io_uring and networking in 2023

Introduction

As an IO model, io_uring is applicable to both storage and networking applications. In UNIX, it's often touted that "*everything is a file*". And while that is (mostly) true, once you have to do IO to the file, then all files are not created equal and must in fact be treated differently.

With its completion model based design, converting storage applications to use io_uring is trivial, and can be done in steps:

1) Initial conversion from eg libaio to io_uring
2) Honing the conversion by utilizing applicable advanced features that are exclusive to io_uring.

For networking, the path to idiomatic and efficient io_uring is a bit more involved. Network applications have been written with a readiness type of model for decades, most commonly using `epoll(2)` these days to get notified when a given socket has data available. While these applications can be adapted to io_uring by swapping epoll notifiers with io_uring notifiers, going down that path does not lead to an outcome that fully takes advantage of what io_uring offers. It'll potentially provide a reduction of system calls compared to epoll, but will not be able to take advantage of some of the other features that io_uring offers. To do that, a change to the IO event loop must be done.

This document aims to highlight some of the features that are available and tailored to networked applications, and assumes that the reader is already familiar with io_uring basics. It does not attempt to be a case study in gains achievable by switching from epoll to io_uring.

Batching

One of the key benefits of io_uring is that multiple actions can be completed in a single system call. This is readily apparent in how retrieving and preparing a new SQE for submission (`io_uring_get_sqe()` + `io_uring_prep_foo(sqe)`) is done separately from informing the kernel (`io_uring_submit()`) that new requests are available for processing.

The io_uring system call for submitting new IO, `io_uring_enter(2)`, also supports waiting for completions at the same time. This was an important design decision, as it allows IO that completes synchronously to be done efficiently, compared to async APIs that separate submission and wait-for-completion into two different operations. liburing has support for this through the `io_uring_submit_and_wait()` helper, allowing an application to not only batch submissions, but also combine submissions and completions into a single system call.

This comes in handy in IO event loops, where reaping completions will often yield new operations that need to be submitted. An application can process these completions while simultaneously grabbing new SQEs and preparing them for submission, and finally run another `io_uring_submit_and_wait()` to repeat the event loop.

Relevant man pages: `io_uring_submit_and_wait(3), io_uring_get_sqe(3), io_uring_submit(3)`

Multi-shot

By default, any SQE submitted will yield a single completion event, a CQE. For example, an application submits a read, and once that read completes, a completion event is posted and this completes the operation. However, io_uring also supports multi-shot requests. These types of requests are submitted once, and will post a completion whenever the operation is triggered.

Consider a network application that accepts new connections. The application could retrieve an SQE and use `io_uring_prep_accept()` to be notified when a connection request is received. Once the connection request comes in, a completion is posted with the file descriptor, and now the application has to submit a new accept SQE to handle future connections.

The application could also choose to utilize multi-shot accept by instead using `io_uring_prep_multishot_accept()` to prepare the request. By doing so, that informs io_uring that it should not finish the request fully once a single connection has been accepted. Rather, it should keep it active and post a CQE *whenever* a new connection request comes in. This reduces the housekeeping the application needs to do when handling new connections. Available since 5.19.

By default, multi-shot requests will remain active until:

a) They get canceled explicitly by the application (eg using `io_uring_prep_cancel()` and friends), or
b) The request itself experiences an error.

If further notifications are expected from a multi-shot request, the CQE completion will have **`IORING_CQE_F_MORE`** set in the flags member of the `io_uring_cqe` structure. If this flag isn't set, the application must re-arm this request by submitting a new one. As long as that flag is set on received CQEs for a request, the application can expect more completions from the original SQE submission.

Accept is not the only request type that supports multi-shot. Outside of accepting connections, it's not unreasonable to expect that networked applications will be receiving data from a socket. If multiple requests are expected, then a multi-shot variant of the receive operation is also applicable. Rather than use `io_uring_prep_recv()` to receive data, `io_uring_prep_recv_multishot()` prepares an SQE for multi-shot receive. Just like with accept, a multi-shot receive is submitted once, and completions are posted whenever data arrives on this socket. The astute reader may now be wondering "but where is the received data placed?". For that, io_uring supports a concept called "provided buffers" which is detailed in the next section. Receive multi-shot is available since 6.0.

