Evolution of io_uring Zero-Copy Receive

Pavel Begunkov,
asml.silence@gmail.com

Abstract

Zero Copy Receive is a new Linux copy avoidance mechanism for receiving data over the network with `io_uring` based API. It reuses the Linux kernel networking stack and allows users to improve performance while utilising regular Linux tools and infrastructure for monitoring, observability, debugging, configuration, and more.

The interface was first introduced in Linux kernel 6.15 and has continued to evolve since then. In this paper, we explore how the project has changed over time, which new capabilities and user API features it has gained, and what problems we have faced while applying it to real-world applications.

Introduction

The growth in computational demand makes software optimisations as relevant as ever. The challenges facing hyperscale data centres are not limited to provisioning enough hardware; they are also constrained by power consumption and other resources. The ever-growing gap between the performance of CPUs and peripheral devices makes the issue even worse. Network interface card (NIC) speeds are constantly improving. Today, 200 GbE adapters are already common in high-performance environments, while newer devices are able to sustain transfer rates over 1 Tb/s. Processing such an amount of data consumes a significant amount of system memory bandwidth and puts substantial strain on CPUs and caches.

One source of this overhead is the copying of data to userspace with traditional networking APIs. The Linux kernel provides multiple solutions for avoiding copies in the receive path with different advantages and disadvantages. These include kernel-bypass solutions such as `DPDK`, and `mmap(2)` techniques such as `TCP_ZEROCOPY_RECEIVE` [1]. `io_uring` Zero Copy Receive implements an approach based on dedicated NIC receive queues and steering rules. It was first introduced in Linux 6.15 and proved to be a high-performance networking solution that reuses the kernel networking stack, allowing users to leverage conventional tooling for monitoring, observability, configuration, and more. However, to be truly successful in real deployments, raw performance alone is insufficient. The interface must also be reliable, scalable, and maintainable. This paper explores how the interface has evolved to better meet the needs of real-world applications.

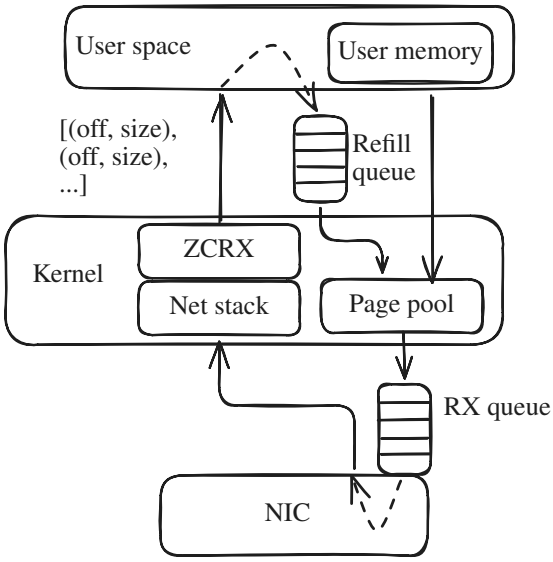

Figure 1: Performance without and with the contention optimisations.

The Approach

The idea behind the approach is to give the user the ability to allocate one or more dedicated NIC receive queues and populate them with userspace-supplied memory called an area. When receiving traffic from such a queue, the network card transfers payloads using DMA directly into the user-accessible memory even before any kernel driver or protocol processing takes place [3]. This method requires hardware steering rules that place traffic intended for the zero-copy application and only for that application into the allocated queues. There are several reasons why we need steering. First, the data is directly accessible to the application only when it goes through a zero-copy queue and otherwise requires a copy. Second, we want to prevent data for other applications being placed into the provided memory as it becomes immediately visible to the zero-copy application, which violates isolation. Third, the supplied pool of memory is finite, and even though it is reused, it is still a limited resource, and we want to avoid interference from other programs.

Memory registration is expensive and cannot happen in the data path for performance reasons. The kernel and hardware are responsible for allocating from the supplied pool of memory and for letting the application know where in the memory its data is located. Application processing may take time, and the kernel must not be able to reuse the buffers and overwrite user data in the meantime. For that reason, the user must explicitly acknowledge the buffers once it has finished using them.

io_uring

io_uring is a Linux kernel API for asynchronous I/O. The communication with userspace happens over a pair of rings: the Submission Queue (SQ) and the Completion Queue (CQ). Requests are submitted by filling a Submission Queue Entry (SQE) and issuing an io_uring-specific system call. The result for a request is posted into the Completion Queue. Traditionally, there was one Completion Queue Entry (CQE) per request. However, requests using multishot semantics can produce multiple CQEs without needing to be resubmitted.

