io_uring

Efficient ASync I/O on Modern Linux

Chris McFarlen, Mo Chen

Before io_uring

- Linux AIO has problems
    - Not truly async
    - Bouncing buffered IO to another thread
        - What if it's actually cached?
- syscalls are expensive
    - Spectre/meltdown mitigations have made it worse

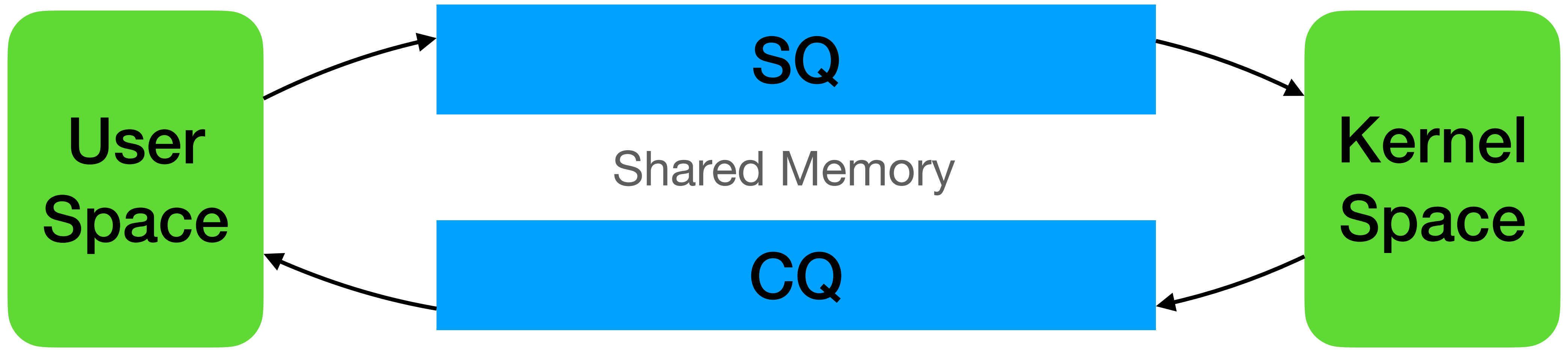
AIO is a horrible ad-hoc design, with the main excuse being "other, less gifted people, made that design, and we are implementing it for compatibility because database people — who seldom have any shred of taste — actually use it".

— **Linus Torvalds (on lwn.net)**

What is io_uring?

- Communication channel
- Lots of operations
  - Block I/O
  - File I/O
  - Network I/O
  - etc.
- Uniform async programming interface

io_uring interface



- Submission queue
- Completion queue
- single producer single consumer queues
  - lock free
  - shared memory between user and kernel
  - Each io_uring instance is typically single threaded

New syscalls

- int io_uring_setup(u32 entries, struct io_uring_params *p)
- int io_uring_enter(unsigned int fd, unsigned int to_submit, unsigned int min_complete, unsigned int flags, sigset_t *sig)

```
struct io_uring_sqe *sqe;
unsigned tail, index;
tail = *sqring->tail;
index = tail & (*sqring->ring_mask);
sqe = &sqring->sqes[index];
/* fill up details about this I/O request */
describe_io(sqe);
/* fill the sqe index into the SQ ring array */
sqring->array[index] = index;
tail++;
atomic_store_release(sqring->tail, tail);
```

Submit work

```
unsigned head;
head = *cqring->head;
if (head != atomic_load_acquire(cqring->tail)) {
    struct io_uring_cqe *cqe;
    unsigned index;
    index = head & (cqring->mask);
    cqe = &cqring->cqes[index];
    /* process completed CQE */
    process_cqe(cqe);
    /* CQE consumption complete */
    head++;
}
atomic_store_release(cqring->head, head);
```

