Fast ZC Rx Data Plane using io_uring

Pavel Begunkov, David Wei
Meta
{sil,davidhwei}@meta.com

Abstract

The Linux kernel networking stack has overheads that limit performance. If more performance is needed, the usual way is to use kernel bypass techniques that remove the kernel from the picture. This is infeasible for many applications since kernel bypass requires significant costs both upfront and continuous to redesign entire systems around them.

In this paper, we propose a hybrid solution that is in between using the full kernel networking stack and full kernel bypass. We take advantage of hardware header splitting and flow steering to split the kernel networking stack into a control plane and a fast ZC Rx data plane. This enables our solution to be deployed on systems that expect the presence of a kernel TCP/IP stack, but with some bottlenecked applications using the fast ZC Rx data plane. We present our design and show some positive preliminary results on the reduction of system memory bandwidth requirements at the same network line rates. We then compare our solution with other similar approaches, and finally discuss the potential challenges of making use of our solution in applications.

Introduction

Memory bandwidth can be a bottleneck in large distributed services e.g. disaggregated storage and recently in AI and ML applications. The Linux TCP/IP networking stack in the Rx path first copies packets into kernel memory via DMA, then copies it again into its final destination in userspace memory and/or device memory. At high network line rates, this puts pressure on overall system memory or PCIe bandwidth and adds CPU overheads. These overheads are well known ever since the early days of Linux, and there has been a technological arms race in both hardware and software optimisations to keep up with ever increasing NIC line rate speeds.

One such hardware optimisation is direct cache access (DCA) in processors, implemented in Intel CPUs as Data Direct I/O (DDIO) technology. This optimisation uses the last level cache (LLC) as the destination for network Rx packets by intercepting data from a network interface over PCIe. It works in the background and is entirely transparent to the kernel. While it reduces memory bandwidth in many cases, it cannot be controlled directly and research has shown that it can cause regressions e.g. tail latencies in some edge cases.[4] Other processor vendors e.g. AMD also do not provide an equivalent feature, so it is not something that can be consistently relied upon in environments with mixed hardware types.

Another optimisation technique is for applications to bypass the kernel entirely and directly place packets in their final intended destination. The main two ways of doing this are:

1. Remote DMA (RDMA) e.g. InfiniBand and RoCE where network interfaces do the heavy lifting in processing packets and copying them via DMA to the final destination.
2. Software kernel bypass frameworks e.g. DPDK and PF_RING where packets are copied via DMA into userspace memory and processed using a userspace TCP/IP networking stack.

When implemented correctly, kernel bypass excels at providing low latency and/or high throughput. However, they require the *entire system* to be designed very specifically around them, not only the application in question. Bypassing the kernel means giving up the usual BSD socket API and features that the kernel provides. Other services and libraries running on the same system that assume the presence of a kernel TCP/IP stack will need to be changed or replaced. For companies not in the business of selling specialised hardware or require incredibly high performance (e.g. financial trading firms), the upfront and maintenance costs of re-architecting software that expect a standard BSD socket API is simply infeasible.

In this paper, we introduce a hybrid solution that sits in between the two kernel bypass options. We use header splitting and flow steering hardware features in NICs to essentially split the Linux network stack into two:

- *Control plane* that is handled by the standard kernel TCP/IP stack.
- Fast, zero copy (ZC) Rx *data plane* that directly delivers payloads into userspace memory via DMA.

This allows the networking parts of a system to mostly operate normally, but offer a faster Rx data plane for specific applications. For the userspace facing API we chose to use io_uring which, as we will see discuss in the paper, provides the necessary supporting features and also addresses some of the other overheads with the TCP/IP networking stack.

This is a work in progress not yet merged with upstream, and for now we have posted a working prototype with Broad-

com BCM57504 using the bnxt driver on the mailing list. We are also working on a veth driver for testing the io_uring parts without requiring specific hardware and out-of-tree patches.

Background

Ring Buffers

There are a number of existing kernel bypass solutions, which we will not go into detail as others have already provided a thorough review.[1] The one thing that is common between them is the use of shared ring buffers for passing data between kernel and userspace. This is chosen for a good reason: shared ring buffers when implemented well do not require syscalls and can be lockless. io_uring adds a way of doing I/O in the kernel using shared ring buffers, starting in kernel v5.1.[2]

io_uring

From a high level, io_uring shares two circular ring buffers between the kernel and userspace: one is the Submission Queue (SQ) and the other is the Completion Queue (CQ). User applications submit requests into the SQ and then enter the kernel to handle them. Once each request is done, it posts a completion event into the CQ to notify userspace of the results. One of the key benefits of io_uring is that userspace submitting requests and processing completions do not require separate syscalls, which allows actions to be batched together. Only one syscall is required when entering the kernel after submitting all the requests to do the work.

