facebook

# Faster IO through io_uring

Jens Axboe

Software Engineer, Facebook

Kernel Recipes 2019, Sep 26th 2019

# Rewind one year...

- **read**(2) / **write**(2)
- **pread**(2) / **pwrite**(2)
- **preadv**(2) / **pwritev**(2)
- **preadv2**(2) / **pwritev2**(2)
- **fsync**(2) / **sync_data_range**(2)

# Rewind one year… aio/libaio

- **io_setup**(2) → **io_submit**(2) → **io_getevents**(2)
- Supports read/write, poll, fsync
- Buffered? lol
- **O_DIRECT** always asynchronous? Nope
- Efficiency
  - System calls
  - Copy
  - Ring buffer
  - Overall performance lacking today

# Adoption

- Limited, `O_DIRECT` is fairly niche
- Which leads to...

```
commit 84c4e1f89fefe70554da0ab33be72c9be7994379
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sun Mar 3 14:23:33 2019 -0800

    aio: simplify - and fix - fget/fput for io_submit()
```

# What do we need – tldr

- Support for missing features
  - Buffered async IO
  - Polled IO
  - New features that allow general overhead reduction
- API that doesn’t suck
- Efficient
  - Low latency
  - High IOPS
  - System call limiting
- Could aio be fixed?

# io_uring

- Yes, I know what it sounds like…
- Merged in v5.1-rc1
  - First posted January 8th 2019
  - Merged March 8th 2019
- So obviously Linus totally loves it

*“So honestly, the big issue is that this is *YET* another likely failed interface that absolutely nobody will use, and that we will have absolutely zero visibility into.”*

*Linus*

*“It will probably have subtle and nasty bugs, not just because nobody tests it, but because that's how asynchronous code works - it's hard.”*

*Linus*

*“And they are security issues too, and they'd never show up in the one or two actual users we might have (because they require that you race with closing the file descriptor that is used asynchronously).”*

*Linus*

*“Or all the garbage direct-IO crap. It's shit. I know the XFS people love it, but it's *still* shit.”*

*Linus*

# Hopeless?

*“So the fundamental issue is that it needs to be so good that I don't go "why isn't this *exactly* the same as all the other failed clever things we've done"?”*

*Linus*

# io_uring

- Yes, I know what it sounds like…
- Merged in v5.1-rc1
  - First posted January 8th 2019
  - Merged March 8th 2019
- So obviously Linus totally loves it
  - Deep down somewhere…

# What is it

- Fundamentally, ring based communication channel
  - Submission Queue, SQ
    - `struct io_uring_sqe`
  - Completion Queue, CQ
    - `struct io_uring_cqe`
- All data shared between kernel and application
- Adds critically missing features
- Aim for easy to use, while powerful
  - Hard to misuse
- Flexible and extendable!

# Ring setup

- `int io_uring_setup(u32 nentries, struct io_uring_params *p);`
  - → returns ring file descriptor

```
struct io_uring_params {
        __u32 sq_entries;
        __u32 cq_entries;
        __u32 flags;
        __u32 sq_thread_cpu;
        __u32 sq_thread_idle;
        __u32 features;
        __u32 resv[4];
        struct io_sqring_offsets sq_off;
        struct io_cqring_offsets cq_off;
};
```

```
struct io_sqring_offsets {
	__u32 head;
	__u32 tail;
	__u32 ring_mask;
	__u32 ring_entries;
	__u32 flags;
	__u32 dropped;
	__u32 array;
	__u32 resv1;
	__u64 resv2;
};
```

# Ring access

```
#define IORING_OFF_SQ_RING              0ULL
#define IORING_OFF_CQ_RING              0x8000000ULL
#define IORING_OFF_SQES                 0x10000000ULL

sq→ring_ptr = mmap(0, sq→ring_sz, PROT_READ | PROT_WRITE,
                   MAP_SHARED | MAP_POPULATE, ring_fd,
                   IORING_OFF_SQ_RING);

sq→khead = sq→ring_ptr + p→sq_off.head;
sq→ktail = sq→ring_ptr + p→sq_off.tail;
[…]
```

# Reading and writing rings

- head and tail indices free running
  - Integer wraps
  - Entry always head/tail masked with ring mask
- App produces SQ ring entries
  - Updates tail, kernel consumes at head
  - `→array[]` holds index into `→sqes[]`
  - Why not directly indexed?
- Kernel produces CQ ring entries
  - Updates tail, app consumes at head
  - →cqes[] indexed directly

# SQEs

```
struct io_uring_sqe {
  __u8 opcode;     /* type of operation for this sqe */
  __u8 flags;      /* IOSQE_ flags */
  __u16 ioprio;   /* ioprio for the request */
  __s32 fd;       /* file descriptor to do IO on */
  __u64 off;      /* offset into file */
  __u64 addr;      /* pointer to buffer or iovecs */
  __u32 len;      /* buffer size or number of iovecs */
  union {
    __u32   misc_flags;
  };
  __u64 user_data; /* data to be passed back at completion time */
};
```

# Filling in a new SQE

```
struct io_uring_sqe *sqe;
unsigned index, tail;

tail = ring->tail;
read_barrier();
/* SQ ring full */
if (tail + 1 == ring->head)
  return FULL;

index = tail & ring->sq_ring_mask;
sqe = &ring->sqes[index];
/* fill in sqe here */

ring->array[index] = index;
write_barrier();
ring->tail = tail + 1;
write_barrier();
```

