed maximum number of IOVs
             * 3. merge would exceed maximum transfer length of backend device
             */
            if (sector_num + nb_sectors != req->sector_num ||
                niov > blk_get_max_iov(blk) - req->qiov.niov ||
                req->qiov.size > max_transfer ||
                nb_sectors > (max_transfer -
                              req->qiov.size) / BDRV_SECTOR_SIZE) {
                submit_requests(blk, mrb, start, num_reqs, niov);
                num_reqs = 0;
            }
        }
        ...
```

QCOW2 l2-cache Influence

background



find out where that data is located from l2 table

- one single L1 table per disk image, stored in memory
- maybe many L2 tables depending on how much space has been allocated in the image

find out how many times that cluster is used from refcounts table because copy-on-write

- 0 means that the cluster is free
- 1 means that it is used
- >= 2 means that it is used and any write access must perform a copy-on-write operation

```
disk_size = l2_cache_size * cluster_size / 8
disk_size = refcount_cache_size * cluster_size * 8 / refcount_bits
```

"l2-cache-size": maximum size of the L2 table cache
"refcount-cache-size": maximum size of the refcount block cache
"cache-size": maximum size of both caches combined

Reference: qcow2 文件格式详解(I) linux-kvm.org qcow2 { why (not)? qcow2-cache.txt

QCOW2 l2-cache Influence(cont.)

All image:
- cluster_size = 64KiB
