Merging the Networking Worlds

David Ahern and Shrijeet Mukherjee
Enfabrica
{ david, shrijeet } @enfabrica.net

Abstract

Modern applications for high performance networking have universally turned towards RDMA and InfiniBand after declaring TCP/IP as too slow and inefficient. As discussed at Linux Plumber's Conference (LPC) 2022 [1], the core of the Linux IP/TCP/ethernet networking stack is capable of running at very high speeds (line rates of existing NICs and next versions), but changes are needed with respect to how the stack is used.

It has long been known that the current BSD socket APIs for sending and receiving data have too much overhead (e.g., system calls, memory allocations, memcpy, page reference counting) that severely limits the data rate, but the choice of simplicity of well-established interfaces and universal applicability has prevailed, and the interfaces have been hard to replace.

Linux has a few new design options for networking - namely, io_uring and XDP sockets. However, neither is a good end-to-end solution for achieving high data rates for a flow while leveraging the Linux protocol stack. io_uring mostly reduces the system call overhead and memcpy on Tx, while XDP sockets are a complete kernel bypass, sending packets directly to userspace buffers.

This paper discusses another option - merging some existing concepts of RDMA with traditional socket-based TCP/IP networking. Specifically, we show that one can leverage the existing IB verbs software APIs to create and manage memory regions between processes and hardware along with better integration of hardware queues and software queues to a kernel driver for submitting work and getting completions. This allows applications to submit pre-allocated and shaped buffers to hardware for zerocopy Rx and Tx, yet still leverage the kernel's TCP stack and its well established congestion control algorithms. This approach blends the fairly large application base using the verbs/RDMA interface with the familiar networking stack, management interfaces and wire protocols of TCP/IP, yielding reduced syscalls, better buffer management, and direct data access into userspace buffers.

This proposal is more about showing how the networking stack, its programming interfaces and control knobs are modular so that a developer can assemble the combination they want to get the best trade-off they seek. Further, this is actually doable today without a major departure from the use of existing Linux APIs.

1.0 Introduction

Modern applications like machine learning and disaggregated storage want to run single flow connections at high rates. It has long been known that the current BSD socket APIs for sending and receiving data have too much overhead, severely limiting the data rate. Linux networking has a number of "recent" changes to address some of the problems, but there is no end-to-end solution that will scale to next-generation line rates (400G and 800G).

Currently, applications have to either accept the overhead and limited flow rates of the socket APIs, or convert to a completely different paradigm, one that bypasses the networking protocols in the Linux stack. RDMA and InfiniBand, for example, have the mindshare when it comes to low latency, high throughput deployments while kernel bypass

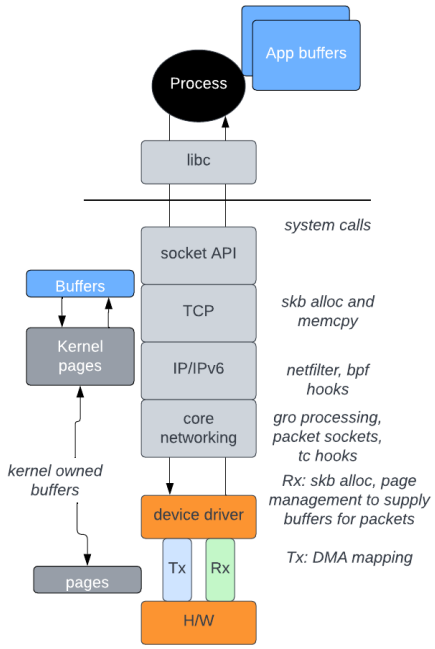

Figure 1. Overhead summary for BSD socket APIs.

frameworks, like DPDK, have significant traction as well.

This paper looks at how concepts from RDMA, along with the existing Verbs software APIs, can be used with socket based applications to get the best of both worlds - significantly reduced overhead in the datapath, while maintaining use of socket APIs and the Linux networking protocols for the control path. Let's start by reviewing the conclusions presented at LPC, and then discuss options to achieve the desired characteristics.