# CQEs

```
struct io_uring_cqe {
  __u64 user_data;  /* sqe->data submission passed back */
  __s32 res;        /* result code for this event */
  __u32 flags;
};
```

# Finding completed CQE

```
struct io_uring_cqe *cqe;
unsigned head, index;

head = ring->head;
do {
  read_barrier();
  /* cq ring empty */
  if (head == ring->tail)
    break;
  index = head & ring->cq_ring_mask;
  cqe = &ring->cqes[index];
  /* handle done IO */

  head++;
} while (1);

ring->head = head;
write_barrier();
```

# Submitting and reaping IO

- ```
  int io_uring_enter(int ring_fd, u32 to_submit,
                     u32 min_complete, u32 flags,
                     sigset_t *sigset);

  #define IORING_ENTER_GETEVENTS  (1U << 0)
  #define IORING_ENTER_SQ_WAKEUP  (1U << 1)
  ```

- Enables submit AND complete in one system call
- Non-blocking
- Requests can be handled inline
  ```

# Supported operations

```
#define IORING_OP_NOP                    0
#define IORING_OP_READV                  1
#define IORING_OP_WRITEV                 2
#define IORING_OP_FSYNC                  3
#define IORING_OP_READ_FIXED             4
#define IORING_OP_WRITE_FIXED            5
#define IORING_OP_POLL_ADD               6
#define IORING_OP_POLL_REMOVE            7
#define IORING_OP_SYNC_FILE_RANGE        8
#define IORING_OP_SENDMSG                9
#define IORING_OP_RECVMSG               10
#define IORING_OP_TIMEOUT               11
```

# I thought you said “easy to use”..?

- Only two hard problems in computer science
  1) Cache invalidation
  2) Memory ordering
  3) Off-by-one errors

# liburing to the rescue

- Helpers for setup

```
static int setup_ring(struct submitter *s)
{
    struct io_sq_ring *sring = &s->sq_ring;
    struct io_cq_ring *cring = &s->cq_ring;
    struct io_uring_params p;
    int ret, fd;
    void *ptr;

    memset(&p, 0, sizeof(p));

    fd = io_uring_setup(depth, &p);
    if (fd < 0) {
        perror("io_uring_setup");
        return 1;
    }
    s->ring_fd = fd;

    ptr = mmap(0, p.sq_off.array + p.sq_entries * sizeof(__u32),
            PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE, fd,
            IORING_OFF_SQ_RING);
    printf("sq_ring ptr = 0x%p\n", ptr);
    sring->head = ptr + p.sq_off.head;
    sring->tail = ptr + p.sq_off.tail;
    sring->ring_mask = ptr + p.sq_off.ring_mask;
    sring->ring_entries = ptr + p.sq_off.ring_entries;
    sring->flags = ptr + p.sq_off.flags;
    sring->array = ptr + p.sq_off.array;
    sq_ring_mask = *sring->ring_mask;

    s->sqes = mmap(0, p.sq_entries * sizeof(struct io_uring_sqe),
            PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE, fd,
            IORING_OFF_SQES);
    printf("sqes ptr    = 0x%p\n", s->sqes);

    ptr = mmap(0, p.cq_off.cqes + p.cq_entries * sizeof(struct io_uring_cqe),
            PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE, fd,
            IORING_OFF_CQ_RING);
    printf("cq_ring ptr = 0x%p\n", ptr);
    cring->head = ptr + p.cq_off.head;
    cring->tail = ptr + p.cq_off.tail;
    cring->ring_mask = ptr + p.cq_off.ring_mask;
    cring->ring_entries = ptr + p.cq_off.ring_entries;
    cring->cqes = ptr + p.cq_off.cqes;
    cq_ring_mask = *cring->ring_mask;
    return 0;
}
```

```
#include <liburing.h>

struct io_uring ring;
int ret;

ret = io_uring_queue_init(DEPTH, &ring, 0);
```

# liburing to the rescue

- Helpers for setup
- Helpers for submitting IO

```
static int prep_more_ios(struct submitter *s, int max_ios)
{
  struct io_sq_ring *ring = &s->sq_ring;
  unsigned index, tail, next_tail, prepped = 0;

  next_tail = tail = *ring->tail;
  do {
    next_tail++;
    read_barrier();
    if (next_tail == *ring->head)
      break;

    index = tail & sq_ring_mask;
    init_io(s, index);
    ring->array[index] = index;
    prepped++;
    tail = next_tail;
  } while (prepped < max_ios);

  if (*ring->tail != tail) {
    /* order tail store with writes to sqes above */
    write_barrier();
    *ring->tail = tail;
    write_barrier();
  }
  return prepped;
}
```

