AF_PACKET V4 and PACKET_ZEROCOPY

Magnus Karlsson, Björn Töpel, and John Fastabend[†]

Intel Inc.
Stockholm, Sweden
{magnus.karlsson,bjorn.topel}@intel.com

[†]Covalent IO
Portland, OR
john.fastabend@gmail.com

Abstract

In this paper, the RFC of AF_PACKET V4 and PACKET_ZEROCOPY is presented and discussed. AF_PACKET V4 is a proposed new interface optimized for high performance L2 packet processing. The interface supports zero-copy semantics, to remove expensive memcpy operations with the PACKET_ZEROCOPY setsockopt option. Experimental results on a set of micro-benchmarks show performance improvements up to 40 times, while tcpdump/libpcap extended with V4 support shows a performance improvement of around 20x. The techniques are fully integrated with eXpress Data Path (XDP) and a possible path forward for a unified implementation between XDP and PACKET_ZEROCOPY is presented.

Keywords

Networking, Linux, AF_PACKET, raw sockets, packet processing, zero-copy.

Introduction

Internet and networking as such has quickly become a vital part of our society. The number of services that is more efficiently and better delivered over the Internet has sky rocketed and with this comes an ever increasing demand for more bandwidth. 10, 25 and 40 Gbit/s Ethernet is the standard today and 50, 100 and 200 Gbit/s speeds are around the corner. To process packets, programmers are today using the highly successful and popular socket abstractions, for example datagram (AF_INET) and raw sockets (AF_PACKET V2 and V3) on the L2 level. However, the problem with these are that they top out around 1 Mpps per core [7] or lower making it hard to get applications to scale to higher networking speeds. As a response to this, solutions such as RDMA [8], Netmap [9], PF_RING [1], or bypass solutions such as DPDK [2] and vendor specific SDKs such as ExaNIC [3] and OpenOnload [11] have been developed. While they offer higher network processing speeds, the problem is that they are quite different from standard sockets and thus require extensive application rewriting. Moreover, as they are not part of Linux, a number of good Linux features such as process isolation, power management, hardware agnostic scheduling, and the security feature ASLR cannot be used which can be detrimental to systems building. The problem then becomes how to design a high performance, HW agnostic and secure packet processing interface that is a logical continuation of the current popular Linux socket interfaces?

In this paper, we present AF_PACKET V4 (abbreviated as V4) and PACKET_ZEROCOPY (ZC). V4 is a proposed new raw socket interface optimized for high performance packet processing. It supports zero-copy semantics to remove the performance penalty of memcpy operations, as well as optimized memcpy semantics to support cases where hardware cannot support zero-copy. When a V4 socket is created without the ZC option, each packet is sent to the Linux stack and a copy of it is sent to user space, so V4 behaves in the same way as V2 and V3.

We then introduce a new optional setsockopt option called PACKET_ZEROCOPY, enabling true zero-copy and zero syscall semantics on the socket, while still maintaining Linux security and isolation properties. This is achieved by mapping the NIC packet buffers into the user space process' memory space, but the HW descriptors are only mapped to kernel space. User space only sees HW agnostic virtual descriptors, and it is the kernel's responsibility to translate between the virtual user space descriptors and HW descriptors. By default in this mode, a packet destined for this socket goes only to user space. A packet destined for the kernel stack is copied out of the packet buffer (not zero-copy anymore) or filtered out by the NIC (still zero-copy) to another ring even before it gets to the user space packet buffer. This way user space cannot manipulate or see kernel data. The packet destination process is determined by programming flow director or RSS with tc or ethtool. With some possible future extensions, XDP (eXpress Data Path) is also a flexible candidate for determining the destination.

V4 plugs into the existing XDP infrastructure even when ZC is enabled. XDP programs will be executed on a supplied page from the V4 packet buffer and XDP_PASS passes a packet to the V4 user space packet buffer, without any copies, when a packet is destined for a V4 process with ZC enabled. The goal is that if you implement ZC support in a driver, you get XDP support for free, hopefully speeding up the adoption of both XDP and ZC, though the current code base is not there yet. We are also planning on proposing an XDP_PASS_TO_KERNEL option that will copy the packet over to an SKB and pass it to the kernel stack in ZC mode.