Outside of receive and accept, io_uring also supports the multi-shot operation for poll requests. The concept is identical - arm a poll request once, and get a notification whenever the specified poll mask becomes true. Available since 5.13.

Relevant man pages: `io_uring_prep_recv_multishot(3)`, `io_uring_prep_multishot_accept(3)`, `io_uring_prep_poll_multishot(3)`

Provided buffers

A readiness based IO model has the distinct advantage of providing an opportune moment to provide a buffer for receiving data - once a readiness notification arrives on a given socket, a suitable buffer can be picked and transfer of data can be started. This isn't true with a completion based model, as a receive operation is submitted ahead of time. Obviously a buffer can be picked by the application when the receive is submitted, but for applications handling hundreds of thousands of requests at the time, this doesn't scale very well and leads to excessive memory consumption.

Instead, io_uring supports a mechanism called provided buffers. The application sets aside a pool of buffers, and informs io_uring of those buffers. This allows the kernel to

pick a suitable buffer when the given receive operation is ready to actually receive data, rather than upfront. The CQE posted will then additionally hold information about which buffer was picked.

io_uring supports two types of provided buffers:

1) The old/legacy type done through `io_uring_prep_provide_buffers()` which has been supported Linux 5.7
2) The newer type, called ring mapped buffers, supported since 5.19.

The two types work identically, the only difference is in how they are provided by the application. The newer type is vastly more efficient (see commit), and should be preferred by applications.

Provided buffers are identified by a buffer group ID, and within that group, a buffer ID. Buffer groups are intended to allow an application to maintain different types of buffers in individual groups. An example of that would be splitting buffers into different groups based on their sizes. Buffer IDs must be unique within a buffer group.

An application must first register a buffer ring for each group ID it wishes to use. This is done via `io_uring_register_buf_ring()`, which sets up a shared ring buffer between the kernel and the application. Once registered, the application may use `io_uring_buf_ring_add()` to add buffers to this group ring, making them available for the kernel to consume. This effectively hands ownership of these buffers to the kernel. In return, once consumed, the kernel will tell the application about which buffer ID from which group ID was used, now handing ownership over to the application again. Once a suitable amount of buffers have been added, the application must call `io_uring_buf_ring_advance()` with the amount of added buffers, making them visible to the kernel.

Once the application has processed the buffer, it may hand ownership back over to the kernel, allowing the cycle to repeat.

Using provided buffers, an application can submit a receive operation without providing a buffer upfront. Instead, it must set **`IOSQE_BUFFER_SELECT`** in the SQE `flags` member, and the buffer group from which the buffer should be picked in the `buf_group` member. No address should be given for these request, instead **`NULL`** should be used. Once the receive operation is ready to receive data, a buffer is picked automatically and the resulting CQE will contain the buffer ID in its output `flags` member. The CQE will have **`IORING_CQE_F_BUFFER`** set in its flags as well, telling the application that a buffer was automatically picked for this operation.

Depending on the rate of arrival of data, it is obviously possible that a given buffer group will run out of buffers before new ones can be supplied. If this happens, a request will fail with **-ENOBUFS** as the error value. One way to deal with that is to have multiple buffer groups for the same type of buffer and round-robin between them. Once an out-of-buffers error is received,the application can replenish that buffer group and swap to using the next one, while rearming the multi-shot (or normal) receive.