- refcount_bits = 16
- disk_size = 16GiB

For vdb, l2-cache conver 16Gib:
- l2_cache_size = 2097152
- refcount_cache_size = 524288

For vdc, l2-cache cover 128MiB:
- l2_cache_size = 16384
- refcount_cache_size = 4096

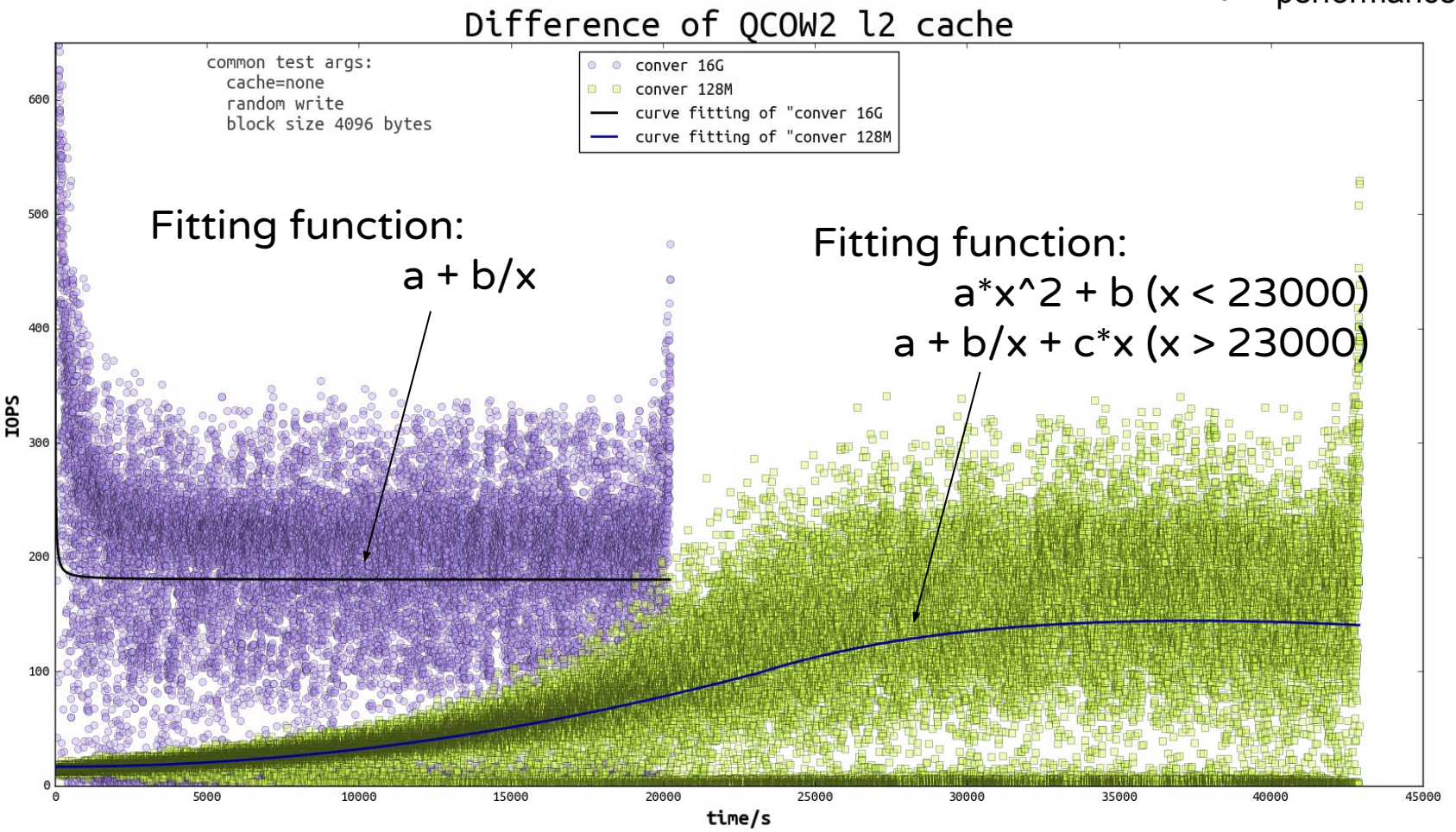
```
# test environment
qemu-system-x86_64 \
    -enable-kvm \
    -cpu host \
    -smp cpus=4,cores=4,threads=1,sockets=1 \
    -m 4G \
    -balloon virtio \
    -device virtio-blk-pci,drive=vda \
    -drive  if=none,file=opensuse42.3.qcow2,format=qcow2,id=vda \
    -device virtio-blk-pci,drive=vdb \
    -drive if=none,file=qcow2-3.qcow2,format=qcow2,id=vdb,l2-cache-size=2097152,cache-size=2621440,cache=none \
    -device virtio-blk-pci,drive=vdc \
    -drive if=none,file=qcow2-4.qcow2,format=qcow2,id=vdc,l2-cache-size=16384,cache-size=20480,cache=none \
    ...

the disk is monopolized by test image, no one use the disk when testing

# test command
fio -direct=1 -sync=0 -size=14G -ioengine=pvsync -bs=4096 ...
```

fio command means:
- `open(file, mode | (O_DIRECT & ~O_SYNC))`
- `pread` / `pwrite` system call

QCOW2 l2-cache Influence(cont.)

Influence:

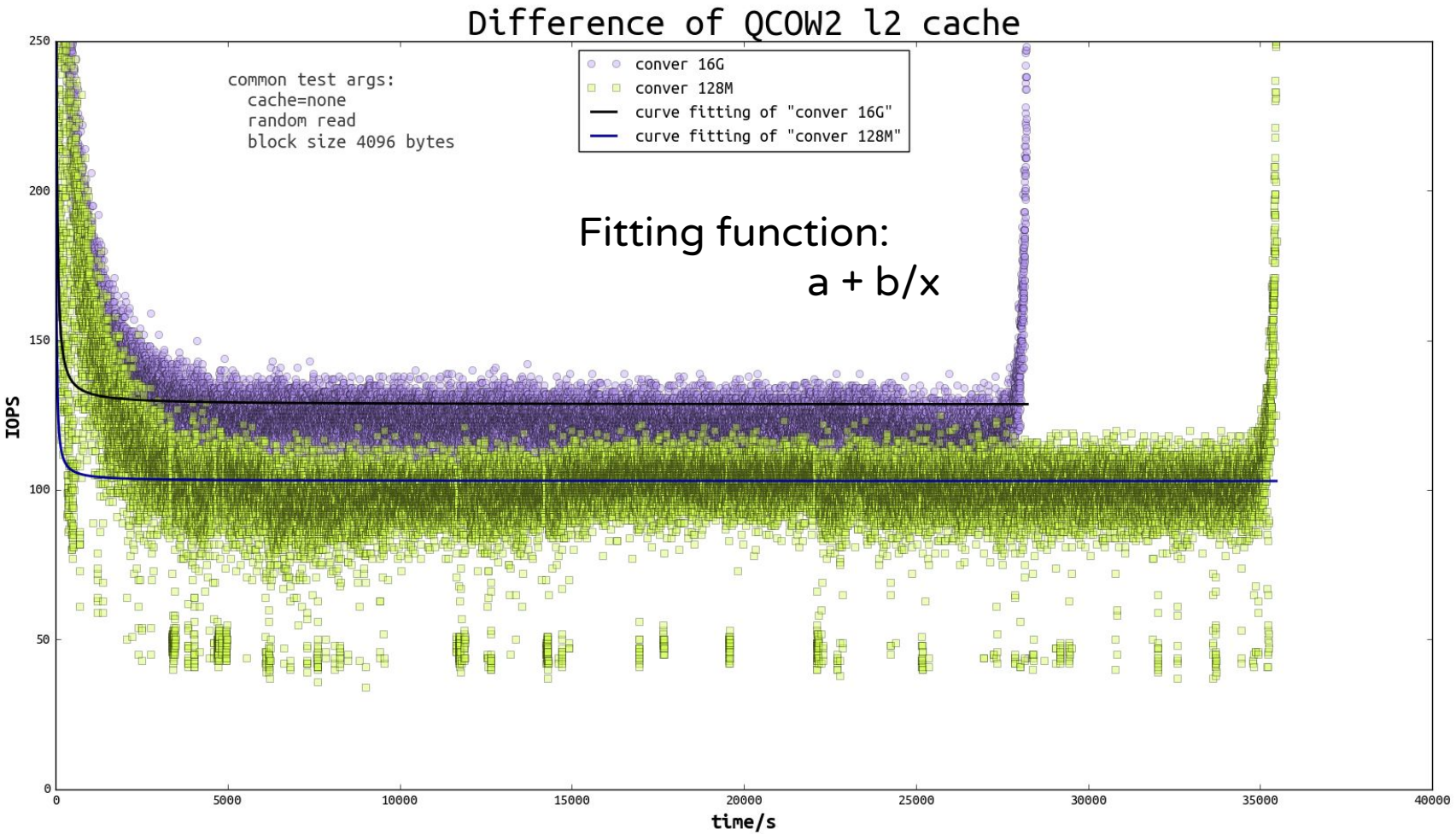
- growing curve
- performance



QCOW2 l2-cache Influence(cont.)

Influence:

- performance



Cache Mode Influence

Comparison

Mode	Guest disk write cache	Host page cache
writeback	on	on
writethrough	off	on
none	on	off
directsync	off	off
unsafe	ignore	on

Note:

- unsafe vs writeback: no flush command