To illustrate the approach, we have implemented support on Intel I40E NIC [5] and veth, but it should hopefully be easy to port to other NICs and virtual devices. We have compared the performance of V2 and V3 against V4 with and without ZC on a number of micro-benchmarks as well as tcpdump/libpcap [6]. We found that the micro-benchmark throughput improvements on the Intel I40E NIC are between 4x to 40x (400% to 4000%) for V4 with ZC and more of mixed bag for V4 without ZC as we see some performance decreases of up to 7% but mainly increases of up to 10% (how to optimize this is discussed in the paper). For veth, the improvements are up to 6x for V4 wit ZC. It was easy to port libpcap to V4 as it already supports V2 and V3. For tcpdump/libpcap, the increase in max number of 64-byte packets that could be captured was around 20 times higher for V4 with ZC than with V3. These are really significant performance increases that we hope will translate to a performance boost in real life applications too.

This paper is outlined as follows. The first section goes through the goals of AF_PACKET V4 and its architecture, while the next one shows how these two features are implemented. This is followed by the experimental methodology and experimental results comparing V2, V3 and V4. The next section gives an overview of related work in the area. Future work is reported and finally the paper is concluded.

Design

In this paper, two new features are proposed: AF_PACKET V4 (V4) and PACKET_ZEROCOPY (ZC). V4 is a new user space interface for AF_PACKET designed to behave largely in the same way as the previous versions V2 and V3 for compatibility and portability reasons, but to offer higher performance and more transparent error reporting. All packets are sent to the kernel stack and a copy of the packet is sent to user space through the V4 interface. V4 does not have to use syscalls that perform a mode or context switch to kernel space for the RX path, though the TX path still uses a syscall to kick the TX code, but this could likely be optimized away in the future. ZC is a new option to setsockopt that puts a V4 socket in zero-copy mode for even higher performance. At this point, packets only go to user space and none go to the kernel stack by default, so the semantics are different.

V4 executing without ZC, is referred to as *copy mode* and V4 with ZC is referred to as *zero-copy mode* or just *ZC*.

Motivation and Goals

Why a new format? To motivate that let us first take a look at the design goals of V4 and ZC. They are as follows:

1. Be an extension to AF_PACKET V2 and V3 so that an application written for these are as easy to port as possible
2. To abide by all the security and isolation rules of Linux user space
3. Complete HW agnostic interface for application portability
4. Better performance than V2 and V3
5. Eliminate copies when packets need to be transmitted or buffered
6. Transparent error reporting for each packet sent and received
7. Work together with XDP as it could greatly enhance the usefulness of V4 in ZC mode
8. If ZC support is implemented in the driver you should get XDP support for free. This is to facilitate adoption of both ZC and XDP in drivers.

While items 1 to 4 likely need no further motivation since programmers usually want easy, secure, but faster networking performance, items 5 - 8 do. In item 5 we would like to avoid having to copy a received packet when transmitting it as is required by V2 and V3. Moreover, when implementing some packet inspection functionality such as DPI or a telecom protocol such as GTP, PDCP, RLC or base-band MAC scheduling and L2 processing, we would like to buffer some packets either to reassemble some higher level protocols (DPI) or to be able to resend them later if an acknowledgment was not received. Therefore it is important that buffering does not require the application writer to pay the penalty of copying out the packet to some temporary buffer as this costs in performance. In V2 and V3 it is necessary to do this in order to use the descriptor again, but with V4 we want to just leave the packet as is and not pay this penalty.

