CS6465: Advance Operating Systems Implementation

Fall 2023

Prof. Ryan Stutsman

# Async IO Benchmark

Muteeb Akram Nawaz, Doctor (u1471482)

Prikshit Tekta (u1319904)

Yuvaraj Chesetti (u1412831)

## 1. Problem

The storage devices are getting extremely fast and have reached sub-10 microseconds latency, and the kernel support for IO needs to catch up and utilize the high bandwidth of these devices. Recent asynchronous IO protocols like *libaio, io_uring, and SPDK* aim to bridge this gap. These protocols scale differently depending on the cores, storage device type, and configurations such as iodepth.

An interesting IO protocol configuration that we wanted to explore is *io_uring* with kernel submission queue polling (SQPOLL). This option has the kernel continuously poll for submission queue requests in the shared memory submission queue, allowing users to submit IO requests without using a system call. This configuration trades the overhead of system calls for an extra CPU cycle for the kernel thread pool to poll IO requests.

The main questions we wanted to answer were

1. How does io_uring + SQPoll configuration compare to other async IO protocols?
2. How does io_uring + SQPoll scale with extra CPU cores?
3. How does the io_uring + SQPoll configuration scale up with the number of assigned kernel submission queue polling workers?

## 2. Key Aspects

Our first goal was to familiarize ourselves with the options Flexible I/O tester (fio) offered. We set up a few tests with fio with different protocols and plotted these results.

The fio setup we used was -

- direct = 1,
- block_size = 1GB
- workloads = read and randread
- runtime = 30sec
- numjobs = 4, 16, 64, 126
- Iodepth = 32, 64, 128

Setting up SPDK was a bit tricky as it had many dependencies, and finding the right machine was a little daunting because of the unavailability of hardware on Cloud Lab. There was one more catch: to run SPDK; we need a dedicated NVME device with no OS running on it. For SPDK to take control of a device, it must first instruct the operating system to relinquish control. This is often referred to as unbinding the kernel driver from the device. SPDK then rebinds the driver to one of two special device drivers bundled with Linux - uio or vfio. In our case, it was binded to the vfio-pci kernel driver.

The io_uring protocol variants we tested out were:

- **iou:** io_uring with interrupts
- **Iou+p:** io_uring with polling
- **iou+k(pin):** io_uring with kernel thread pinned to a single CPU core
- **iou+k:** io_uring with SQPoll with no. of jobs = no. of available CPU cores
- **iou+k(+2):** io_uring with SQPoll with no. of jobs = (no. of available CPU cores - 2)

The configuration we tested out was to test out if,

1. Given the same number of cores, does (io_uring + SQPoll) scale better? [Yes]
2. Does the performance of (io_uring + SQPoll) scale better if given extra CPU cores? [Yes]

The results of testing the above configuration were:

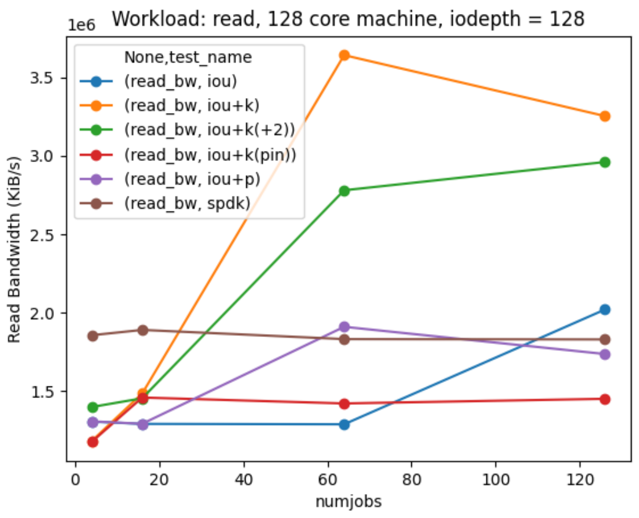

Fig 1. Read Pattern; Scale with cores

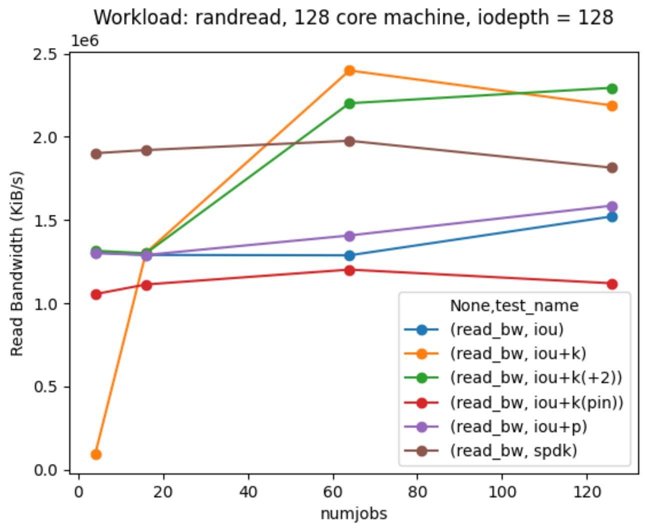

Fig 2. Randread Pattern; Scale with cores

Our next goal was to find the balance between the number of kernel and user threads for the given number of cores. Fio spawns a new kernel thread poller for every job that uses the SQPoll submission. io_uring does provide an option to use kernel workers using the `IORING_ATTACH_WQ` option, but it is not exposed in the fio tool. An ideal setup here would be to extend fio to set IOUring to accept this as an option, but it would require more time and effort. We decided to write our simple benchmark that set up io_uring with this option and benchmark this result. The custom workload fills up the submission queue waits, or polls for entries from the completion queue. Every time an item is retrieved from the queue, a new entry is added to the submission queue.