Reference: An Updated Overview of the QEMU Storage Stack IBM 2011 & Antoine Ginies Slides

Cache Mode Influence(cont.)

Implementation

```
drive_new
 -- value = qemu_opt_get(all_opts, "cache");
    bdrv_parse_cache_mode(value, &flags, &writethrough) // first store value in all_opts, then bs_opts
    blockdev_init
     -- blk_set_enable_write_cache(blk, writethrough);
         -- blk->enable_write_cache = wce;
```

```
int bdrv_parse_cache_mode(const char *mode, int *flags, bool *writethrough)
{
    *flags &= ~BDRV_O_CACHE_MASK; // #define BDRV_O_CACHE_MASK  (BDRV_O_NOCACHE | BDRV_O_NO_FLUSH)

    if (!strcmp(mode, "off") || !strcmp(mode, "none")) {
        *writethrough = false;
        *flags |= BDRV_O_NOCACHE;
    } else if (!strcmp(mode, "directsync")) {
        *writethrough = true;
        *flags |= BDRV_O_NOCACHE;
    } else if (!strcmp(mode, "writeback")) {
        *writethrough = false;
    } else if (!strcmp(mode, "unsafe")) {
        *writethrough = false;
        *flags |= BDRV_O_NO_FLUSH;
    } else if (!strcmp(mode, "writethrough")) {
        *writethrough = true;
    } else {
        return -1;
    }

    return 0;
}
```

Reference: [QEMU block cache参数分析]

Cache Mode Influence(cont.)

Implementation

mode	BDRV_O_NOCACHE	BDRV_O_NO_FLUSH	blk->enable_write_cache
writeback	off	off	false
writethrough	off	off	true
none	on	off	false
directsync	on	off	true
unsafe	off	on	false

Cache Mode Influence(cont.)