Read completion events

liburing

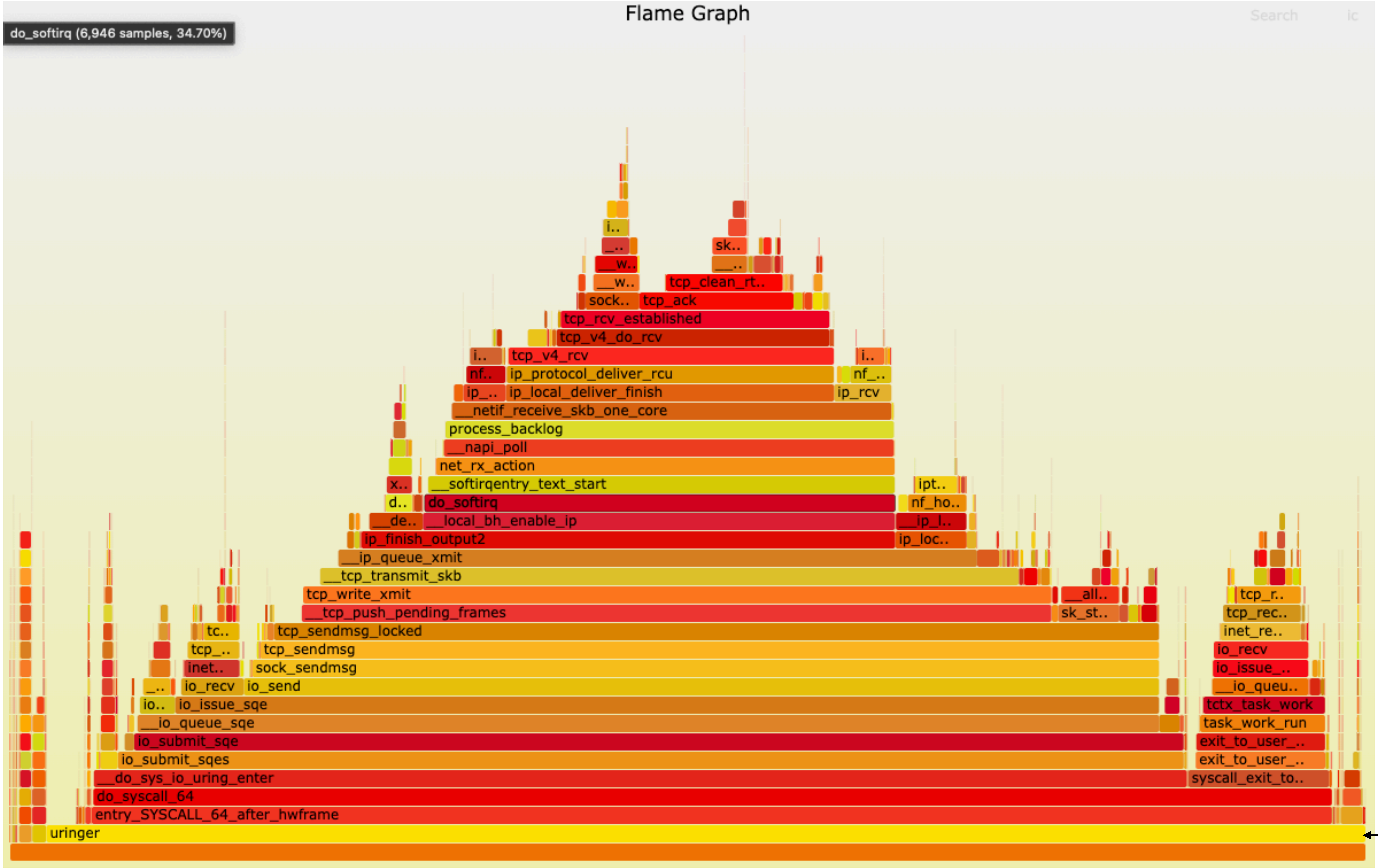
- Userspace library
- Easy to use

```
struct io_uring ring;
ret = io_uring_queue_init(1, &ring, 0);
if (ret) {
        fprintf(stderr, "queue init failed: %d\n", ret);
        return 1;
}

io_uring_sqe* cqe;
io_uring_sqe* sqe = io_uring_get_sqe(&ring);
io_uring_prep_send(sqe, sockfd, data, len, 0);
sqe->user_data = 1;

ret = io_uring_submit(&ring);
if (ret <= 0) {
        fprintf(stderr, "submit failed: %d\n", ret);
        return 1;
}

ret = io_uring_wait_cqe(&ring, &cqe);
if (cqe->res != len) {
        fprintf(stderr, "failed cqe: %d\n", cqe->res);
        return 1;
}
```