```
struct io_uring_sqe *sqe;
struct io_uring_cqe *cqe;
struct iovec iov;

sqe = io_uring_get_sqe(ring); ← previous example to here
iov.iov_base = some_addr;
iov.iov_len = some_len;
io_uring_prep_readv(sqe, ring->fd, &iov, 1 offset);

io_uring_submit(ring);

io_uring_wait_cqe(ring, &cqe);
[read cqe]
io_uring_cqe_seen(ring, cqe);
```

# liburing to the rescue

- Helpers for setup
- Helpers for submitting IO
  - Eliminates need for manual memory barriers
- Mix and match raw and liburing without issue
- liburing package contains kernel header as well
- Use it! Don’t be a hero
- **git://git.kernel.dk/liburing**

# liburing at a glance

- `io_uring_queue_{init,exit}();`
- `io_uring_get_sqe();`
- `io_uring_prep_{readv,writev,read_fixed,write_fixed}();`
  `io_uring_prep_{recv,send}msg();`
  `io_uring_prep_poll_{add,remove}();`
  `io_uring_prep_fsync();`
- `io_uring_submit();`
  `io_uring_submit_and_wait();`
- `io_uring_{wait,peek}_cqe();`
- `io_uring_cqe_seen{};`
- `io_uring_{set,get}_data();`

# Feature: Drain flag

- Set **`IOSQE_IO_DRAIN`** in `sqe→flags`
- If set, waits for previous commands to complete
- Eliminates write→write→write, wait for all writes, sync

# Feature: Linked commands

- Form arbitrary length chain of commands
  - “Do this sqe IFF previous sqe succeeds”
- write→write→write→fsync
- read{fileX,posX,sizeX}→write{fileY,posY,sizeY}
  - See liburing *examples/link-cp.c*
- Set **`IOSQE_IO_LINK`** in `sqe→flags`
  - Dependency chain continues until not set
- Ease of programming, system call reductions

# Registering aux functions

- `int io_uring_register(int ring_fd, u32 op, void *arg,`
  `u32 nr_args);`