The transparent and rapid error reporting is especially important in systems where certain packets are more important than others and/or the relevance of a packet quickly decreases with time. Mobile communication protocols are one good example of an area that has both of these properties. When a packet is received by the base-band stack it is transformed to a digital version of a wave that is going to be sent over the air. This wave (and the time it is sent) is optimized so that it will reach its mobile device target with as few errors as possible given an energy budget and the current quality of the air environment between the antenna and the mobile device. Unfortunately, the environment changes very rapidly so a packet might only be good for 1 ms. After that, it will not reach its target without numerous bit errors. With these kind of protocols, it is important to get feedback on errors quickly so packets can be resent immediately, otherwise we have to waste cycles to recompute the packet.

XDP (eXpress Data Path) is rapidly taking off as a key Linux networking technology for high performance packet processing operations. It executes secure code that can be injected from user space. This XDP code is executed right after the driver, but before an SKB is allocated, and can look at the packet and make decisions if it should be dropped, forwarded or sent up the Linux stack. It is our belief that XDP can, with some possible future extensions outlined in the future work section, be used for a number of interesting features in conjunction with V4 and ZC such as packet scheduling, load balancing, and supporting arbitrary descriptor formats (e.g. virtio-net). XDP is ideally placed right after the driver to provide these kind of services for V4 in ZC mode. As of today, it can already be used with ZC for introspection, DDoS mitigation, etc, but we believe this is only the start. To achieve this, it should be as easy as possible to adopt ZC and XDP. If you implement ZC support in a driver you should get XDP support for free. One implementation supporting both features is the goal.

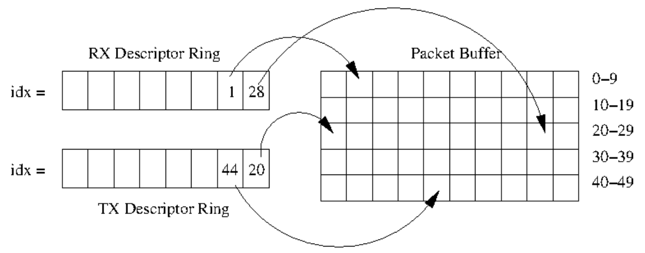

Figure 1: There are three buffers shared between the kernel and one user space process in V4: the RX descriptor ring, the TX descriptor ring, and the packet buffer. The descriptor rings point to packets located in the packet buffer. There are no packets in the descriptor rings themselves.

How do we then offer better performance than V2 and V3? In short, it is the combination of the following techniques:

- Batching of packets for sending and receiving
- Poll mode support to avoid taking interrupts when not necessary, in the same spirit as NAPI
- No syscalls in the data path (Current implementation still requires syscall for TX operations)
- Support a zero-copy mode for even higher performance
- Lock-less structures between user and kernel space
- Storing packets separately from descriptors in order to avoid copying memory
- Make deviations from V2 and V3 when this offers a large performance boost and the impact on contemporary applications is small

AF_PACKET V4

Let us start with the V4 in copy mode and how it achieves the goals outlined above. V4 uses three shared memory areas to communicate between kernel and user space as seen in Figure 1. Just as V2 and V3, it has one RX ring area containing *descriptors* that describe received packets and one TX ring area with descriptors that describe packets user space wants to send. The third area is the *packet buffer*, the area containing the packets themselves. So in contrast to V2 and V3, descriptors point to packets located in the packet buffer. There are no packets in the RX or TX descriptor rings as all packets reside in the packet buffer. The descriptor rings are shared between a single user space process and the kernel. They can never be shared between multiple user space processes, therefore every process will have its own set to communicate with the kernel. The packet buffer can, on the other hand, be shared between processes if desired.

As seen in Figure 2, V4 is configured much in the same way as V2 and V3. The only difference is that we have to register the packet buffer with the kernel using the new PACKET_MEMREG setsockopt option. Note that the socket used in creating the packet buffer is fed into the setsockopt creating the RX and TX descriptor ring to associate a ring with a packet buffer. In the example code, we have created a new socket for the packet buffer so that if the application programmer would like to share the packet buffer with another process, it could do so by sharing the file descriptor. Another use case is if you want one packet buffer for RX and another one for TX, as this can make sense if they do not share packets. But you could of course use the same fd as the one used for the RX and TX descriptor rings. The rest of the configuration is equivalent with V2 and V3. Note that V4 only accepts the PF_PACKET family and the ETH_P_ALL protocol options.