Implementation

```
// block/file-posix.c
.bdrv_file_open = raw_open
 -- raw_open_common
     -- raw_parse_flags(bdrv_flags, &s->open_flags);
         -- /* Use O_DSYNC for write-through caching, no flags for write-back caching,
             * and O_DIRECT for no caching. */
             if ((bdrv_flags & BDRV_O_NOCACHE)) {
                 *open_flags |= O_DIRECT;
             }
            fd = qemu_open(filename, s->open_flags, 0644);
             -- open
```

Cache Mode Influence(cont.)

Implementation

```
blk_aio_write_entry
 -- blk_co_pwritev
     -- if (!blk->enable_write_cache) {
            flags |= BDRV_REQ_FUA;
        }
        bdrv_co_pwritev
        bdrv_aligned_pwritev
         -- bdrv_driver_pwritev
             -- if (ret == 0 && (flags & BDRV_REQ_FUA)) {
                    ret = bdrv_co_flush(bs);
                }
                 -- } else if (bs->drv->bdrv_aio_flush) { // for QCOW2
                        acb = bs->drv->bdrv_aio_flush(bs, bdrv_co_io_em_complete, &co);

.bdrv_aio_flush = raw_aio_flush
 -- if (bs->open_flags & BDRV_O_NO_FLUSH) {
        goto flush_parent;
    }
    ...
    paio_submit
     -- thread_pool_submit_aio(pool, aio_worker, acb, cb, opaque)
         -- handle_aiocb_flush
             -- qemu_fdatasync
    ...
    flush_parent:
    ...
```

```
int qemu_fdatasync(int fd)
{
#ifdef CONFIG_FDATASYNC
    return fdatasync(fd);
#else
    return fsync(fd);
#endif
}
```

Cache Mode Influence(cont.)

Implementation Summary

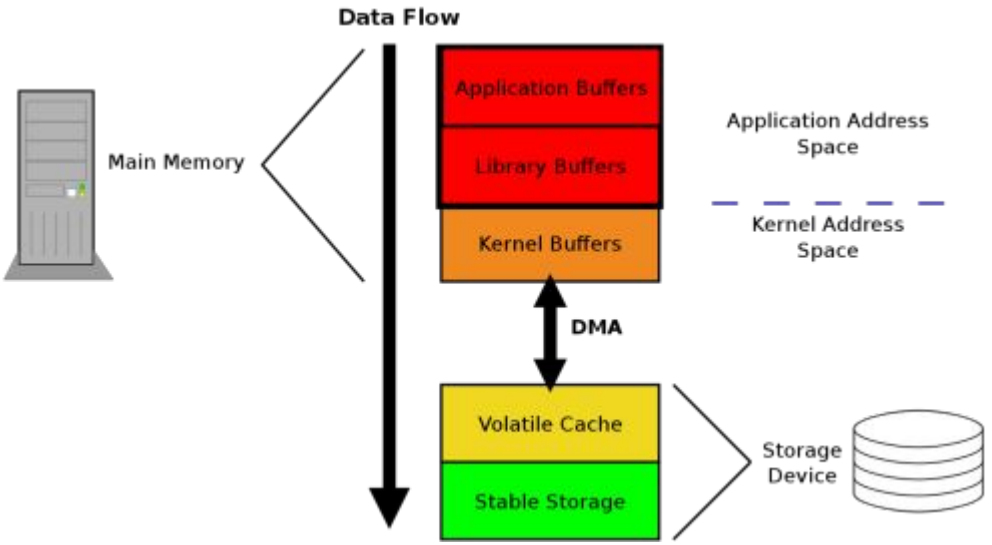
Mode	Guest disk write cache (`fdatasync`)	Host page cache (`O_DIRECT`)
writeback	timing	`` `mode & (~O_DIRECT)` ``
writethrough	every request	`` `mode & (~O_DIRECT)` ``
none	timing	`` `mode | O_DIRECT` ``
directsync	every request	`` `mode | O_DIRECT` ``
unsafe	no	`` `mode & (~O_DIRECT)` ``

Cache Mode Influence(cont.)

O_DIRECT vs O_DSYNC

O_DIRECT try to **minimize cache effects** of the I/O to and from this file. **Don't guarantee completely cross cache**. Kernel will avoid copying data from user space to kernel space, and will instead write it directly via DMA (Direct memory access; if possible).