Zero Copy Receive

There are several zcrx design decisions that we need to mention. First, steering rule configuration is left out of the interface and is expected to be handled by other already available APIs and tools. While keeping it as part of the project could have been beneficial to API coherence and capability handling, it would have also made it more complicated, taking into account the fact that the number of rules we can allow is limited by hardware, and the application may need to use elaborate masking to aggregate all its traffic into the minimal number of steering rules.

Another core design decision of zcrx is using a kernel-user shared ring called Refill Queue (RQ) for buffer acknowledgments which tell the kernel when buffers previously given to userspace can be reused. For each data completion the user is expected to eventually post one RQ entry (RQE). There are no system calls required for the handoff, and the queue is polled asynchronously by the NIC driver when it needs more buffers.

The interface is built as an io_uring API, and to start using it, we first need to create a zcrx instance within io_uring by specifying the NIC, the receive queue index, the memory address of the area and other parameters, and making a registration system call. From then, assuming that the steering rules are properly configured, it can be used with any TCP socket using a new io_uring request type. In many ways it behaves like any other receive request, but the main difference is that instead of copying data into a given buffer, a CQE returns an offset and size into the zcrx area telling where the next chunk of data was DMAed. zcrx requests also support the multishot mode, in which case multiple ordered CQEs are returned. Figure 1 The figure presents a high-level, simplified representation of a typical data and buffer flow.

Design Evolution

The initial version of zcrx consisted of the minimal API and features for simplicity. Some changes were planned and anticipated; other problems were discovered while using it. While performance and efficiency remain a major factor for choosing the underlying infrastructure, they are just one concern that real applications have. Any interface should also be robust, reliable and predictable. In this section, we walk through the issues the first iteration of zcrx had and the solutions to them.

Refill Queue

The return of buffers via the refill queue proved to be an effective solution to the problem [2]. Not only is it highly performant, but it also gives a simple and convenient interface to userspace. It provides good natural batching without more complex userspace solutions for amortising the cost.

However, consumption happens asynchronously, and when consumption can't keep up with production the queue can become full. In this case, the user can't return buffers and there was no fallback mechanism to address it. The only way for the userspace to handle it is to retry later when the queue becomes empty. This approach is clunky, unpredictable and relatively high-effort.

In Linux 6.19 [8], we introduced a system call based API for synchronously emptying the queue by userspace. To use it, the user should make the io_uring registration system call with opcode IORING_REGISTER_ZCRX_CTRL, and set the control method to ZCRX_CTRL_FLUSH_RQ. On return from the system call, the queue should be empty unless an error occurs. We're reusing the same queue instead of passing a separate array as it simplifies the user space handling. However, we might need to add another option to facilitate multithreaded applications and zcrx sharing.

ZCRX_CTRL_FLUSH_RQ helps handle bursts of traffic and latency spikes, but it carries a higher cost than the original method. It is added to augment and not replace the normal refill queue flow. The user still needs to pay close attention to sizing the refill queue well, especially if it needs to use RQ flushing often.

Memory Pool Exhaustion

For traditional Linux kernel networking, buffers for incoming traffic are allocated from normal kernel reserves via the page allocator. It gives it access to a potentially large amount of memory and fallback mechanisms like memory reclaim and the OOM killer. zcrx has to use a different model, where the user allocates and provides the operational memory in advance at the registration time. It can slightly improve predictability in low system memory situations, but it limits memory available for receiving, and on average it makes allocation failures more likely to occur. The fact that the buffer lifetime is extended to userspace processing makes the situation even worse. Not only does the user need to consider the NIC bandwidth and syscall handling rate while sizing the area, but now it has to estimate and account for application processing times. It becomes even harder if that includes other I/O or involves other libraries and frameworks.

There are two problems with memory exhaustion for zcrx. First, the allocation failures occur asynchronously from the socket processing and CQE delivery path, and, as there is no simple way to propagate the error, this happens

Table 1: CPU utilisation with different rx page sizes.