1.1 Improving Socket Based Networking

The core of the Linux IP/TCP networking stack is capable of running at very high speeds (line rates of existing 100G and 200G NICs and even versions on the horizon, 400G and 800G) [1], but changes are needed with respect to how the stack is used to bypass or eliminate unnecessary overhead in the datapath. Specific to this paper:

1. memcpy between user and kernel space limits data rates to at best 30Gbps. To achieve high data rates, an architecture is needed where hardware pulls (or lands) packet payload directly from (or into) application buffers.

2. Reduce or eliminate system calls. Calling `recvmsg` and `sendmsg` - even with high buffer sizes - adds significant overhead, which affects performance, latency, and CPU cycles needed for a given packet rate.

3. Avoiding the various infrastructure hooks (netfilter, tc, ebpf, packet sockets) in the networking stack for data path packets. Linux is a general purpose OS, and as such, has many hooks through the networking stack to implement policy decisions, such as managing allowed connections. Once the connection to a peer is established (i.e., the control path), the hooks are mostly overhead in the data path for many use cases. In the case of packet sockets (e.g., tcpdump), the hook severely affects the performance of all connections if a single packet socket is open as packets are cloned.

4. The ability for an application to manage buffers supplied to hardware to send and receive packet payloads. This avoids the need for per-packet or per-buffer reference counting on pages. In an architecture where 1. above is satisfied, the application using zerocopy interfaces knows best which buffers are available for incoming packets. Allowing userspace to manage a queue that supplies buffers to hardware moves the buffer management overhead to userspace, avoiding complexity in the kernel. Further, by registering that memory ahead of time, the CPU cost to pin pages and manage DMA mappings can be handled as part of the control path, rather than done on a packet-by-packet basis.

(Note, other important factors identified in [1] for high speed networking, such as a solid H/W GRO scheme, are not directly relevant for this paper, but do factor in based on the hardware-specific driver managing hardware queues for the QP.)

1.2 Zerocopy APIs

Linux has a zerocopy (ZC) API for Tx [2], [3] and Rx [4],[5]. The Tx API is fairly easy to use and removes the need to copy buffers from userspace to kernel when using `sendmsg`. While it avoids the need for memory copy, it retains the overhead of system calls to send packets and adds the need for more system calls to reap completions. In addition, it brings in the need to

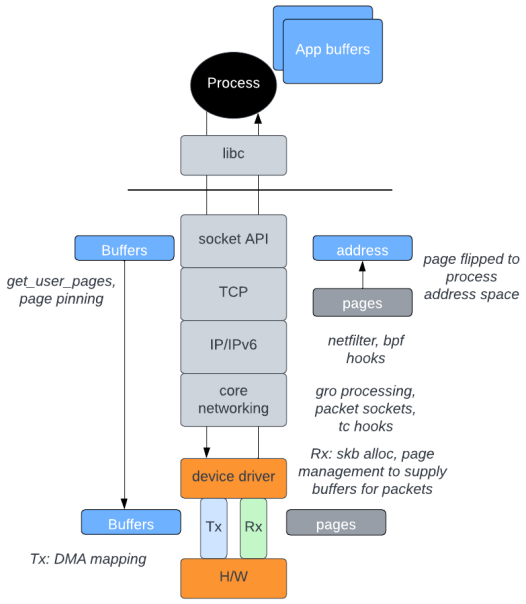

Figure 2. Process using Linux ZC APIs.

manage pages backing the userspace buffers (pinning and reference counting). Even with new sources of overhead, the Tx API brings a significant improvement for flow rates up to ~100Gbps; it alone is not sufficient to reach 200G and beyond.[1]

The Rx ZC API on the other hand is quite limiting. It requires a specific MTU such that packet headers land in one buffer to be consumed by the networking stack, and then payloads destined for a userspace process must consume an entire page, which is then flipped into the process' address space. A `setsockopt` system call is used to receive pending data at an address with a side buffer for payload fragments less than a page size. Given the constraints, the API is quite limited in applicability, and not practical beyond very specific environments.