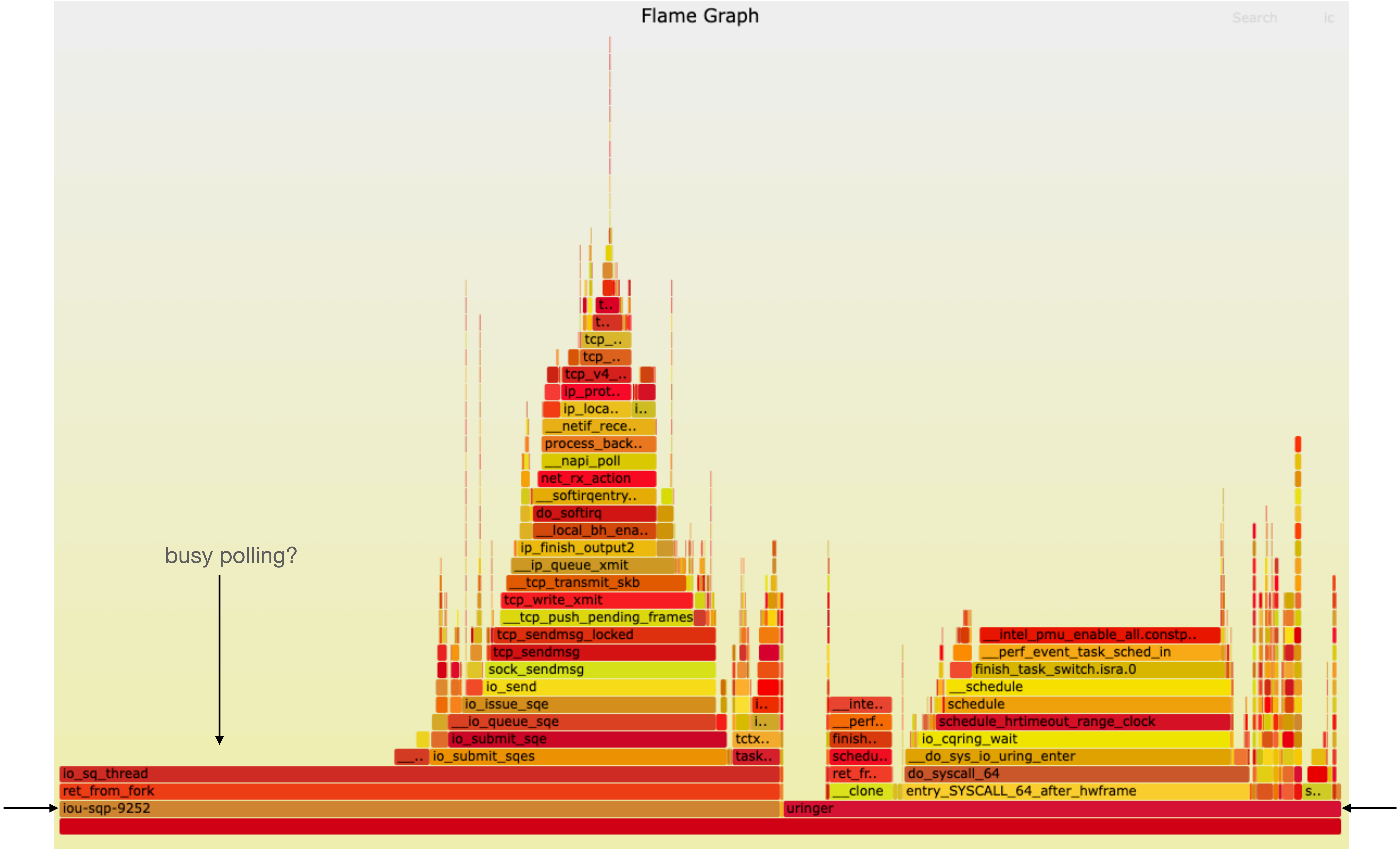
Features

Polling

- IORING_SETUP_SQPOLL
  - Submission queue polling
- busy polling completions
  - Get completed operation results
- IORING_SETUP_IOPOLL
  - Perform busy-waiting for an I/O completion, as opposed to getting notifications via an asynchronous IRQ (Interrupt Request).
  - lower latency busy polling for supported file system or block devices


Flame Graph
Search
ic
do_softirq (6,946 samples, 34.70%)
i..
_..
_w..
_w..
sk..
_..
tcp_clean_rt..
sock..
tcp_ack
tcp_rcv_established
tcp_v4_do_rcv
i..
tcp_v4_rcv
i..
nf..
ip_protocol_deliver_rcu
nf_..
ip_..
ip_local_deliver_finish
ip_rcv
__netif_receive_skb_one_core
process_backlog
__napi_poll
net_rx_action
x..
__softirqentry_text_start
ipt..
d..
do_softirq
nf_ho..
_de..
__local_bh_enable_ip
__ip_l..
ip_finish_output2
ip_loc..
__ip_queue_xmit
__tcp_transmit_skb
tcp_write_xmit
__all..
__tcp_push_pending_frames
sk_st..
tcp_r..
tcp_rec..
tc..
tcp_sendmsg_locked
inet_re..
tcp_..
tcp_sendmsg
io_recv
inet..
sock_sendmsg
io_issue_..
_..
io_recv
io_send
__io_queu..
io..
io_issue_sqe
tctx_task_work
__io_queue_sqe
task_work_run
io_submit_sqe
exit_to_user_..
io_submit_sqes
exit_to_user_..
__do_sys_io_uring_enter
syscall_exit_to..
do_syscall_64
entry_SYSCALL_64_after_hwframe
uringer

main thread



busy polling?

SQ polling thread

main thread

Registered Resources

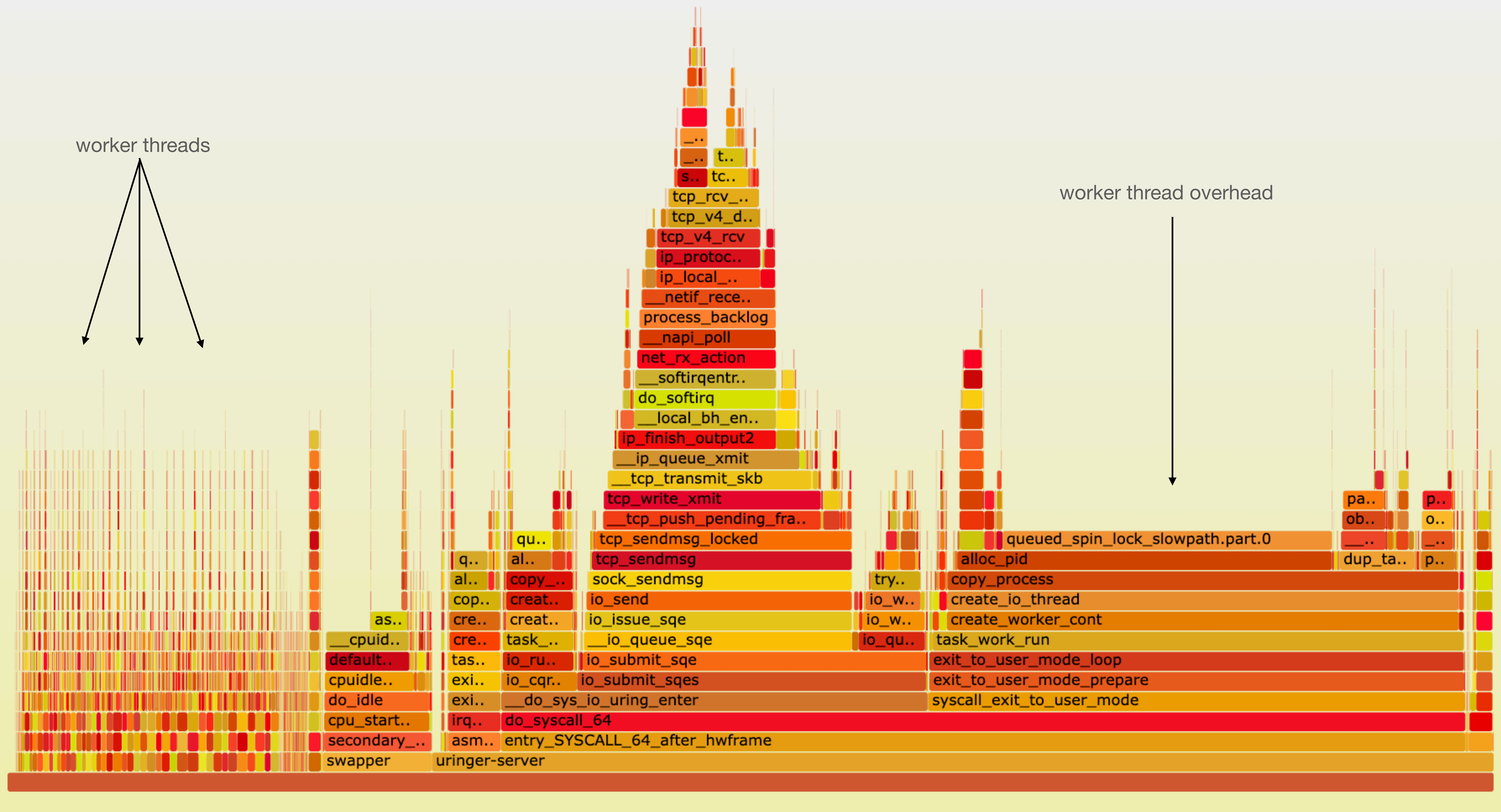
- Often-used resources
  - buffers
  - files
- Lower overhead