Page	%usr	%sys	%iowait	%irq	%soft	%idle
4 KB	1.53	27.78	2.72	1.31	66.45	0.22
32 KB	0.69	8.26	31.65	1.83	57.00	0.57

silently for the user. Recovery from such failures is difficult, and the packet drop can have a cascading effect.

In Linux 7.2 [7], we're introducing asynchronous `zcrx` event notifications. The user needs to specify the types of events it wants to receive and a unique tag while registering `zcrx`, and if the event triggers, it will get a CQE with the specified tag and the event type. Currently, there are two types of events available, `ZCRX_EVENT_ALLOC_FAIL` for allocation failures and `ZCRX_EVENT_COPY`, which notifies when kernel-allocated buffers were used and usually indicates problems with steering configuration. To prevent spamming with identical CQEs from a recurring condition before the user can react, `zcrx` stops posting events of a specific type until the user explicitly confirms that it has received and handled the previous notification with a new `zcrx` control opcode called `ZCRX_CTRL_ARM_EVENT`. Another reason to mediate it via a system call is that the kernel may potentially need to take extra steps to complete a recovery. For example, we may need to restart the NIC queue processing in case it is completely halted and can't pull buffers out of the RQ.

For now, the memory exhaustion event is delivered when `zcrx` memory pools are completely exhausted. It doesn't mean that the NIC will immediately fail as there might still be entries in the hardware receive queue, but it's close to that. In the future, we might need to add options for better anticipating a failure, for example, by notifying when only a specified percentage of memory remains available.

The other part of the problem is handling allocation failures. To deal with the failure, some applications can flush caches, speed up buffer processing or cancel requests, however, it is not always an option, and we need a fallback mechanism. For Linux 7.3, we plan to add a way to dynamically extend the memory pool by providing areas at runtime via a new `zcrx` control opcode called `ZCRX_CTRL_ADD_AREA`.

Receive queue sharing

Another problem we faced is network cards supporting only a limited number of receive queues. For applications at scale, it's not enough to use just one CPU for network processing, and they need to utilise all available resources. The most natural design is to derive the number of threads from the number of available CPU cores and create a separate `zcrx` instance for each of them. However, modern servers can have hundreds of CPU cores, and it would require as many receive queues, which might not be available. On top of that, in many cases we also need to leave a high number of queues for normal network processing.

To mitigate that, in Linux 6.19 we introduced `zcrx` instance sharing between multiple `io_uring` contexts. First, the user needs to export a `zcrx` as a file via `ZCRX_CTRL_EXPORT`, and then it can be imported into another `io_uring` by specifying the `ZCRX_REG_IMPORT` flag during registration. There is just one refill queue and the user is responsible for synchronising accesses to it in multithreaded environments. The most conventional method would be to use mutexes, which work well in practice for sharing between a low number of threads. Otherwise, the user may want to implement multi-producer lockless queue semantics. Possibly, we might extend `zcrx` in the future with support for multiple refill queues.

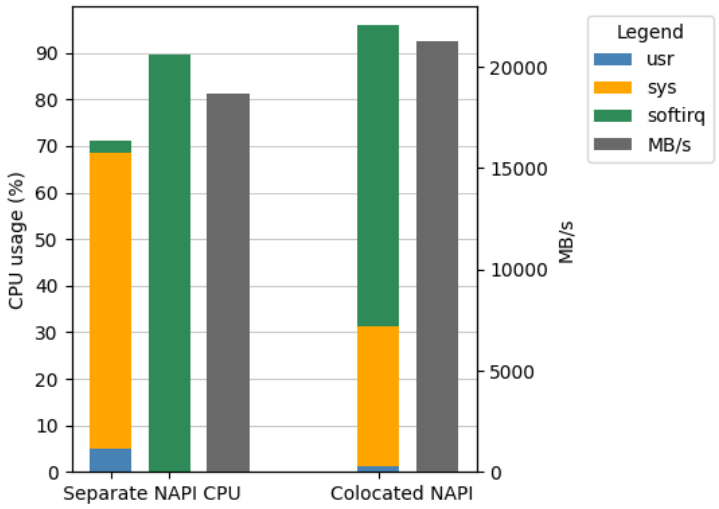

Figure 2: Experimental results for userspace and softirq sharing vs using different CPUs.