1.3 io_uring

io_uring is a new take for asynchronous I/O with Linux starting with v5.1.[6] As shown in Figure 3, it provides software queues (Completion Queue and Submission Queue) between kernel and userspace, with the idea of reducing the number of system calls to submit work and to manage completions. The liburing library provides abstractions for applications to simplify interaction with the kernel APIs. One feature of note for this paper is that io_uring allows a process to register buffers with an io_uring instance to amortize the costs of using userspace buffers.

io_uring was started with the intent of speeding up file-based I/O, but quickly gained support for networking.[7] Networking applications use the normal socket API to establish a connection to a peer. Once established, the socket fd can be passed to an io_uring instance to manage the `sendmsg` and `recvmsg` calls via the SQ. With the 6.0 kernel release, io_uring gained support for the networking Tx ZC API which removes the need for `memcpy` on Tx and handles the ZC completions in the kernel, reducing the overhead for managing completions. In addition, the feature can be used with registered buffers to also reduce the overhead of pinning buffers when used with Tx ZC.[8]

In summary, io_uring for the networking datapath solves a few of the problems discussed earlier: it reduces system calls, avoids memory copy on Tx, simplifies overhead of ZC completions and the page management overhead for Tx buffers. However, as a generic, higher level kernel facility, io_uring does not work with (or have access to) hardware queues, packets go through the entire network stack (from driver to

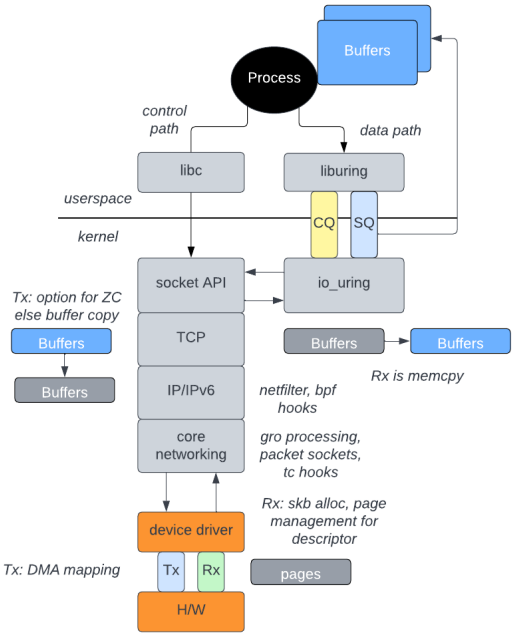

Figure 3. Process using io_uring for data path.

socket queue), and there is still the memcpy on Rx - all of which adds cpu cycles in the datapath and thus limits flow rates. Even if io_uring had the best possible performance, it still introduces a new application API, which means all the controls around pinning, placement, and logical buffer pools will need to be added to the interface.

1.4 XDP sockets

Linux v4.18 introduced a new option for networking in the form of AF_XDP sockets. [9],[10] AF_XDP sockets are designed to bypass the datapath of the Linux kernel, passing packets from the driver up to a userspace application via software queues.

Applications open an AF_XDP socket and register a buffer, called umem, with the hardware driver that owns the netdevice to which the socket is bound. The umem provides fixed and equally sized chunks for sending and receiving packets directly from userspace memory.

There are 4 software queues: a *Fill ring* for handing off buffers to the driver for use with hardware queues (e.g., ZC mode to avoid a memcpy), an *Rx queue* for the driver to tell the application about received packets, a *Tx queue* for an application to pass packets to the driver and a *Completion queue* for the driver to notify the application of sent packets (umem buffers are passed back to userspace).