O_SYNC guarantees that **the call will not return before all data has been transferred to the disk (as far as the OS can tell)**. This still does not guarantee that the data isn't somewhere in the hard disk write cache, but it is as much as the OS can guarantee.



Reference:
[How are the O_SYNC and O_DIRECT flags in open(2) different/alike?]
[Ensuring data reaches disk]
[Linux DirectIO机制分析]

Cache Mode Influence(cont.)

test result

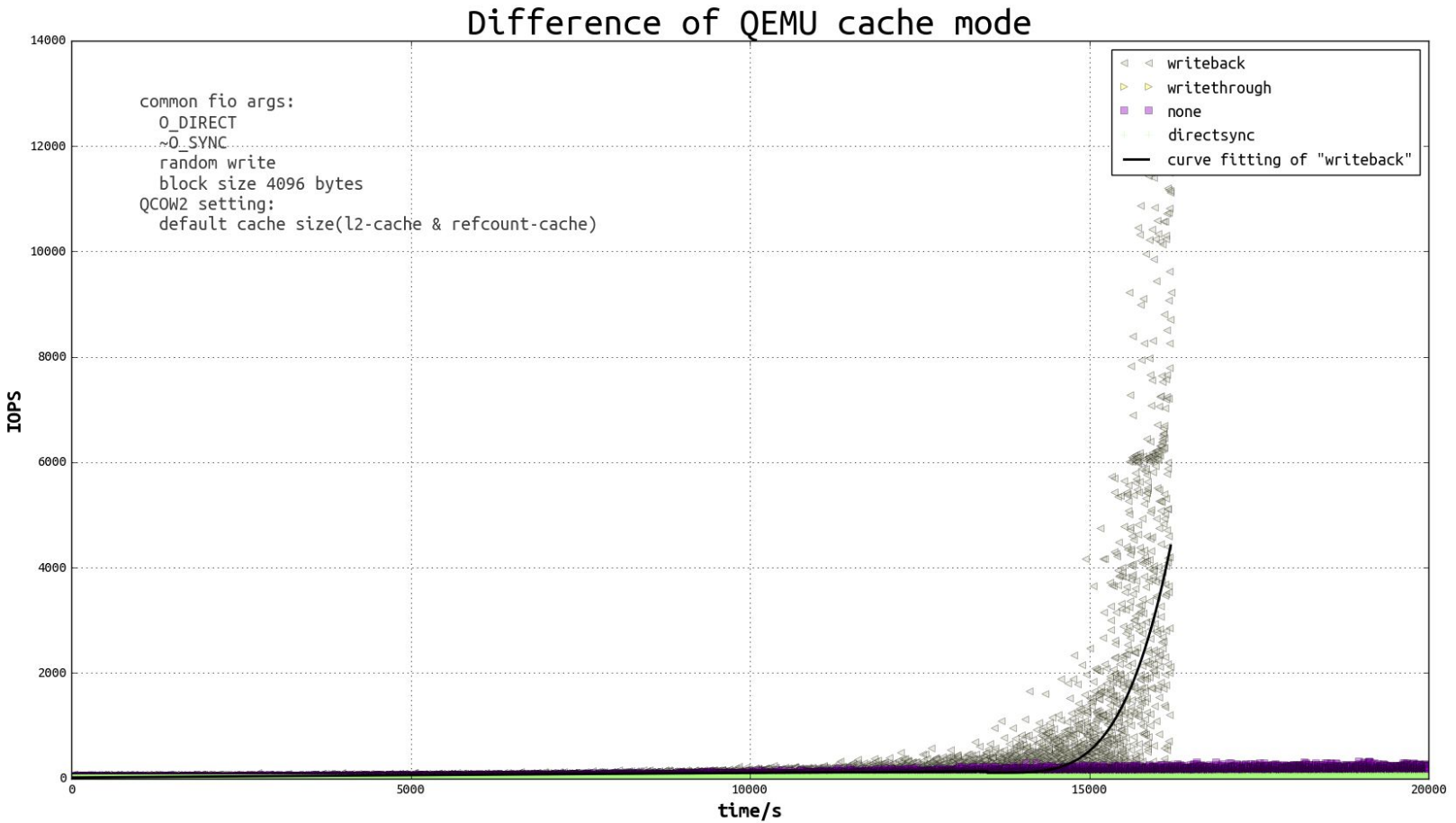
model	random read 4KiB	random write 4KiB
writeback	14286.67	139.76
writethrough	2146.72	59.93
none	104.72	104.33