Feature Probing

Many applications can't rely on having a recent enough version of the Linux kernel, which means some of the features might not be available. A robust application needs to probe support for capabilities it wants to use. The common way of doing that is to try using the feature and inspecting the result for errors. It is often cumbersome and complex, and to help userspace we introduced support for querying available `zcrx` features via `IO_URING_QUERY_ZCRX` sub-operation of io_uring's `IORING_REGISTER_QUERY`. It returns a number of attributes, such as a mask of available registration flags, the number of available control opcodes and more. It also returns a description of the refill queue, specifically, it tells how much space the queue header takes. Previously, the user had to overestimate it in case of user allocated refill queues.

Device Memory

One of the most frequently requested features was support for device memory for direct peer-to-peer transfers from a NIC to other PCIe devices. From Linux 6.16 [6], `zcrx` optionally supports areas backed by dma-buf. Dma-buf is a framework for sharing buffers that can be backed by device memory and hides most of the details about setting up DMA mappings and more. A dma-buf is exported as a file descriptor, which can be passed during `zcrx` registration together with a new flag

Table 2: Colocated vs cross-CPU configurations.

Configuration	%usr	%sys	%irq	%soft	%idle	GB/s
application + driver	1.23	30.33	1.02	64.86	2.56	20.7
application	4.57	59.89	1.63	2.50	31.41	18.2
driver	0.00	0.00	0.40	81.46	18.14	—

called `IORING_ZCRX_AREA_DMABUF`.

Large Receive Page Support

As explained in the API description section, the result of executing a `zcrx` request is multiple ranges pointing into the memory area. The actual size of these segments varies and depends on a number of factors, but it's often limited by how the data arrives over the wire and how it is split into packets. With a higher MTU size and packed streams of data, it is reasonable to expect a lower number of larger segments. Segment processing could be expensive for both the Linux kernel and userspace. The more segments the stack needs to handle, the more expensive it is.

Another limiting factor is the size of buffers we're providing to the NIC, and by default `zcrx` allocates in system page size chunks. However, modern network interfaces can support larger sizes. When coupled with a large MTU and/or hardware-GRO, it can produce much larger segments, drastically reducing processing cost for the kernel and userspace.

In Linux 7.0 [**?**], we introduced support for larger `zcrx` page sizes. It doesn't require any changes from applications in the data path, and they only need to specify the desired size during registration. It requires that the area memory consists of physically contiguous chunks of the matching size, which implies the use of huge pages. It also needs support from the driver and hardware. Because of that, the registration can fail, and the user is generally expected to fall back to the system page size by setting the parameter to zero.

We conducted our experiments on x86-64 servers located in close proximity and equipped with 200 Gbit/s Broadcom NICs. The MTU was set to 9000 bytes with hardware GRO enabled. The benchmark application uses a single TCP stream at maximum capacity, and we tested 4 KB and 32 KB receive page sizes. As shown in Table 1, we see a 40% reduction in CPU utilisation while sustaining the maximum transfer rate supported by the cards. It was also independently tested with Mellanox network cards and yielded a 12% increase in bandwidth while improving CPU utilisation by 20% [4].

While hardware GRO yields impressive results in benchmarks like the one above, it's not a silver bullet. Perfect packing is not guaranteed, and the resulting segments could be smaller than the receive page size. Efficacy decreases as the number of active connections and overall networking noise grow. Real applications should expect more moderate improvements from using larger page sizes.

Caches and Contention

There are two main components of the data handling path. The first is socket processing that returns CQEs back to the user. It is run in the application process context as a part of the `io_uring_enter` system call. The second is buffer allocations by the driver for the network card, which is typically executed asynchronously with respect to the application and on a different CPU, unless the user takes extra steps to set up CPU affinities.

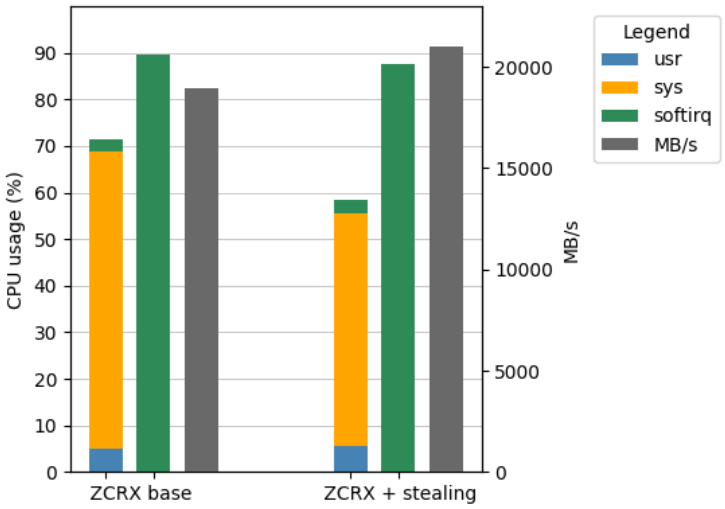

Figure 3: Performance without and with the contention optimisations.