Other Features

- Linked operations
- Timeouts
- Cancellation
- eventfd
- CPU affinity

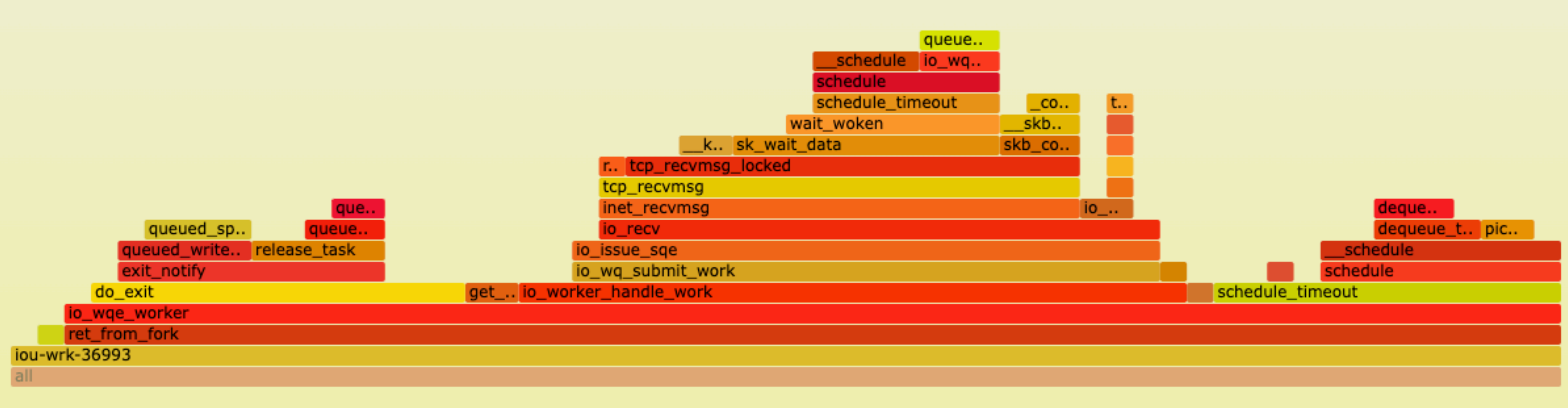
Internals

- How does io_uring actually perform I/O?
    - bounded vs unbounded
    - immediate
    - poll set
    - worker pool