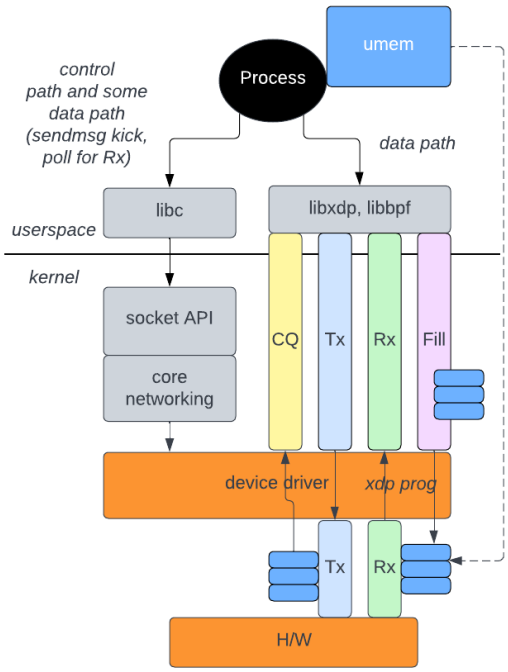

Figure 4. Process using XDP sockets.

An XDP program running in the driver picks which AF_XDP socket to land the packets. libxdp and libbpf provide the userspace abstractions for the kernel APIs, to ease writing and managing programs.

As a full-kernel bypass, AF_XDP by definition avoids all of the overhead discussed earlier, but it also bypasses the TCP/IP stack, affecting how networking is done by a program. In short, the application has to implement all of the networking protocols of interest to it in order to process and manage packets through the XDP socket. This paper is about how to leverage the kernel's TCP/IP stack, so the AF_XDP approach is not inline with this goal. What is relevant to this work is that AF_XDP is an example of a solution for Rx zerocopy where hardware directly lands packets into userspace provided buffers, along with S/W queues that cross the user-kernel boundary for reducing system calls when queuing and managing work. The former addresses limitations of an io_uring architecture in the form of Rx ZC.

The targeted architecture of this paper is, in a sense, a merging of AF_XDP and io_uring: dedicated hardware queues for a flow at the bottom enabling ZC in both directions, S/W queues at the top to avoid system calls, and retaining the hooks into the TCP/IP stack in the middle.

1.5 RDMA and IB Verbs

Another existing subsystem in the Linux kernel for networking is RDMA and the IB Verbs APIs. RDMA and InfiniBand have been around for over 20 years and have the mindshare for HPC type deployments looking for high throughput with low latency. InfiniBand, however, is a completely separate ecosystem from ethernet networking, consisting of different hardware, software and protocols. And it exists in a predominantly ethernet world, where a push for converged infrastructure has led to the adoption of RoCE, now in its second version. While ROCE allows RDMA to run over ethernet in fairly directed networks, it has been difficult to build shared use networks where TCP's

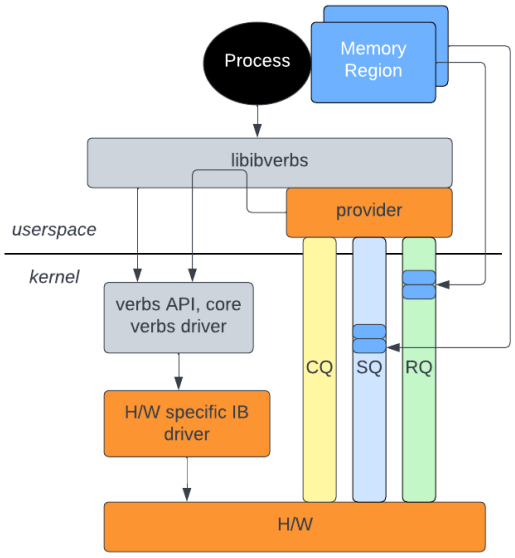

Figure 5. IB verbs software architecture

congestion control and performance in lossy environments is much better understood.[11,12]

The IB Verbs API is the software piece of this and is used for both InfiniBand and RoCE deployments, so really the "IB" in the name is a distraction - a legacy moniker. The effort in this paper targets re-use of these S/W APIs in conjunction with traditional socket based networking.

RDMA in general is a huge topic with a lot of concepts. This paper is only concerned with specific aspects of RDMA and specific capabilities from the S/W Verbs API, and all of these concepts of interest have overlaps with io_uring and AF_XDP.

In RDMA, memory regions (MR) allow applications to register buffers (contiguous memory) with hardware, amortizing the cost of pinning pages and mapping the addresses with hardware. It is useful to think of MRs as being analogous to AF_XDP's umem, and similar to registering buffers in io_uring.