```
#define IORING_REGISTER_BUFFERS     0
#define IORING_UNREGISTER_BUFFERS   1
#define IORING_REGISTER_FILES       2
#define IORING_UNREGISTER_FILES     3
#define IORING_REGISTER_EVENTFD     4
#define IORING_UNREGISTER_EVENTFD   5
```

# Registered buffers

- Takes a `struct iovec` array as argument
  - Length of array nr_args
- Eliminates `get_user_pages()` in submission path
  - ~100 nsec
- Eliminates `put_pages()` in completion path
- Use with **`IORING_OP_READ_FIXED`**, **`IORING_OP_WRITE_FIXED`**
  - Not `iovec` based
  - `sqe→buf_index` points to index of registered array
  - `sqe→addr` is within buffer, `sqe→len` is length in bytes

# Registered files

- Takes a `s32` array as argument
  - Length of array as nr_args
- Eliminates atomic `fget()` for submission
- Eliminates atomic `fput()` for completion
- Use array index as `fd`
  - Set **`IOSQE_FIXED_FILE`**
- Circular references
  - Setup socket, register both ends with io_uring
  - Pass io_uring fd through socket
  - **https://lwn.net/Articles/779472/**

# Registered eventfd

- Takes a `s32` pointer as argument
  - nr_args ignored
- Allows completion notifications

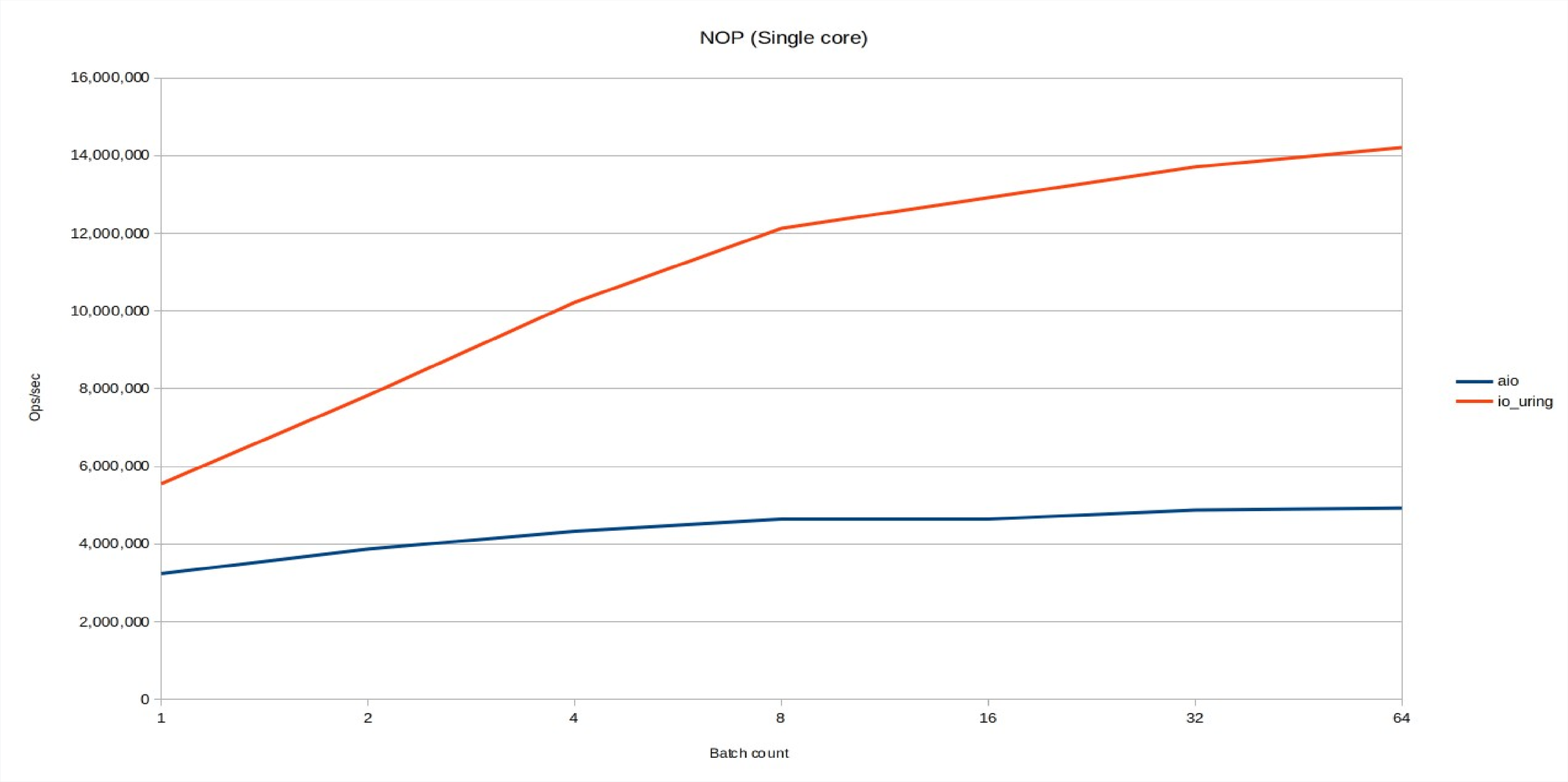
# Polled IO

- Not `poll(2)`
  - Are we there yet?
- Trades CPU usage for latency win
  - Until a certain point
- Absolutely necessary for low latency devices
- Use **`IORING_SETUP_IOPOLL`**
- Submission the same, reaping is polled
- Can’t be mixed with non-polled IO