worker threads

worker thread overhead

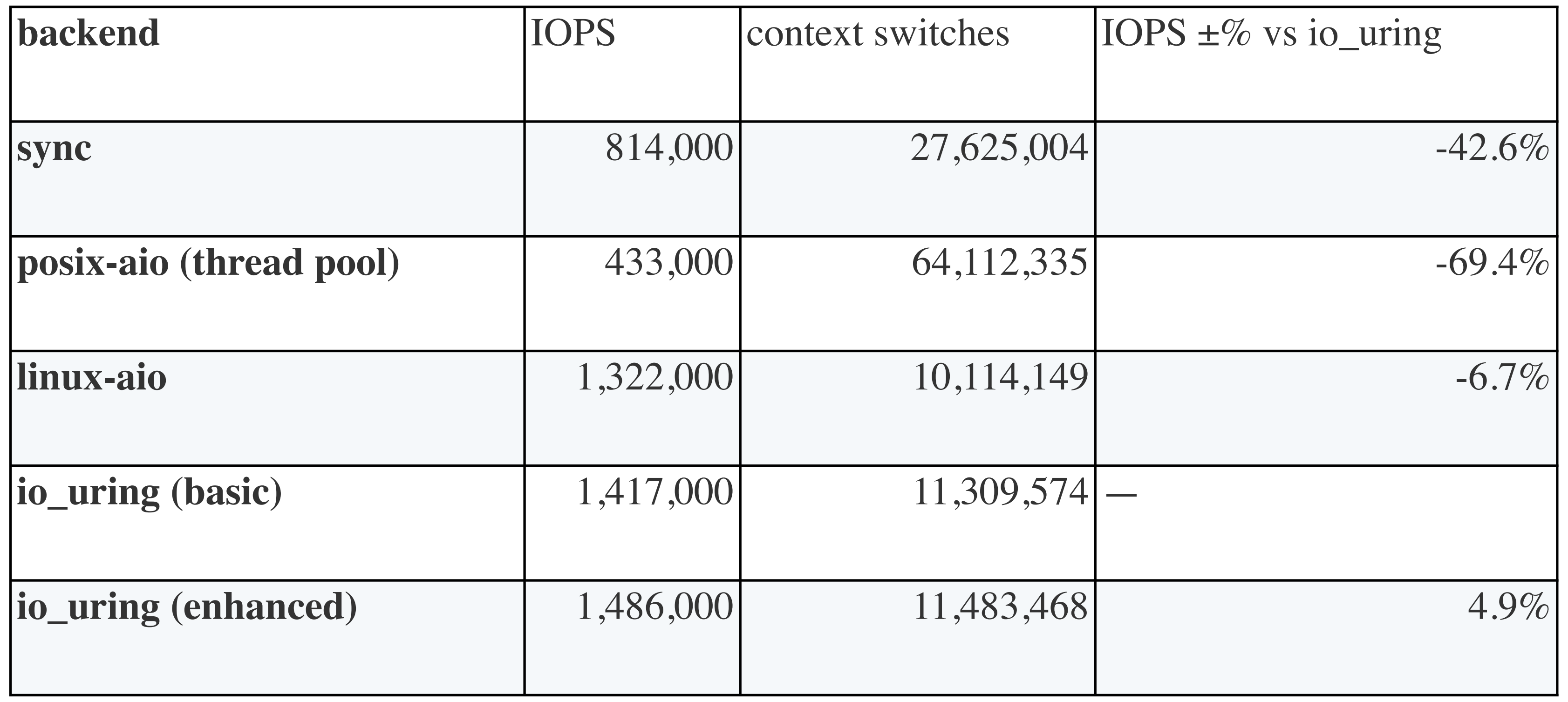

queue..
__schedule
io_wq..
schedule
schedule_timeout
_co..
t..
wait_woken
__skb..
__k..
sk_wait_data
skb_co..
r..
tcp_recvmsg_locked
tcp_recvmsg
que..
inet_recvmsg
io_..
deque..
queued_sp..
queue..
io_recv
dequeue_t..
pic..
queued_write..
release_task
io_issue_sqe
__schedule
exit_notify
io_wq_submit_work
schedule
do_exit
get_..
io_worker_handle_work
schedule_timeout
io_wqe_worker
ret_from_fork
iou-wrk-36993
all

C++ coroutines

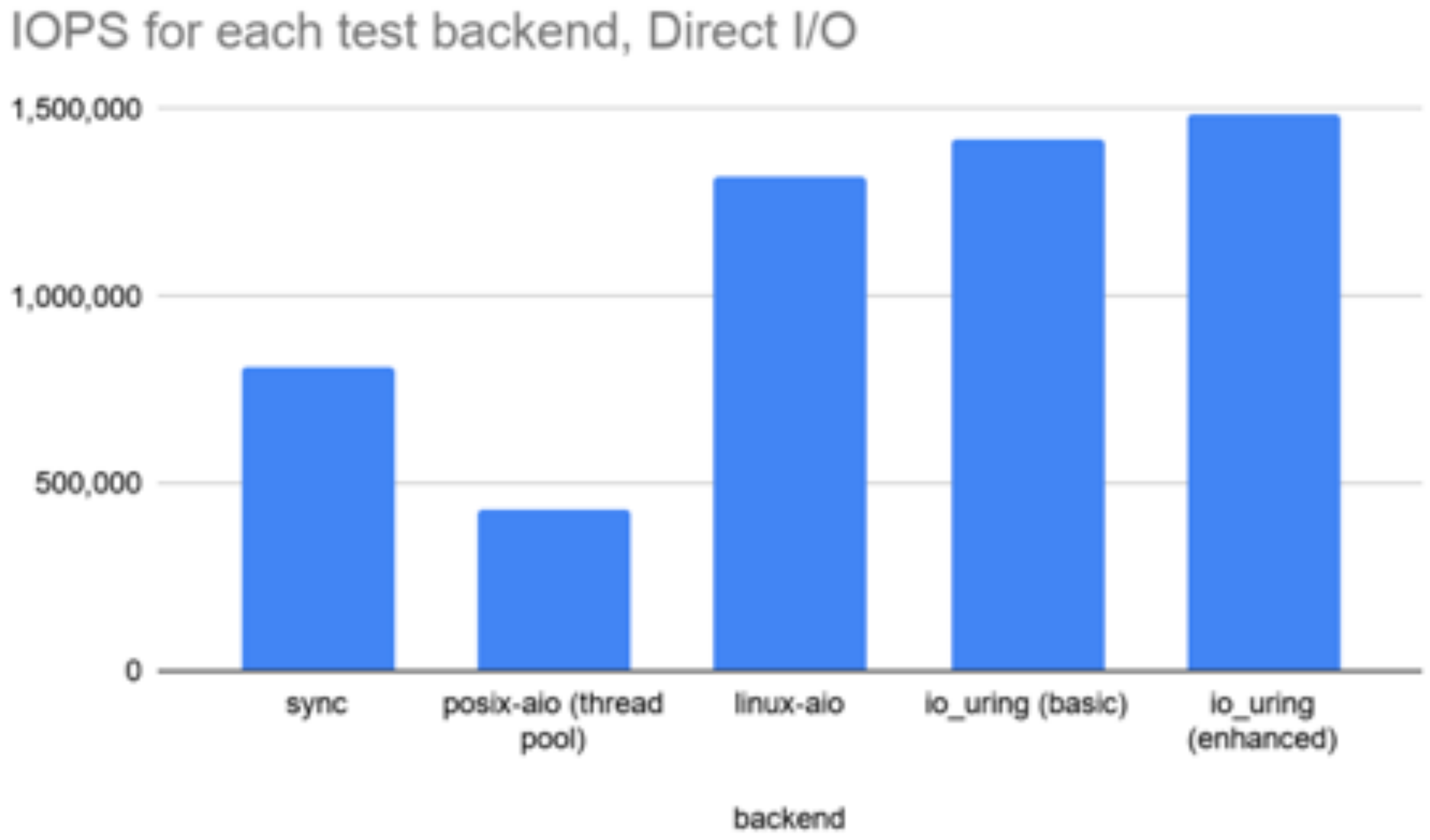
```
auto op = co_await r;

int rc = co_await op.open(path, dfd, O_CREATIO_TRUNCIO_WRONLY);
if (rc > 0) {
    int fd = rc;
    auto wr = co_await r;

    rc = co_await wr.write(fd, content);

     co_await (co_await r).close(fd);
 }
 co_return rc;
```