A queue pair (QP) in RDMA is a set of receive and transmit queues between a userspace application and hardware. The queues are used to submit work request entries directly to hardware - e.g, posting buffers to receive data or posting buffers to transmit. Where AF_XDP relies on the driver to convert between XDP descriptors and hardware descriptors, RDMA allows applications to have direct access to hardware queues. The provider layer that hooks into libibverbs handles the vendor unique aspects of managing hardware queues in an application.

QPs have a completion queue for notification of work requests that have been completed. An application can have separate completion queues for send and receive, or one completion queue for both.

Finally, there is a protection domain (PD) that associates QPs and MRs from a security perspective.

RDMA has four basic operating transport modes: *Reliable Connection* (RC) has a single QP associated with only one other QP. Messages transmitted by the send queue of one QP are reliably delivered, in order to the receive queue of the other QP. RC mode is very similar to a TCP connection. With *Unreliable Datagram* (UD), a QP may transmit and receive single-packet messages to and from any other UD QP. Ordering and delivery are not guaranteed and delivered packets may be dropped by the receiver. Multicast messages are supported (one to many) with UD. UD is very similar to a UDP connection. The other 2 modes, *Reliable Datagram* (RD) and *Unreliable Connection* (UC), are not of interest here, and in fact we will focus exclusively on RDMA-RC for now.

The software architecture for RDMA looks very similar to XDP sockets, but with some key differences - namely the layers of software-hardware integration. In the IB verbs architecture, there is a hardware specific driver that handles the vendor unique details on the kernel side of the verbs APIs, and a userspace software provider that handles vendor unique implementations in userspace (e.g., managing hardware descriptors), with both going through common layers that carry much of the Verbs API.

From a data transfer perspective, RDMA is, in essence, an example of zero copy networking (ignoring many aspects of the larger RDMA concept and focusing solely on parallels with socket-based networking). For Tx, hardware reads from userspace memory, pulls that data into packets, adds protocol headers, and sends the packets to a peer. On Rx, hardware looks up a QP based on data in the packet headers and then writes packet

payload into application memory at a specific location - ie., ZC Rx.

Furthermore, it is quite common today for NICs to be able to run in either mode - ethernet or InfiniBand - with the toggle of a configuration setting. Clearly, there is already quite a bit of convergence between the two worlds.

On the S/W side the IB verbs API is just a means for configuring the hardware for a zerocopy flow and managing the details of an application directly accessing the hardware queues. As an existing API for setting up ZC networking, this effort only needs to focus on how to integrate the verbs API with socket networking.

2.0 Using RDMA Concepts with Linux Stack

The goal of this paper is to show how RDMA concepts can be used today to speed up socket-based networking. Specifically, the goal is for applications to continue to use socket APIs and the kernel networking layers to establish a connection to a peer. Then, applications that want the lower overhead path (e.g., need low latency or high throughput or wanting to reduce the CPU load) may opt-in to the Verbs APIs to set up the efficient datapath - one that gets the OS out of the way for actual payload data movement, but leverages the OS for packet headers and routing decisions.

Figure 6 shows such an architecture. Applications leverage a hardware vendor-provided kernel driver and libibverbs provider layer to obtain a more seamless integration of the data path between application and hardware, with much less overhead.

Applications use IB verbs calls to register memory with hardware, allowing hardware direct access to land and pull data for ZC networking. The QP concept is extended to have both hardware and software queues. The software queues avoid the need for system calls when submitting work and getting completions; the hardware queues provide a flow-unique resource for sending and receiving packets from the network. The hardware queues are directly managed by the vendor-provided kernel module, avoiding the need to translate between queue formats, and allowing an RSS rule to be created in hardware to direct packets for the flow to the correct queues. Further, depending on vendor choices and hardware design, the queues can just as easily be mapped directly to userspace (e.g., allow applications to submit buffers directly to hardware for Rx, bypassing any software based middle layer).