- Raw bdev support (eg nvme), files on XFS

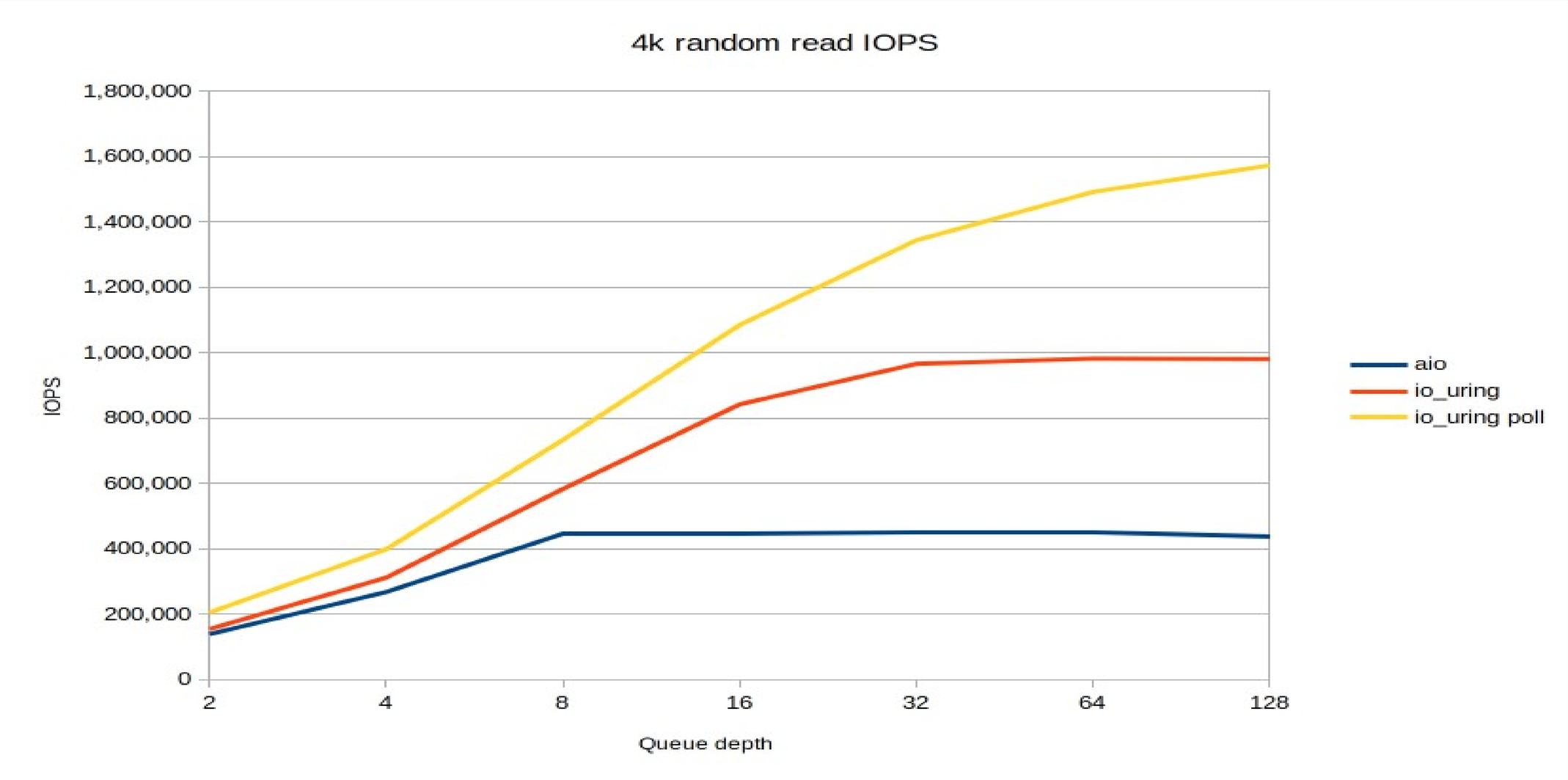
# Polled IO submission

- Use **`IORING_SETUP_SQPOLL`**
  - **`IORING_SETUP_SQ_AFF`**
- Submission now offloaded, reaping is app polled
- Independent of **`IORING_SETUP_IOPOLL`**
- Busy loops for `params→sq_thread_idle` msec when idle
  - Sets `sq_ring→flags |=` **`IORING_SQ_NEED_WAKEUP`**
- Allows splitting submit / complete load onto separate cores

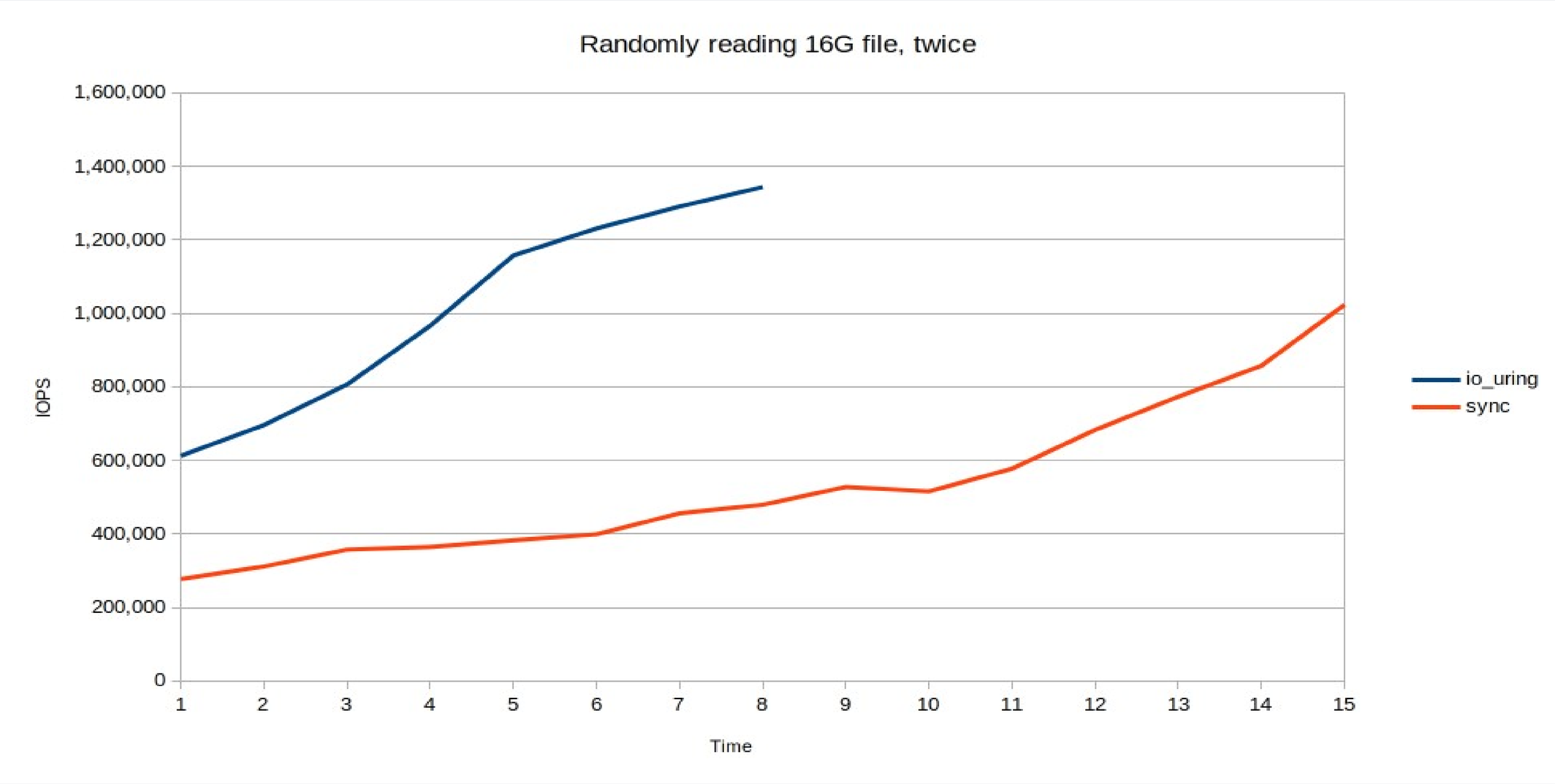
# NOP


NOP (Single core)
16,000,000
14,000,000
12,000,000
10,000,000
8,000,000
6,000,000
4,000,000
2,000,000
0
Ops/sec
1
2
4
8
16
32
64
Batch count
aio
io_uring

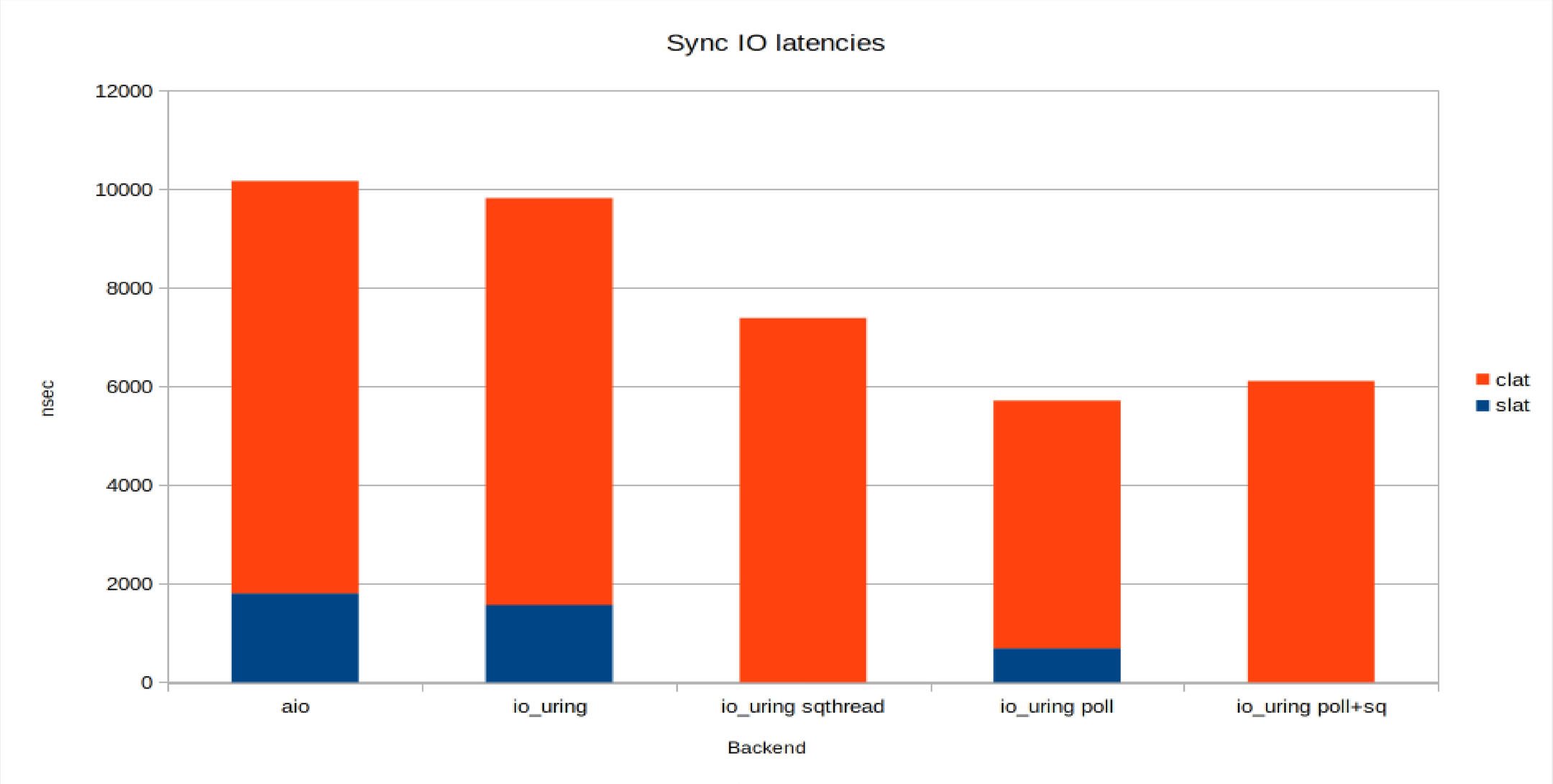
# io_uring vs aio peak



# Buffered perf



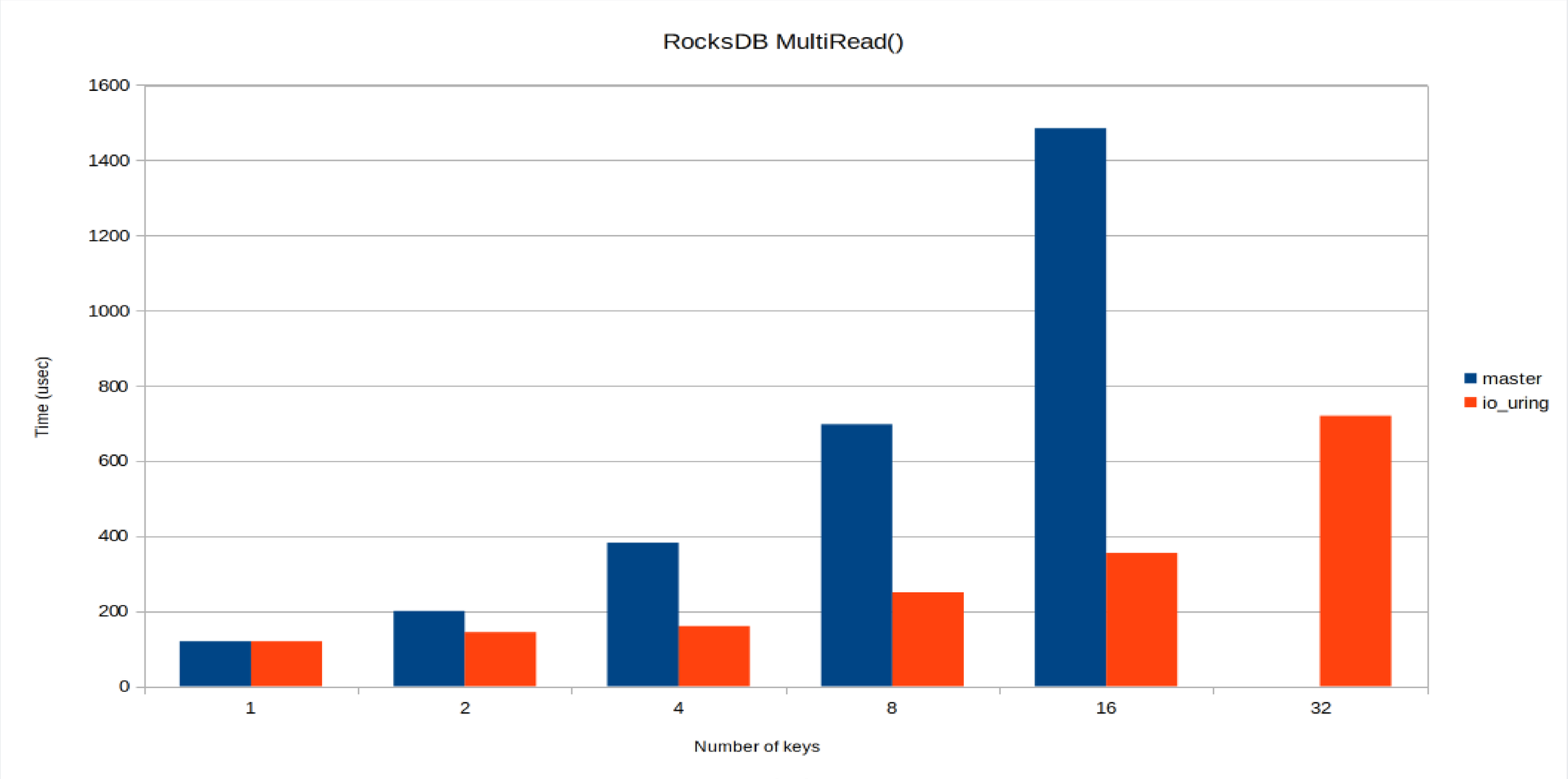