Normally there is a 1:1 relationship between submitted requests and completions, such that there is a completion for every request. To keep receiving more data, an application must continuously submit receive requests. For network receive requests, this becomes increasingly inefficient at high network line rates, so io_uring offers *multishot requests* that allow a single request to be processed multiple times, posting completions whenever it is triggered.[3] For example, a single multishot receive request on a socket posts completions whenever data is received on the socket. These multishot requests persists until either explicitly cancelled by the application or if an error is encountered.

This type of networking model, called a completion based model, is different to the classic readiness based model that APIs such as epoll provide. In readiness based models, userspace selects buffers for data transfer once it receives notifications that an action e.g. read or write is possible on a socket. But for completion based models where a single multishot request is submitted ahead of time for many completions, io_uring provides a way for userspace to register resources ahead of time as well that the kernel will then use to handle multishot requests. For example, userspace may register memory as a buffer pool that will be used for receiving data in multishot receive request. As data arrives on a socket, io_uring chooses a buffer from the pool to copy into. The completion events posted holds information on which buffer in the pool contains the data.

Page Pool

The page pool allocator in the networking stack has been evolving from a fast path allocator for XDP to a general purpose allocator. At the time of writing, both Broadcom's bnxt and Mellanox's mlx5e drivers use page pool to fill hardware Rx queues with pages. Page pool provides a unified API for getting and putting pages, mapping DMA addresses, and offers optimised page caching and recycling.

Design and Implementation

Figure 1 provides a high level overview of our proposal.

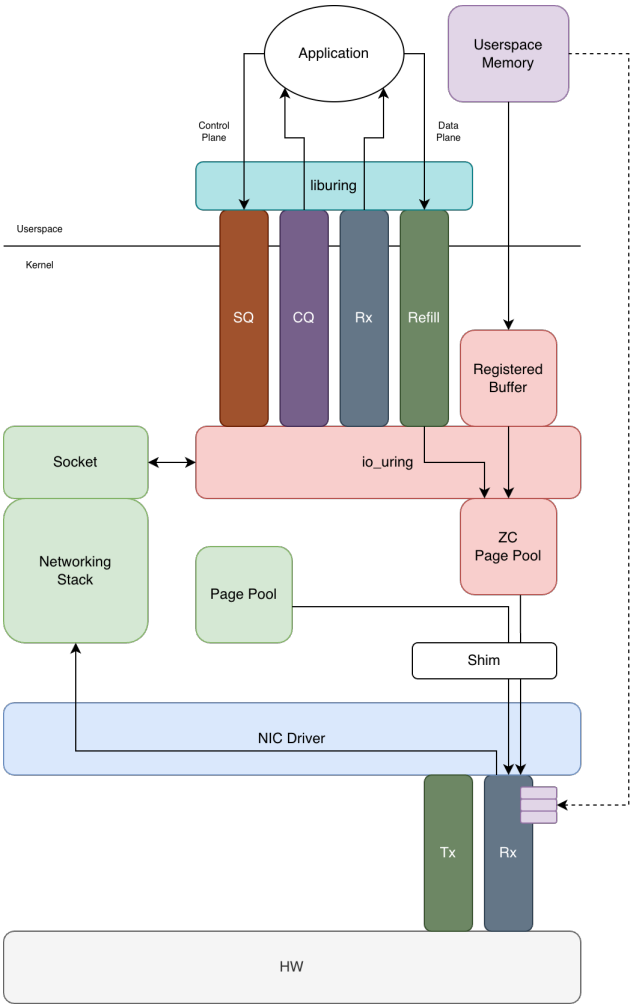

Figure 1: An overview of our proposal of doing ZC Rx using io_uring. We essentially expose userspace memory to hardware Rx queues and use io_uring to manage buffers.

Hardware Side

Buffer Management Currently for networking, io_uring only interacts with the socket API and does not have access to underlying networking resources such as Rx queues. In our proposal, we extend it down into the kernel's networking layer by adding a new ZC page pool allocator in the net-

working stack. This is analogous to the existing page pool allocator, except instead of being backed by kernel memory, it is backed by userspace memory registered with io_uring as registered buffers. A thin shim layer is added that is used by drivers to decide which allocator to use for a given Rx queue.