directsync	107.61	51.58

Note: default setting about disk except cache mode

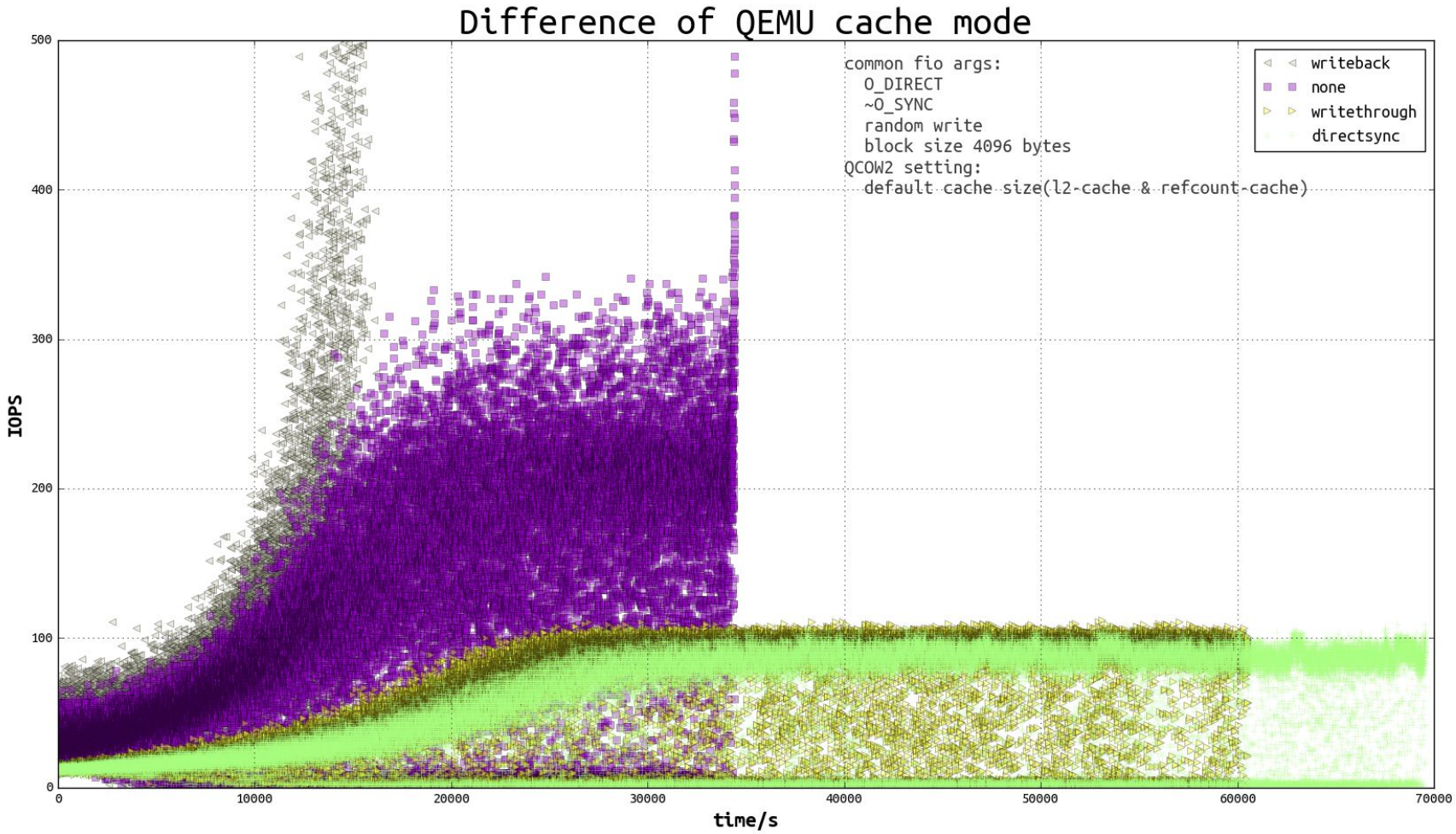
conclusion:

- read performance is better if `mode & ~O_DIRECT`
- guarantee data integrity means write performance is not good

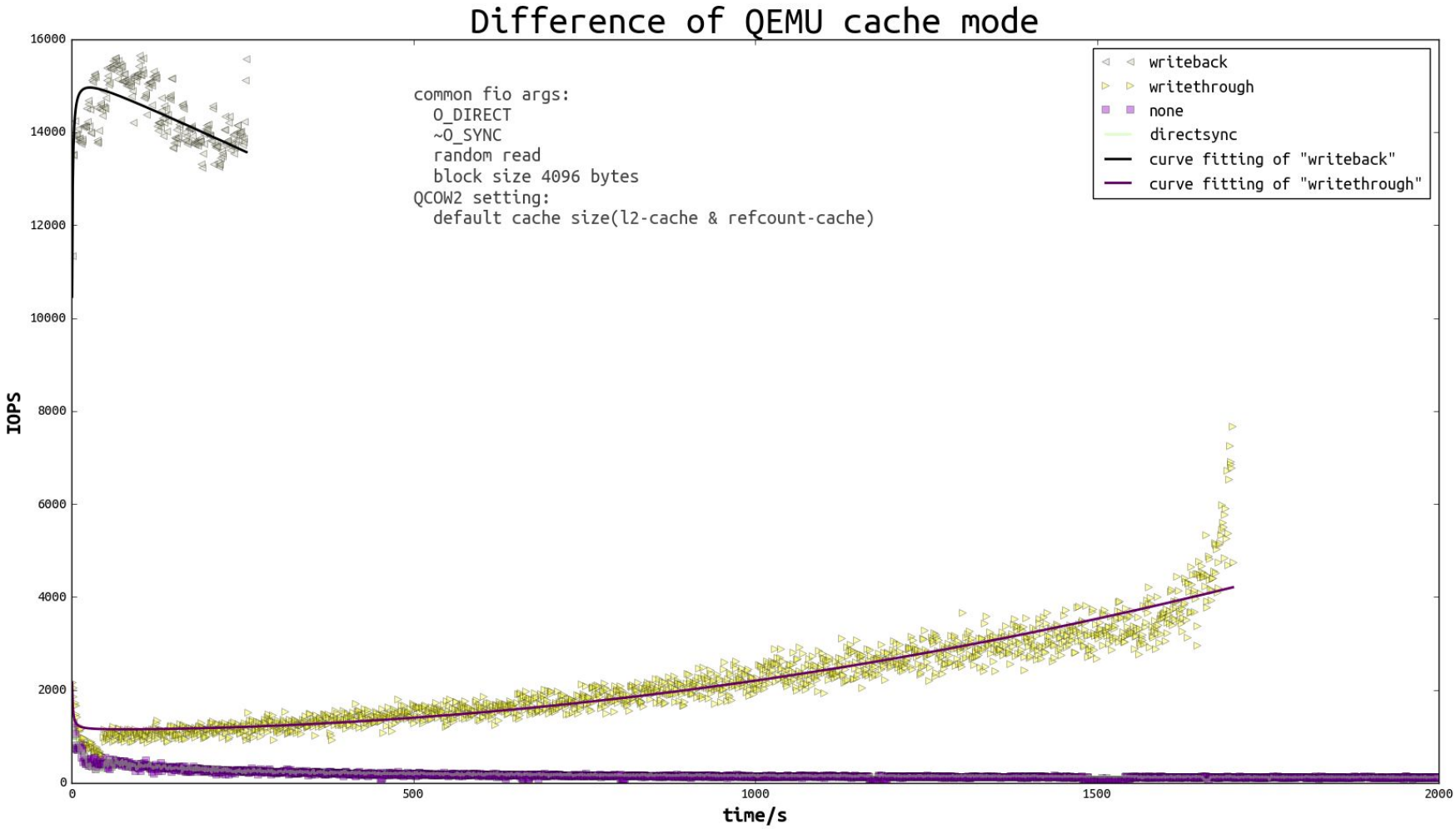
Cache Mode Influence(cont.)