In addition, the kernel module takes ownership of the socket and directly interacts with the Linux networking stack, just like so many other in-kernel networking drivers (e.g., nvme over tcp and nfs). The in-kernel handling of the socket allows a more efficient Tx ZC API and handling of completions - something now realized with io_uring. The RSS rule tells the hardware which queue to use for the flow realizing a ZC Rx solution. Comparing Figure 6 to Figure 1, we see that all of the aforementioned overhead in the datapath of the Linux networking stack is now gone with one exception - the need to allocate skbs to interact with the TCP stack.

In a huge sense, this is the desired blending of io_uring and AF_XDP concepts, but done with a holistic approach to solving the networking overhead, rather than going at this piecemeal and stitching layers together. Further, by leveraging the existing verbs primitives, this architecture can be achieved today while avoiding the need to add new APIs specific to the

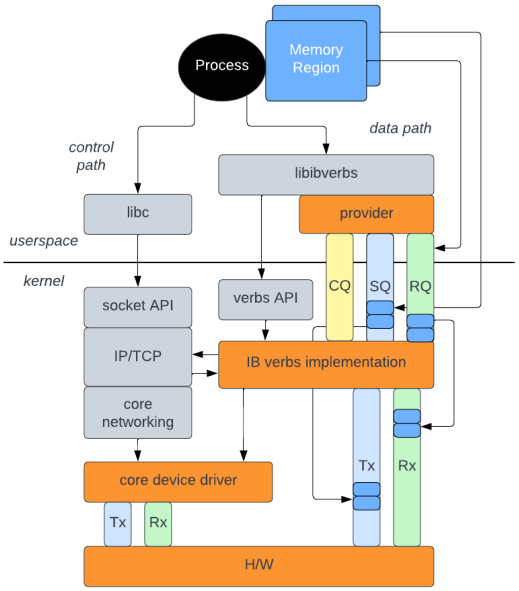

Figure 6. Socket networking with IB verbs.

Linux networking stack. Rather, all that is needed is vendor-unique glue in the form of a provider and kernel module to implement the two halves of the Verbs S/W APIs in a way that works with the vendor's hardware.

3.0 Example Application

At boot the hardware driver creates a netdevice (e.g., eth0) that represents the ethernet port on the NIC to the software networking stack. In addition, loading the associated IB driver (IB as in the vendor unique module that handles the S/W APIs) creates an RDMA virtual device to present the hardware port to the IB S/W stack.

An application wanting to leverage the IB verbs APIs will need to find and open the IB device that represents the physical port to get the supported attributes. This is done using the following libibverbs APIs:

```
ibv_get_device_list
ibv_get_device_name
ibv_open_device
ibv_query_device
ibv_query_port
```

From there the application gets a protection domain for its MR(s) and QP(s), `ibv_alloc_pd`.

Buffers to be used for sending and receiving messages are allocated using `mmap, memalign or malloc`, and then the buffers are registered with hardware as MRs using the `ibv_reg_mr` function.

Before creating the QP, the application needs to create 1 or 2 CQs using `ibv_create_cq`(); the CQs are passed in the attribute argument when creating a QP using `ibv_create_qp`() to associate the send and receive queues with a completion queue. The provider and driver implementations for the create_qp verbs API create hardware queues and software queues for the flow, and `mmap`'s the software queues to userspace (SQ to submit send requests, RQ to learn of filled buffers, and a WQ to submit buffers to hardware).

Once the QP is created, the application proceeds to establish a socket connection with a peer using the typical socket APIs and full Linux networking stack.
Once the socket is established, the QP is transitioned through various states INIT -> RTR - > RTS using `ibv_modify_qp`. During the transitions the application expecting incoming messages posts receive buffers using `ibv_post_recv`(), the data socket is handed off to the kernel module to manage the interaction with the TCP stack. At that point the kernel driver knows the 5-tuple and installs an RSS rule in hardware to steer packets for the flow to the socket specific set of hardware queues.