To configure an Rx queue for ZC, a new `bpf_netdev_command` opcode `XDP_SETUP_ZC_RX` on the NIC driver side and a corresponding io_uring registration opcode `IORING_REGISTER_ZC_RX_IFQ` on the userspace API side are added. This command creates an interface queue object in io_uring that holds all the necessary context, associate a registered buffer region with the ZC page pool allocator, then do driver specific work to prepare an Rx queue by filling it with pages from our ZC page pool. It is required for applications to register buffers first, before setting up ZC Rx. There is a 1:1 relationship between an Rx queue and an interface queue/ZC page pool in io_uring.

Flow Steering In addition to configuring specific Rx queues for ZC, we also only want network flows from specific applications to be directed to them. When these connections intended for ZC Rx are established and a socket is created, userspace must register them with io_uring, associating the socket with an interface queue (and hence, a hardware Rx queue). There is a 1:M relationship between an interface queue and sockets corresponding to connections that want to use ZC Rx. As we previously mentioned, hardware flow steering is used to direct desired flows into specific ZC Rx queues, and receive-side scaling (RSS) is used to make sure other flows do not enter ZC Rx queues. For this paper, we assume that userspace has already set up the correct flow steering and RSS rules using external tooling e.g. ethtool. This is not ideal and we want to tie this to the io_uring registration API. We will discuss this, and other improvements, in the future works section later.

Header Splitting We take advantage of hardware header splitting in certain NICs, where there are typically two different Rx queues per queue index, one for headers and one for payloads. Only the payload Rx queues are filled with pages from the ZC page pool. This ensures that headers are still copied into kernel memory where it cannot be modified by userspace. Note that this is not a hard requirement for ZC Rx to work. The kernel could take an authoritative copy of the headers such that userspace can trash them without affecting the kernel TCP/IP stack.

The hardware side is fully set up at this point. As a NIC receives packets in a ZC Rx queue, they are split such that the header portion is copied into kernel memory as usual, but the payload portion is copied into userspace memory. The NIC notifies the kernel via hardware IRQs, then NAPI poll processes the Rx queues and construct skbs to pass up the networking stack. The only difference is that the skb page frags now point to our userspace pages. These skbs are marked with a special ZC flag to prevent the networking stack from inadvertently freeing the pages.

User Side

Data Plane For the most part, io_uring ZC receive requests are just like standard multishot receive requests, where it reads sk_buffs (skbs) from a socket whenever there is data in the socket. The main difference is that ZC receive requests do not pick a buffer from registered buffers and copy the skb payload into it, as the payload is already copied by the NIC via DMA. Instead, a ZC receive request is mostly a notification mechanism for userspace, to let it know where to look for the payload.

To do this efficiently, two new shared ring buffers between kernel and userspace are added, called *registered buffer rings*: one *refill queue* and one *ZC Rx queue*. These two are analogous to the main io_uring SQ and CQ respectively. As a ZC receive request processes skbs in a socket, it posts entries into the ZC Rx queue, one per skb frag. Each queue entry contains the ZC pool region, an offset into the region, its length, and flags. With this information, an application can find and process the payload data.

The refill queue, as its name suggests, is for an application to return buffers to the ZC page pool that it has finished with. The ZC page pool is also tiered like page pool, with a fast lockless cache, a ptr_ring cache, a refill queue, and finally the main ZC pool region. When refilling hardware Rx queues with pages, the ZC page pool goes through this hierarchy in order. The refill queue is above the main ZC pool region to prevent it from overflowing.

The pair of Rx queue and refill queue pairs form the fast ZC *data plane*.

Control Plane Once all skbs in a socket have been processed, or if the ZC Rx queue is full, a completion event for the ZC receive request is posted in the main io_uring CQ. This serves to tell an application to go look at the ZC Rx queue. This standard io_uring submission request and completion event pairs form the *control plane*.

Resource Management

At any point, a userspace page from a ZC page pool could be in one of several places:

- Hardware Rx queue
- sk_buff in the networking stack
- ZC Rx queue
- Userspace
- Refill queue
- ZC page pool

The ZC page pool refcounts userspace pages to manage their lifetimes as pages are handed out or returned. When pages are ref'd inside of skbs in the networking stack, a custom sk_buff ubuf_info destructor callback is used. This ensures that userspace pages in ZC skbs return back into the ZC page pool when freed.

When tearing down an application, the ZC page pool cannot be freed until all of its pages ref'd inside of skbs have been returned from the Linux networking stack. The ZC page pool schedules delayed work that periodically checks this, similar to how the standard page pool works.