Performance

Disk I/O performance



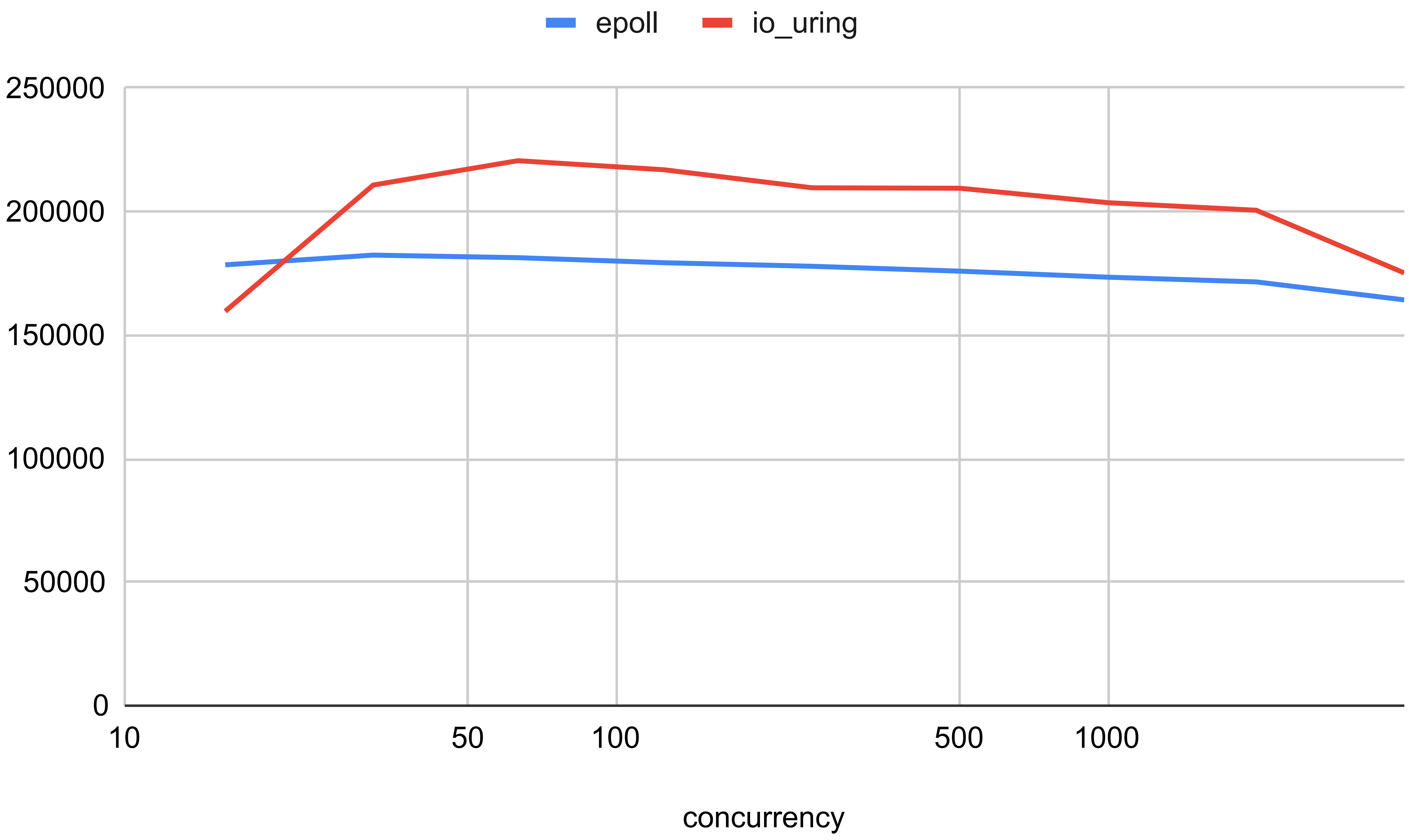
backend	IOPS	context switches	IOPS ±% vs io_uring
sync	814,000	27,625,004	-42.6%
posix-aio (thread pool)	433,000	64,112,335	-69.4%
linux-aio	1,322,000	10,114,149	-6.7%
io_uring (basic)	1,417,000	11,309,574	—
io_uring (enhanced)	1,486,000	11,483,468	4.9%



From: https://thenewstack.io/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux/