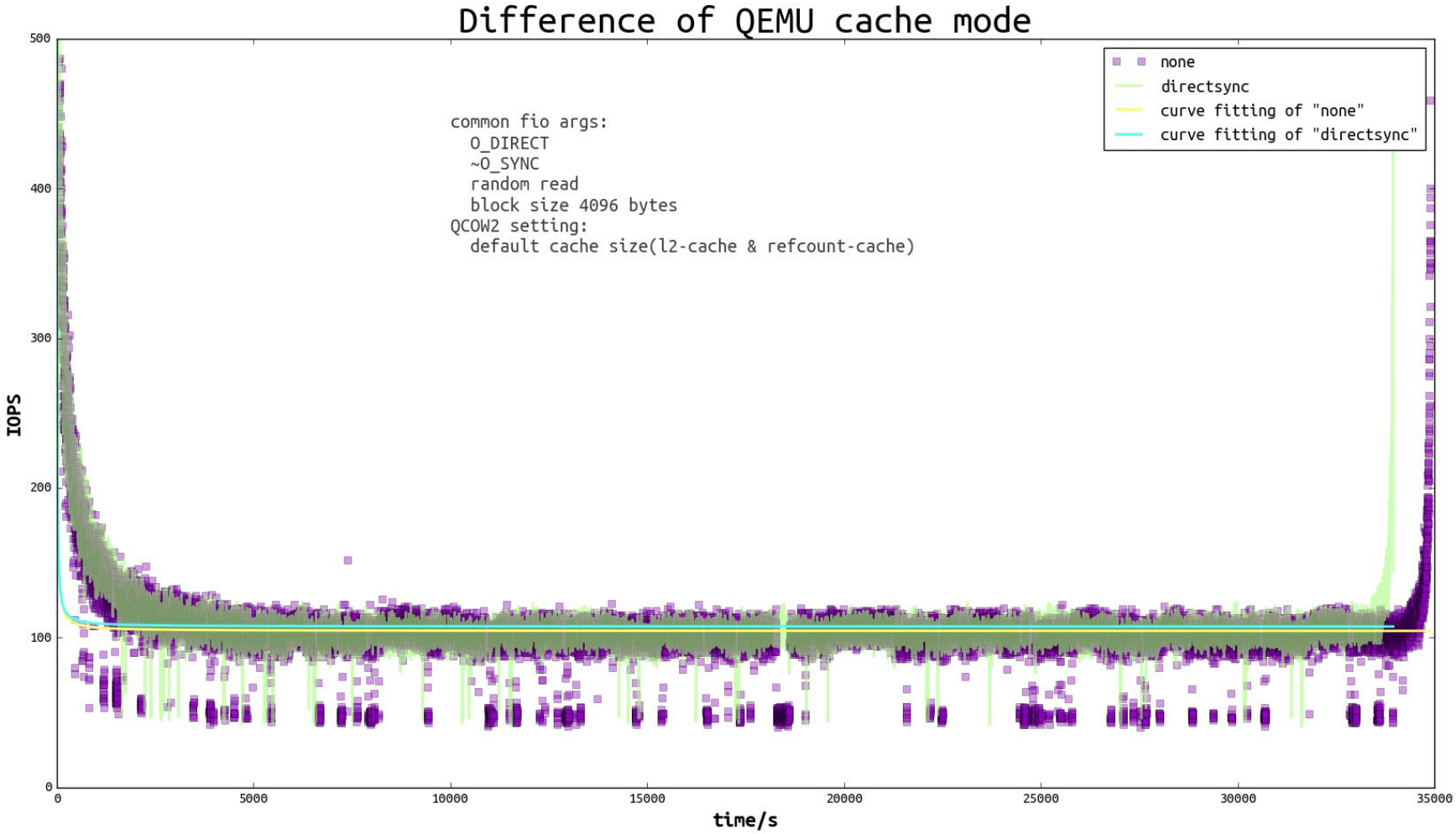
Cache Mode Influence(cont.)



Cache Mode Influence(cont.)



Cache Mode Influence(cont.)



Suggestions

when testing

- set cache size when using QCOW2 image format
- be careful of curve growing
- cut off begin and end result because of unstable

cache mode select

- not suggest `writeback` or `directsync` in product environment
- prefer write performance, select `none`
- prefer data integrity or read performance, select `writethrough`

test data: https://drive.google.com/open?id=17zPRDRr9NOlolKEk_I1kn0fzgz_BHbZp


SUSE
We adapt. You succeed.