Error Handling

Resource Exhaustion There are several failure conditions that need to be handled gracefully. The main one is, as with any upfront resource registration, if the userspace memory region registered for ZC becomes exhausted. This would happen if the packet arrival rate is much greater than the rate at which an application can process them. When this happens, the hardware Rx queues can no longer be kept filled with userspace pages from the ZC page pool, so we fall back to filling with kernel pages from the standard page pool. Note that this is a permanent change in the state of ZC Rx, even as buffers are returned via the refill queue.

Userspace pages from the ZC pool are distinguished from others by being tagged with a special cookie in the page private field. This cookie is checked when handling skb page frags in ZC receive requests, and if a non-ZC page is detected then it is copied into a page from the ZC page pool instead.

An astute reader may notice that in order to handle copy fallback due to resource exhaustion, we require the very same resource that became exhausted! This is why entering this state where Rx queues configured for ZC are filled using kernel pages again is permanent, even as userspace pages are returned via the refill queue, since those are required for fallback.

This may only provide a temporary respite, as even if kernel memory is inexhaustible, buffer space in sockets is finite. When ZC Rx enters this state, an application is notified in ZC receive completion events with a flag. This gives it an opportunity to address the underlying issue by e.g. allocating and registering a new memory region for ZC. Once the application is satisfied, it can kick the system back into ZC Rx mode, where Rx queues are once again filled using userspace pages from the ZC page pool.

Hardware Failure The other source of errors is if dependent hardware features i.e. header splitting and flow steering fail. Let's look at what happens in detail for each case. Header splitting may fail by splitting too much or too little. If too much, then some payload data will end up in the linear part of skbs, which is gracefully handled by copy fallback. If too little, then we rely on the kernel networking stack to reject the skb.

For flow steering, there are again two cases. A packet from a ZC flow may end up in a different Rx queue and get copied into a kernel page. When this skb is read in a ZC receive request, we again lean on copy fallback to do the right thing.

Or, if a packet from a non-ZC flow ends up in a ZC Rx queue, then it will get accidentally copied into a userspace page. This would not be a problem, as this skb will be read using a standard receive function e.g. `recv()` and be copied into its final destination in a different userspace process. When this skb is done, proper resource management will free its page frags back into a ZC page pool.

Results

We ran a modified version of iperf3 to use our new io_uring API on both an AMD and an Intel system with 62 GB of DRAM and a Broadcom BCM57504 NIC with a 25 Gbps link. The AMD system has an AMD EPYC 7D13 processor, while the Intel system has an Xeon Platinum 8321HC processor.

We ran iperf3 for five minutes and used provided tooling to measure system memory bandwidth: uProf for AMD and pcm-memory for Intel. Figures 2 and 3 show our preliminary results. In both cases, the measured throughput is at line rates. We see that the system memory bandwidth is reduced significantly on both systems.

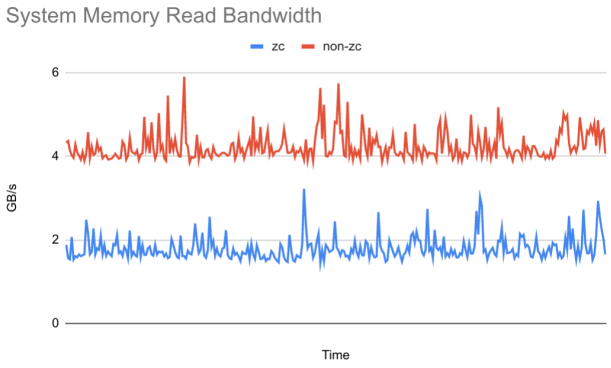

Figure 2: Measured system memory bandwidth on an AMD system.

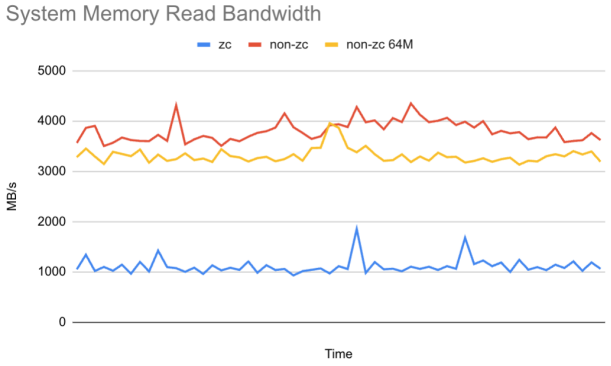

Figure 3: Measured system memory bandwidth on an Intel system.

Discussions

There are a number of existing ways of doing ZC Rx, and when compared to them, our proposal has the following advantages:

- Network packet headers are still processed by the kernel's networking stack. This makes our proposal deployable in situations where this is expected and full kernel bypass is infeasible. All of the infrastructure hooks in the networking stack (e.g. BPF) are available.
- Depending on hardware and driver support, there are fewer limitations on the sizes of payloads and MSS. For example, it is not required for payloads to be exactly 4 kB in size.