# io_uring vs aio sync



# Adoption

- Rust, C++ I/O executors
- Ceph (bluestore, new backend)
- libuv
- Postgres
- RocksDB (and MyRocks)

# RocksDB MultiRead() test



# Adoption

- Rust, C++ I/O executors
- Ceph (bluestore, new backend)
- libuv
- Postgres
- RocksDB (and MyRocks)
- High performance cases
- TyrDB

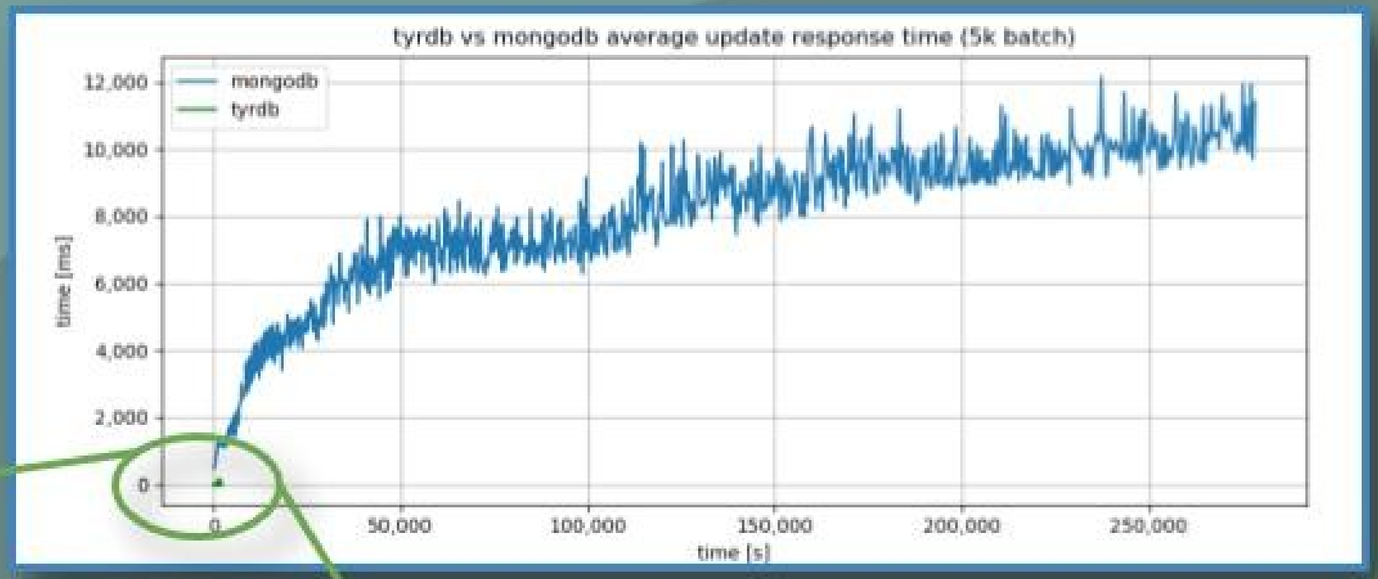
# TyrDB 0.1 vs MongoDB 4.2

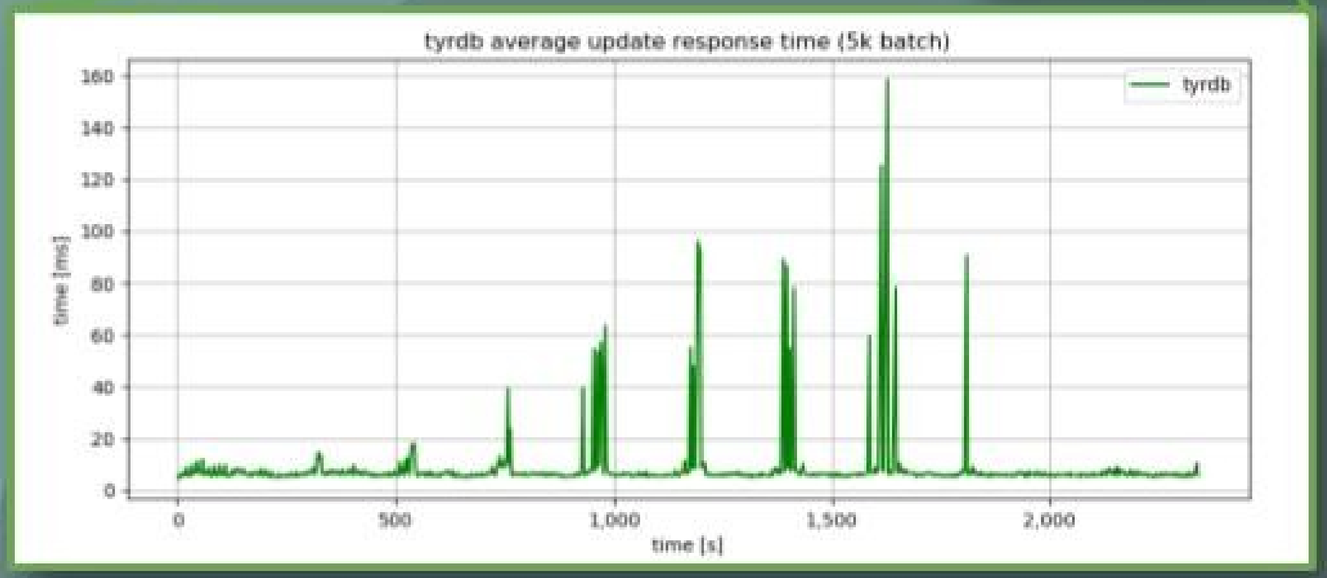
## Benchmark Test 9/9/2019:

Inserting 200M Keys

1 cpu server process

4 GB ram




⬆ MongoDB: 77 hours

⬅ TyrDB: 38 minutes

# Results from the wild

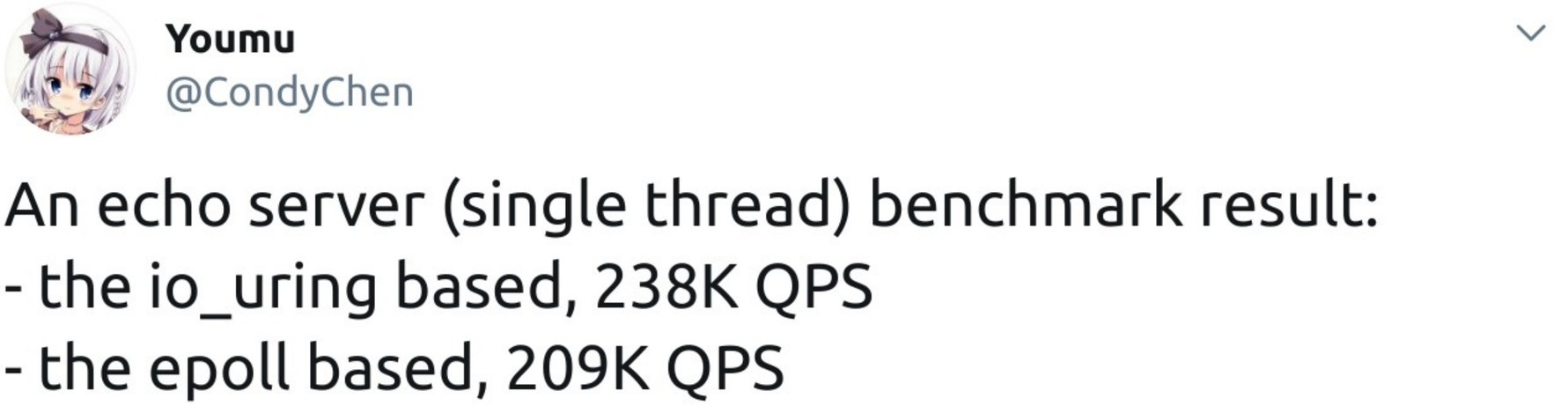
- FB internal bigcache project
  - 1.7M QPS → 2.3M QPS

# Results from the wild

- FB internal bi
  - 1.7M QPS →

# Results from the wild

- FB internal bi
  - 1.7M QPS →




- Building an application for next generation of NVMe SSDs?
  - AIO: 500K IOPS/Core
  - IO_URING: 1 – 2 million IOPS/Core

# Future

- Any system call fully async
- Linked commands with BPF?
- Key/Value store
- Continued efficiency improvements and optimizations
- Continue to improve documentation

# Resources

- **http://kernel.dk/io_uring.pdf**
  - Definitive guide
- **git://git.kernel.dk/fio**
  - io_uring engine (*engines/io_uring.c*)
  - *t/io_uring.c*
- liburing has man pages (for system calls…)
  - Regression tests, example use cases
- **https://lwn.net/Articles/776703/**
  - Not fully current (Jan 15th 2019)