We fixed the number of jobs and varied the number of kernel worker threads and cores, adding an additional core for each kernel worker. The kernel workers were evenly distributed across all user jobs. The results were:

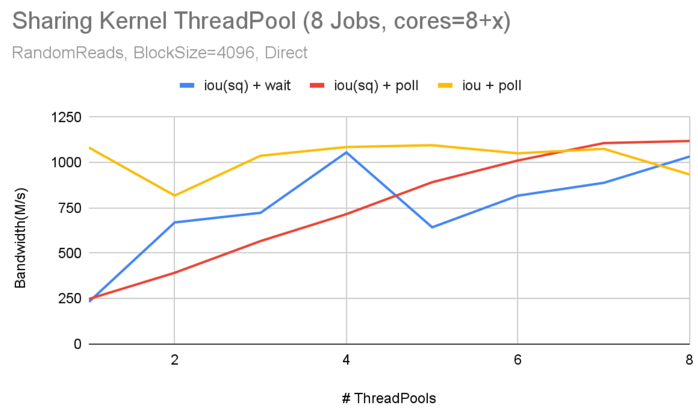

Fig 3. Adding kernel worker. Block Size: 4K

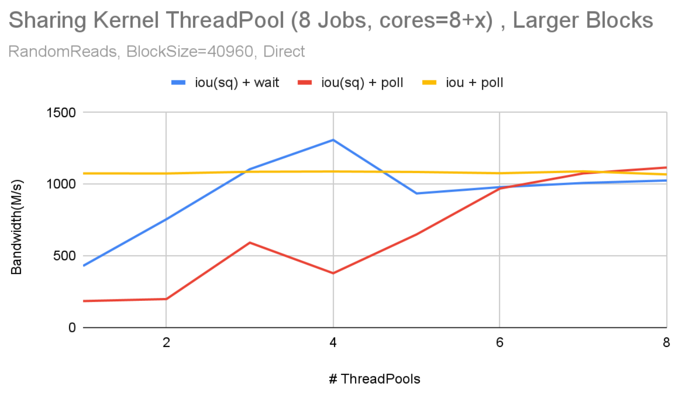

Fig 4. Adding kernel worker. Block Size: 40KB

The blue line is iouring(sq) and makes random reads using `iou_ring_wait_cqe.` The red line uses `iou_ring_poll_cqe,` polling the completion queue. The yellow line is iou with the standard interrupt-based setup. The first graph makes random requests of blocksize = 4KB, while the second graph makes larger requests using 40KB. For this setup and machine, the standard iou setup seems to be the best. The SQPoll setups are bounded by the number of kernel thread workers and can reach parity only when there are enough kernel workers to meet the io demand, at around (j+3, j+4 for both the setups, respectively). Once the kernel workers are able to satisfy the worker demand, additional kernel workers do not add any benefit; we are either bound by the user's job or the io device. We could not run this benchmark on Cloudlab, having lost our reservation for the machine.

## 3. Interesting

1. From our cloudlab benchmarks, at low IO bandwidth utilization, the io_uring SQPoll with multiple kernel workers seems to outperform the SQPoll pinning. This was also seen in our custom benchmark. As the number of jobs/cores increases, the SQPoll setup performs better.

2. The number of kernel threads to assign is dependent on the workload. For example, for highly CPU bound tasks which require bare minimal IO, a single kernel thread could suffice to satisfy multiple user jobs. The CPU-bound tasks can now work without any context switches.

## 4. Limitations

1. Setting up the right benchmark is also tricky. For example, in the initial few runs, we did not realize that pinning all SQPoll threads to the same CPU core was a clearly bad configuration. SQPoll threads tend to utilize all the CPU to full utilization as they are polling, and we were dumping all kernel workers to the same core, creating contention among the kernel workers.
2. Reasoning about benchmarking results is hard. A few things we couldn't figure out are why we get wildly different results for sequential vs random reads. A few surprising results, like (io_uring + sqpoll) beating SPDK.
3. We benchmarked SPDK using SPDK perf as well, and it showed better IOPS performance than reported by Fio. However, the bandwidth results were not completely clear to us. Moreover, the perf documentation did not seem to have any option to specify the no. of jobs either. Analyzing the results more closely using perf or a system call tracer would be helpful.
4. Scale matters; we would have liked to run our benchmark of (io_uring + sqpoll) with kernel worker sharing at higher CPU core machines instead of just the smaller machine. Another benchmark we missed was running with multiple drives and faster storage devices like Optane.
5. We are not sure if our simple custom benchmark stresses the device as much as fio does. Understanding Fio's internals and extending them to enable kernel sharing would make us more confident in our results.

## 5. References

1. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring
2. References to implement benchmark:
   a. Welcome to Lord of the io_uring
   b. How to create a single SQPOLL thread in io_uring for multiple rings (IORING_SETUP_SQPOLL) - Stack Overflow
3. Class Presentation Slides
4. GitHub Repo Link