While experimenting with the interface, we saw a much higher CPU efficiency when the system call execution and softirq processing occur on the same CPU. While the multi-CPU setup is using more cycles in total, there were drops in throughput, and the throughput per CPU was up to twofold lower. See Table 2 and Figure 2.

Reference Counting Investigations showed high cache contention concentrated around buffer reference counting. `zcrx` manages the area in chunks of a predetermined size, which is normally the system page size unless large receive pages are requested. There are two different references for each chunk. The first one guarantees that a chunk is not freed or reused while traversing the networking stack. It lives in `net_iov` and is analogous to page pool page referencing via `pp_ref_count`. The second tracks whether a chunk is given to userspace for processing and is expected to be returned via the RQ. Its main purpose is to prevent the user from returning a chunk it doesn't currently own and from other user misuses and mischiefs.

Following the buffer lifecycle, both cache lines for buffer references start in the exclusive state for the driver CPU, bounce to the application CPU after being read and modified, and later come back to the driver CPU. In the best-case scenario, that's four exclusive-to-exclusive cache line ownership transfers for each segment, and, for example, with a 200 Gbit/s NIC, it is tens of millions of transitions per second.

To solve the problem, we are combining two approaches. First, we steal `net_iov` references from `sk_buff`, which eliminates one pair of atomic modifications. More impor-

tantly, it results in the net_iov cache line being only written by the driver CPU. It is worth noting that the cache line is still read by the application CPU, which causes exclusive to shared cache transitions.

The second optimisation is addressing the "user" reference contention. Instead of performing this accounting immediately in the syscall receive path, we are going to pack multiple segments into a container and send them to the driver CPU so that it can do it for us. Fortunately, they already come in a container called sk_buff, and we only need to steal it from the networking stack.

We see an over-20% increase in CPU-normalised throughput for test cases where the driver and userspace are run on different CPUs, and overall improvement in throughput, see Figure 3. There is also a smaller improvement in CPU utilisation in single CPU tests. The multi-CPU setup still doesn't achieve the same level of efficiency as we observe with a single CPU, and further research can be conducted to improve it.

Refill Queue Another direction for improvement was reducing false sharing of the refill queue headers. First, we moved head and tail pointers to different cache lines to avoid exclusive-to-exclusive cache line transfers. Second, as the kernel doesn't always consume all RQ entries in a single run, we cache and reuse the value of the tail. There are many papers describing single-producer single-consumer queues, and most optimisations can be applied to the refill queue [5].

Conclusion

Zero copy receive is a new interface within the kernel ecosystem and it keeps evolving to better suit current users and to cover more use cases. There are many plans for future extension and improvement and much work remains.

References

[1] Alex Markuze, Igor Golikov, C. D. 2021. Rethinking zero-copy networking with maio. *netdevconf*.

[2] Nabil Bitar, Jamal Hadi Salim, V. N. P. T. The battle of the zcs: Who is the prettiest of them all? https://netdevconf.info/0x19/sessions/talk/the-battle-of-the-zcs-who-is-the-prettiest-of-them-all.html.

[3] Pavel Begunkov, D. W. 2023. Fast zc rx data plane using io uring. *netdevconf*.

[4] Tariq Toukan, D. T. 2026. mlx5e high order rx pages. https://lore.kernel.org/all/20260223204155.1783580-1-tariqt@nvidia.com/.

[5] Torquati, M. 2010. Single-producer/single-consumer queues on shared cache multi-core systems. *CoRR* abs/1012.1824.

[6] zcrx: dma-buf support. https://lore.kernel.org/all/cover.1746097431.git.asml.silence@gmail.com/.

[7] zcrx: notifications. https://lore.kernel.org/all/20260518153532.2835502-1-cleger@meta.com/.

[8] zcrx: refill queue synchronous flushing. https://lore.kernel.org/all/cover.1763029704.git.asml.silence@gmail.com/.