Setting up a buffer ring can seem a bit daunting. Since the data is mapped by the kernel, we'll need to ensure that the shared memory is page size aligned. Here's an example of how to setup a buffer ring and fill it with buffers:

```
struct io_uring_buf_ring *setup_buffer_ring(struct io_uring *ring)
{
      struct io_uring_buf_reg reg = { };
      struct io_uring_buf_ring *br;
      int i;

      /* allocate mem for sharing buffer ring */
      if (posix_memalign((void **) &br, 4096,
                  BUFS_IN_GROUP * sizeof(struct io_uring_buf_ring)))
            return NULL;

      /* assign and register buffer ring */
      reg.ring_addr = (unsigned long) br;
      reg.ring_entries = BUFS_IN_GROUP;
      reg.bgid = BUF_BGID;
      if (io_uring_register_buf_ring(ring, &reg, 0))
            return 1;

      /* add initial buffers to the ring */
      io_uring_buf_ring_init(br);
      for (i = 0; i < BUFS_IN_GROUP; i++) {
            /* add each buffer, we'll use i buffer ID */
            io_uring_buf_ring_add(br, bufs[i], BUF_SIZE, i,
                              io_uring_buf_ring_mask(BUFS_IN_GROUP), i);
      }

      /* we've supplied buffers, make them visible to the kernel */
      io_uring_buf_ring_advance(br, BUFS_IN_GROUP);
```

```
	return br;
}
```

This example uses 4096 as the page size, a real application would use `sysconf(_SC_PAGESIZE)` or similar to retrieve it. When retrieving a CQE, the application would do:

```
	io_uring_wait_cqe(ring, &cqe);
	/* IORING_CQE_F_BUFFER is set in cqe->flags, get buffer ID */
	buffer_id = cqe->flags >> IORING_CQE_BUFFER_SHIFT;
	/* find the buffer from our buffer pool */
	buf = bufs[buffer_id];
	[... app work happens here ...]
	/* we're done with the buffer, add it back */
	io_uring_buf_ring_add(br, bufs[buffer_id], BUF_SIZE, buffer_id,
	                      io_uring_buf_ring_mask(BUFS_IN_GROUP), 0);
	/* make it visible */
	io_uring_buf_ring_advance(br, 1);
	/* CQE has been seen */
	io_uring_cqe_seen(ring, cqe);
```

If replenishing a buffer is done in conjunction with incrementing the CQ ring, like in the example above, it's also possible to combine the buffer ring and CQ ring advance into a single operation:

```
	/* mark CQE as seen and update buffer ring index */
	io_uring_buf_ring_cq_advance(ring, br, 1);
```

Relevant man pages: `io_uring_register_buf_ring(3)`, `io_uring_buf_ring_add(3)`, `io_uring_buf_ring_advance(3)`, `io_uring_buf_ring_cq_advance(3)`, `io_uring_enter(2)`

Socket state

By default, io_uring will attempt a receive operation when submitted. If no data is available, it'll rely on an internal poll implementation to get notified when data is available. To aid the application in helping io_uring make more informed decisions on when to receive, io_uring will tell the application if a given socket had more data to be read, and similarly allow the application to control whether to go directly to internal poll

rather than attempt a receive upfront. This can help increase efficiency by not attempting a receive on a socket that is presumed (or most likely) empty.

When a CQE is posted for any of the receive variants that io_uring supports, the `flags` member also contains information on if the socket was empty after the receive had been completed. If **`IORING_CQE_F_SOCK_NONEMPTY`** is set in the cqe `flags`, then the socket had more data available post receive. This may be because the receive asked for less data than was originally available, or perhaps more data arrived after the receive was submitted. Available since 5.19.

Conversely, when submitting a receive operation, if the socket is assumed currently empty, then it is pointless to first attempt a receive and then fall back to internal poll. For this case, the application may set **`IORING_RECVSEND_POLL_FIRST`** in the SQE `ioprio` member to tell io_uring to not bother with attempting the send/receive initially, rather it should go directly to internal poll and wait for notification on when data/space is available.

`IORING_RECVSEND_POLL_FIRST` is available for both send and receive operations. Note that if this flag is set and data/space IS available on submission, the operation will proceed right away. Available since 5.19.

Relevant man pages: `io_uring_prep_send(3)`, `io_uring_prep_sendmsg(3)`, `io_uring_prep_recv(3)`, `io_uring_prep_recvmsg(3)`.

Task work

As mentioned in the previous section, io_uring relies on an internal poll mechanism to trigger (or re-trigger) operations. Once an operation is ready to proceed, `task_work` is used to process it. As the name suggests, `task_work` is run by the task itself, specifically the task that originally submitted the request. By default, task work is run whenever an application transitions between kernel and userspace. Following from this, depending on how the task is notified of the existence of task work, it may include a forced transition between kernel and userspace. Internally this is done with an IPI, or inter-processor interrupt, which will trigger a reschedule of the task. This is similar to how a task ends up processing signals.