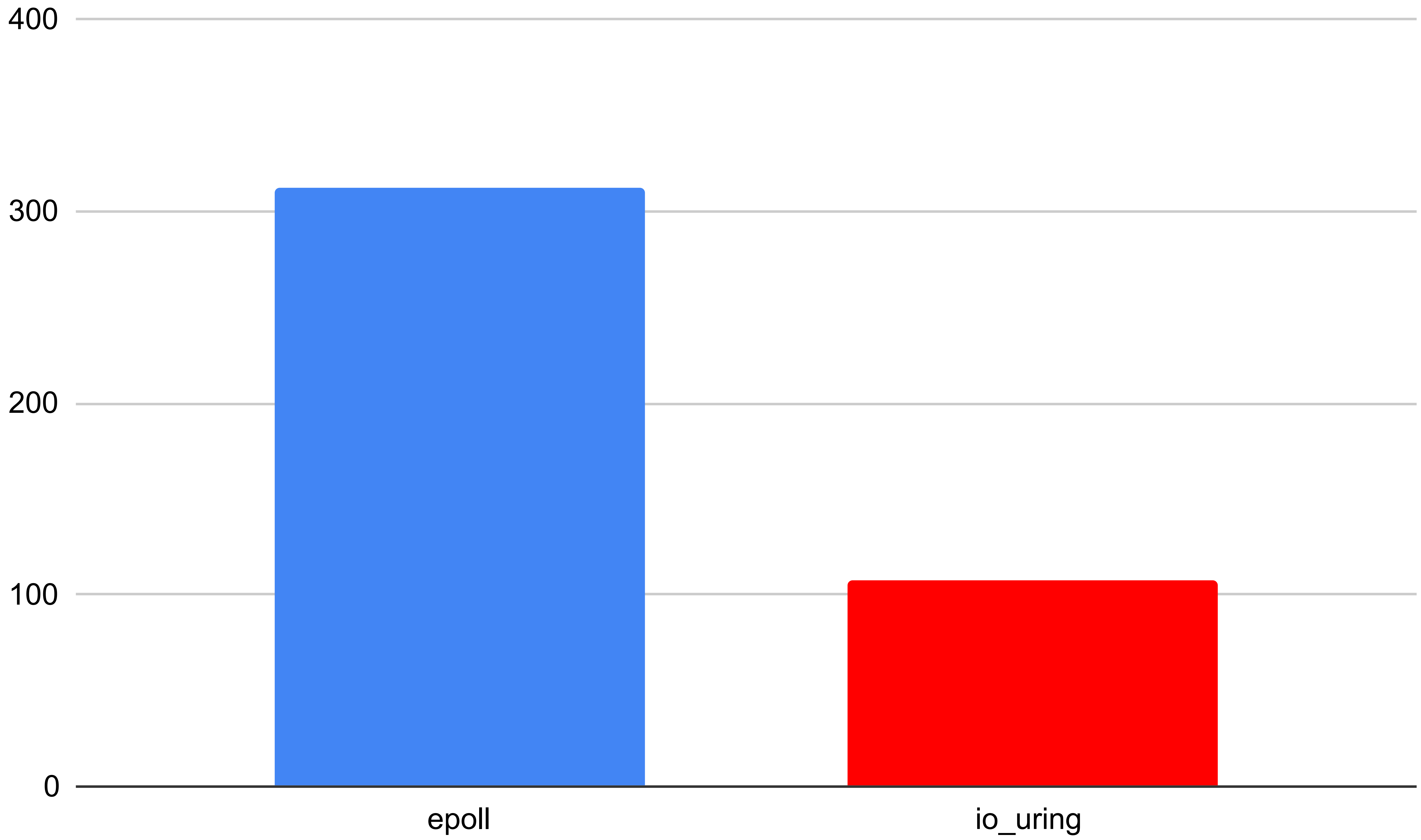
Echo server

throughput



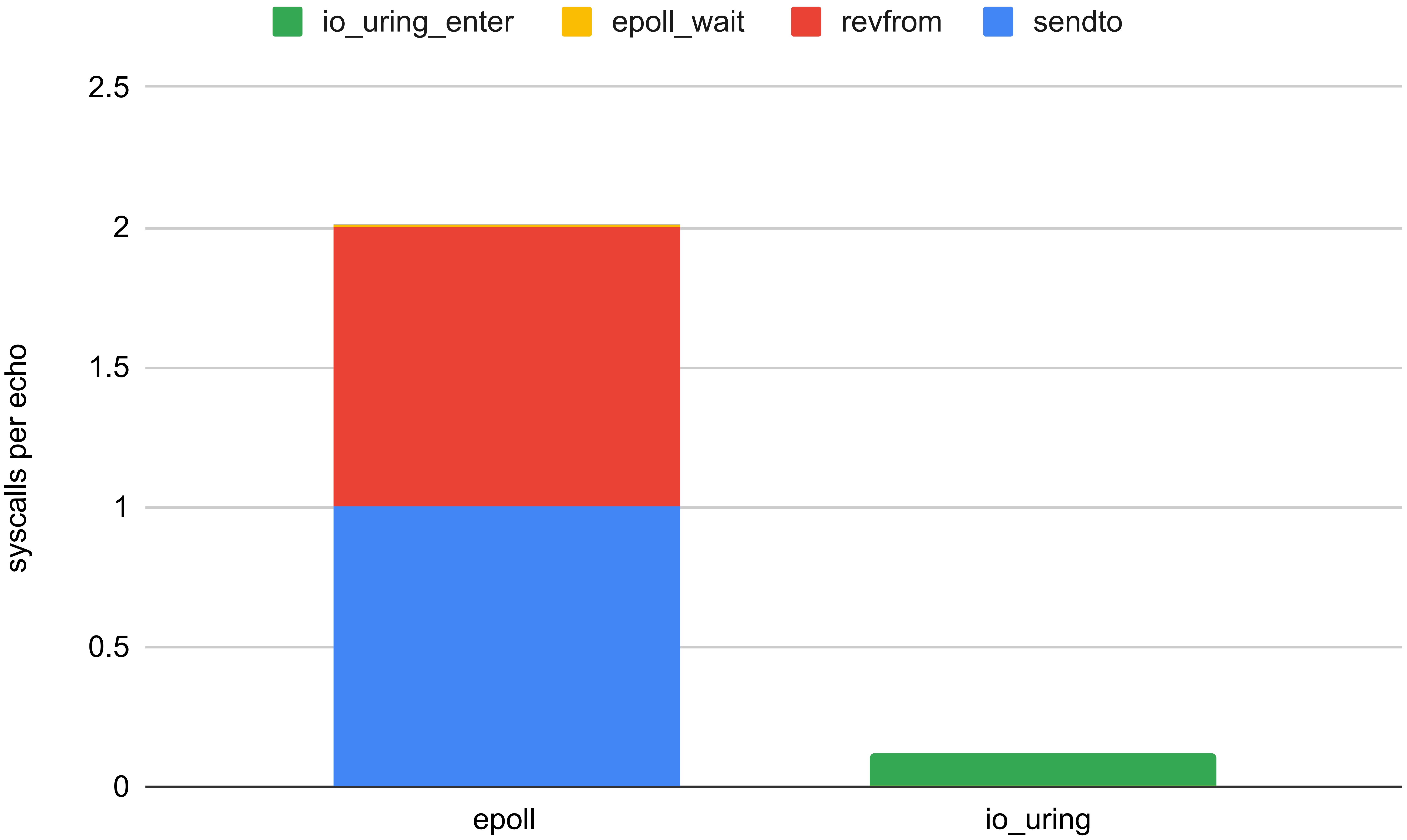
Echo server, c=1000

cycles per echo

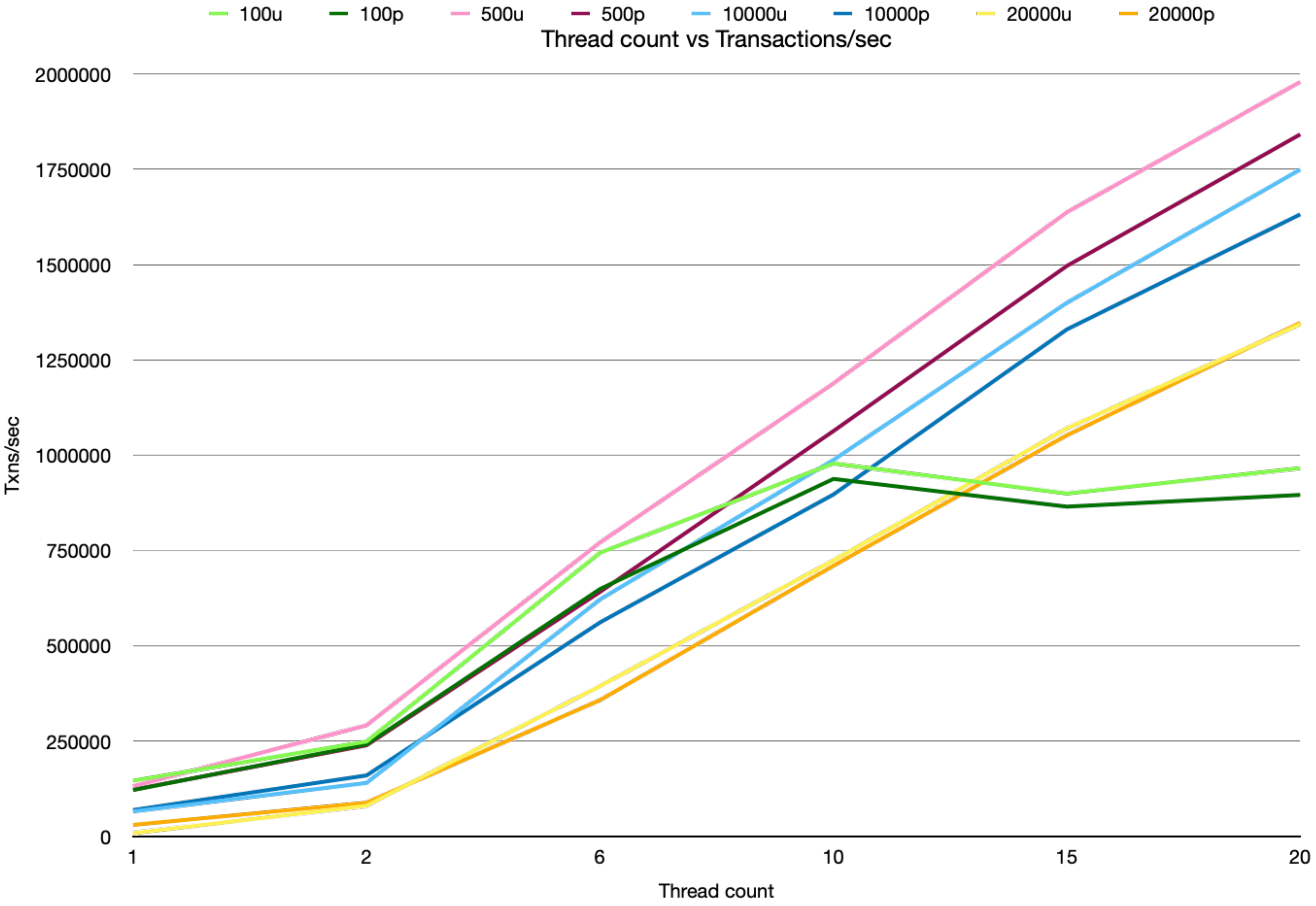


Echo server, c=1000

syscall count per echo



Client performance - io_uring vs poll



Debugging io_uring

- probes
  - kprobe - kernel addresses
  - tracepoint - kernel static tracepoints
- eBPF
  - run sandboxed programs in the kernel
- bpftrace

```
$ sudo bpftrace --btf -e 'kr:create_io_thread { @[retval] = count(); } i:s:1 { print(@); clear(@); } END { clear(@); }' -c '/usr/bin/sleep 3' | cat -s
Attaching 3 probes...
@[-11]: 293631
@[-11]: 306150
@[-11]: 311959
```

io_uring development is ongoing