The sender posts send requests using `ibv_post_send`(). As the application adds entries to its SQ, it can either do a kick (i.e., system call) to tell the kernel module to send the buffer to the peer or the kernel module can use a kernel thread and spin poll checking the SQ. Either way, it sees the new entry and invokes `sendmsg` kernel side to push data through TCP using the zerocopy APIs. Transmit handlers send skb's (packets) back to the driver to submit to hardware via its hardware queues. Once the peer acknowledges the payload, a ZC callback is invoked, and the application is notified via a CQE that the peer has received the data.

On the receiver side, packets are received at the NIC, steered to the flow specific queues, and the packets are written into the application supplied buffers (ZC Rx). The kernel module manages the Rx descriptors for its hardware queue, creates an `skb` and pushes the received data through TCP to advance its state machine and ensure in-order delivery. The `sk_data_ready` hook for sockets is used to come back to the kernel module to handle the received data. Once a posted buffer has been consumed, it adds a completion event to the receive CQ, telling the application about the new data.

Applications on both ends poll for completions on the CQ using `ibv_poll_cq`(). Completions for the sender mean the buffer has been acknowledged by the peer; completions for the receiver indicate a posted buffer has been consumed.

4.0 Summary

Modern applications want to run single flows at high rates. The Linux TCP/IP stack is more than capable, but a design is needed that avoids the unnecessary overhead in the datapath (i.e., the OS needs to get out of the way). This paper shows how concepts from RDMA along with the existing IB Verbs S/W API provides such a solution allowing traditional socket based communication via the TCP/IP networking stack to run at higher speeds.

The proposal outlined in this paper is really not that radical of an idea. RDMA is in many ways a form of zerocopy networking. Further, comparing the software architecture to io_uring and AF_XDP, there are a number of common factors: userspace memory registered with a kernel driver and hardware, software queues to avoid/reduce system calls, hardware queues specific to an application flow, and a library to simplify application development. Applications wanting better performance opt in to APIs that remove overhead.

5.0 Acknowledgements

The work presented in this paper is a collaborative effort from the entire software engineering team at Enfabrica.

6.0 References

[1] David Ahern, “Can the Linux networking stack be used with high speed applications?,” Linux Plumbers Conference 2022, September 2022. https://lpc.events/event/16/contributions/1345/

[2] Willem de Bruijn, Eric Dumazet, “sendmsg copy avoidance with MSG_ZEROCOPY,” netdevconf 2.1, April 2017. https://legacy.netdevconf.info/2.1/papers/debruijn-msgzerocopy-talk.pdf

[3] Jonathan Corbet, “Zero-copy networking,” LWN, July 2017. https://lwn.net/Articles/726917/

[4] Eric Dumazet, “TCP/misc works,” netconf 2018. http://vger.kernel.org/netconf2018_files/EricDumazet_netconf2018.pdf

[5] Jonathan Corbet, “A reworked TCP zero-copy receive API,” LWN, May 2018. https://lwn.net/Articles/754681/

[6] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf

[7] Jonathan Corbet, “The rapid growth of io_uring,” LWN, January 2020. https://lwn.net/Articles/810414/

[8] Pavel Begunkov, “io_uring zerocopy send,” https://lore.kernel.org/netdev/cover.1657643355.git.asml.silence@gmail.com/

[9] AF_XDP Documentation in Linux source tree, URL: https://www.kernel.org/doc/html/latest/networking/af_xdp.html

[10] Magnus Karlsson, Björn Töpel, “The Path to DPDK Speeds for AF_XDP,” Linux Plumbers Conference 2018. URL: http://vger.kernel.org/lpc_net2018_talks/lpc18_paper_af_xdp_perf-v2.pdf)

[11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitendra Padhye, Marina Lipshteyn, “RDMA over Commodity Ethernet at Scale,” SIGCOMM, 2016, https://dl.acm.org/doi/10.1145/2934872.2934908

[12] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, Scott Shenker, “Revisiting Network Support for RDMA,” SIGCOMM, 2018, https://dl.acm.org/doi/10.1145/3230543.3230557

[13] Roland Dreier, Jason Gunthorpe, “RDMA programming tutorial” https://netdevconf.info/0x16/session.html?RDMA-programming-tutorial