Being rudely interrupted by a signaled notification may adversely affect the performance of the application. If that application is busy processing data, then a forced transition in and out of the kernel will slow that down. Luckily, io_uring provides a way to avoid that. When initially creating the ring, the application may set **`IORING_SETUP_COOP_TASKRUN`**.

Doing so will avoid the IPI and forced reschedule of the task when `task_work` is queued, deferring it to when a user/kernel transition happens next time. This is generally whenever a system call is performed. Depending on how events are waited for and processed, the application may also set **`IORING_SETUP_TASKRUN_FLAG`** which will then flag the ring as needing a syscall to process `task_work`. This can be useful if the application relies on `io_uring_peek_cqe()` to know if completions may be available for reaping, and tells liburing that `io_uring_enter(2)` needs to be run to force a transition to the kernel to complete these events. Available since 5.19.

One potential downside of **`IORING_SETUP_COOP_TASKRUN`** is that `task_work` is still run on every transition to/from the kernel. A busy networked application may perform tons of system calls that are not related to io_uring, and each of these will run task work when returning to userspace. This makes it hard for an application to achieve good batching of completions. To better manage this, **`IORING_SETUP_DEFER_TASKRUN`** was introduced. If set at ring creation time, task work is handled explicitly when an application waits for completions. In real life applications, this has shown to yield very nice benefits in terms of efficiency, as it gives the application full control over the batching on the completion side as well. Note that **`IORING_SETUP_DEFER_TASKRUN`** must be used in conjunction with **`IORING_SETUP_SINGLE_ISSUER`**, which tells io_uring that this ring only allows a single application to submit requests. This enables further internal optimizations as well. Not sharing a ring between threads is the recommended way to use rings in general, as it avoids any unnecessary synchronization. Available since 6.1.

Relevant man pages: `io_uring_setup(2)`, `io_uring_queue_init(3)`, `io_uring_queue_init_params(3)`

Ring messages

Ring messages provide a way to send messages between rings. This may be used to simply wake up a task waiting on a ring, or it may be used to pass data between two rings.

The simplest way to use ring messages is to just transfer 8 bytes of data between them. io_uring treats the data as a cookie and doesn't interpret it, hence it can be used to transfer either a pointer or simply an integer value. One use case might be a backend handling new connections and separate threads dealing with said connections, providing a way to pass a connection from one ring to another. `io_uring_prep_msg_ring()` is a way to set up such an SQE. Or it may be used directly

from the thread handling a given connection, to offload expensive work to another thread. Available since 5.18.

Another use case may be passing file descriptors between rings, particularly when using direct descriptors. One use case here may be a backend accepting requests with direct descriptors, and passing an incoming connection to a different thread. Available since 6.0.

Relevant man pages: `io_uring_prep_msg_ring(3)`, `io_uring_prep_msg_ring_cqe_flags(3)`, `io_uring_prep_msg_ring_fd(3)`, `io_uring_register_files(3)`

Conclusion

The above are just some of the optimizations and features that are available for applications to take advantage of in more recent kernels. While this document is mostly centered around networking, some of them are equally applicable to other types of IO done through io_uring as well. It should also not be considered a complete guide, other optimizations and features exist that may improve performance and/or reduce overhead as well. One example of that is direct descriptors, avoid use of a shared file descriptor table between threads. Another would be the zero-copy transmit that io_uring supports. Those topics will be featured in a future installment.

Also important to note that most of the above features came about from adopting io_uring in networked applications, like Thrift. As more adoptions in the networking space are done, there certainly will be more opportunities to innovate in this space, and as a result, this document is also very much a work in progress.

References

io_uring kernel tree: https://git.kernel.dk/cgit/linux-block/ (git clone https://git.kernel.dk/linux.git)
liburing source and man pages: https://git.kernel.dk/cgit/liburing/ (git clone https://git.kernel.dk/liburing.git)

Jens Axboe 2023-02-14