- One common trait shared between kernel bypass methods is the use of lockless shared ring buffers, reasons being it is an efficient way of passing data without needing syscalls. We use io_uring for this, which conveniently provides in-kernel shared ring buffers and APIs for interacting with them.
- Networking with io_uring also addresses some of the other overheads of the Linux networking stack e.g. reducing syscall overheads and increasing batching.
- We are working to get our solution merged upstream, and if it does, then there is no need for out-of-tree modules, userspace libraries (beyond liburing for io_uring), and so on.

However, as with many performance optimisations, the trade off is with increased complexity, upkeep and possible modes of failure. Our solution requires deeper plumbing from userspace into the NIC hardware, as the refill/Rx ring buffers we introduced can be viewed as a proxy to hardware Rx queues that is accessible from userspace.

Unlike Tx where the data size is known ahead of time, Rx is both unknown and bursty. This makes it challenging for applications as it must decide upfront the amount of memory and ring buffer sizes to allocate. Making optimal choices here requires an application to understand its workload and how it varies in time. We designed our solution with this in mind by making sure it continues to work even if ZC memory regions become exhausted, and applications can dynamically allocate more resources if needed.

For a complex application, the data hop between a NIC and userspace memory is simply one of many in a long pipeline. Simply eliminating one copy using ZC Rx is unlikely to be sufficient in isolation, if say the data needs to be manipulated after it is received, requiring a copy. We observed that an application needs to carefully coordinate the shape of the data from end to end, eliminating copies along the entire pipeline, while also satisfying constraints such as alignment requirements. As an example, consider a distributed disaggregated storage service, where each storage block has some error correction redundancy added at its tail and writing the blocks directly to hardware has specific memory alignment requirements. A client now cannot simply send the stored data as before, where the server can copy it as needed. For ZC Rx to work well in this case, a client needs to correctly pad the data to allow a server to write the redundancy information without copying, and the NIC on the server must respect the final alignment requirements when placing the data.

One corollary of this is that our solution may not work well with kTLS, which can combine decryption with the kernel to userspace memcpy. Encrypted payloads are copied into userspace directly with ZC Rx, so any decryption (whether kernel or userspace) will require another copy, thus neutralising any benefits.

Future Work

Our primary goal is to get an initial version of our proposal merged upstream. Google is working on a similar ZC Rx proposal targeting device/GPU memory instead of userspace host memory, and in their proposal they are making use of new infrastructure in the Linux networking stack called *page pool memory providers*. This allows custom backends to be plugged into the page pool, and using this would eliminate the need for our custom ZC page pool and reduce driver level changes. We will work to move our solution to this infrastructure.

Once an initial version is merged, we have a long list of future work we would like to do:

- Implement overflow handling for the new refill/Rx ring buffers.
- io_uring already has ZC Tx support. We would like to unify its API with ZC Rx, such that there is a consistent API for applications.
- Performance optimisations, particularly around locking.
- Tighter integration with hardware by being able to configure flow steering rules inline with ZC Rx configuration. Or being able to reconfigure hardware Rx queues dynamically without requiring bringing down the entire interface.
- Add support for device memory as a destination as well, building on top of the work done by Google.

Conclusion

In this paper, we presented a solution for doing network ZC Rx using io_uring on NICs that support hardware header splitting and flow steering. From preliminary benchmarks, we showed that this reduced memory bandwidth requirements at the same network speeds, which has potential to alleviate this bottleneck in some distributed services and AI/ML applications. Its key advantage is that it can be used on systems that expect a kernel TCP/IP stack, unlike kernel bypass techniques.

In some ways, our proposal answers the call in [1], addressing all four overheads in the Linux networking stack: kernel to userspace memcpy, reduce syscalls, adds a fast data plane, and enables applications to manage hardware Rx queues. This is done by extending io_uring down into NIC hardware, which was one shortcoming of io_uring that [1] also noted.

References

[1] Ahern, D., and Mukherjee, S. 2022. Merging the network worlds. In *Netdev 0x16*.

[2] Axboe, J. 2019. *Efficient IO with io_uring*.

[3] Axboe, J. 2022. What's new with io_uring. In *Kernel Recipes 2022*.

[4] Farshin, A.; Roozbeh, A.; Jr., G. Q. M.; and Kostić, D. 2020. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *Proceedings of the 2020 USENIX Annual Technical Conference*.