The descriptor format of V4 is shown in Figure 3. Each descriptor is 16 bytes, so on a system with 64 byte cache lines there are four descriptors on each cache line. The descriptor consists of a 64 bit index into the packet buffer to which the descriptor ring is associated, the length of the packet, and the offset (in bytes) into the packet where packet data starts. This might be > 0 if the user has explicitly requested some headroom using the PACKET_RESERVE setsockopt option, XDP has added headroom, or if the HW device has added some metadata. There is also an errno number passed along with each descriptor so that we can get errors on each single transaction, and finally a flag field. The flag field is currently used for indicating if a descriptor is ready for consumption in user space or in kernel space, called TP4_DESC_KERNEL, and another flag for indicating if a packet continues in the next frame, called TP4_PKT_CONT. For the rest of the paper we will refer to a *packet* as a logically contiguous set of data bytes that is contained in one or more fixed-sized *frames* that do not necessarily have to be contiguous. Each descriptor points to a single frame. A packet can be of variable size but all frames are of the same size (between 2K and PAGE_SIZE), set in the PACKET_MEMREG call. Default frame size is 2K as this is large enough for a maximum sized standard Ethernet frame.

One thing that deviates from both V2 and V3 is that we do not have a packet header in V4, and the reason for this is performance. We do not want the kernel to touch the packet data as this will greatly reduce the performance (by up to 50% in ZC mode). Only the application should touch the packet data. If the kernel touches the packet data, it needs to be brought into the cache on which the RX softirqd is running. The application is, in all likelihood, running on another core, so the packet needs to be transferred from the softirqd core to the application core, incurring yet another cache transfer. Having the kernel touch the packet data also interferes with prefeching in the application.

This also means that we cannot have a *struct sockaddr_ll* structure passed along with the packet in V4, but we do not actually need the information stored there. This because we made a design choice in V4 that packets will not start to flow until you have made a `bind()` call. This means a V4 application knows what family, protocol, and ifindex that were used to do the bind. Most of the information in struct sockaddr_ll is for the case when traffic flows from all interfaces before a bind, a case that seems to be rarely used these days. Removing it makes the implementation faster and simpler. As for the sll_pkttype and sll_addr fields, these can be trivially parsed by the application if it wants them. We do not want to incur a performance penalty for all applications not

```
int tpver = TPACKET_V4;
sfd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
setsockopt(sfd, SOL_PACKET, PACKET_VERSION, &tpver, sizeof(tpver));
fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
mreq.addr = start_of_packet_buffer;
mreq.len = length_of_packet_buffer;
mreq.frame_size = 2048;
setsockopt(fd, SOL_PACKET, PACKET_MEMREG, &mreq, sizeof(mreq));

req.mr_fd = fd;
req.desc_nr = number_of_descriptors;
ret = setsockopt(sfd, SOL_PACKET, PACKET_RX_RING, &req, sizeof(req));
ret = setsockopt(sfd, SOL_PACKET, PACKET_TX_RING, &req, sizeof(req));

/* txring is, as in V2/V3, mmapped in the same mmap call. */
rxring = mmap(0, 2 * req.desc_nr * sizeof(struct tpacket4_desc),
              PROT_READ | PROT_WRITE, MAP_SHARED | MAP_LOCKED | MAP_POPULATE, sfd, 0);
txring = rxring[req.desc_nr];

/* Only for ZC mode */
setsockopt(sfd, SOL_PACKET, PACKET_ZEROCOPY, &queue_pair, sizeof(queue_pair));

ll.sll_family = PF_PACKET;
ll.sll_protocol = htons(ETH_P_ALL);
ll.sll_ifindex = if_nametoindex(interface_name);
bind(sfd, (struct sockaddr *)&ll, sizeof(ll));
```

Figure 2: The configuration of V4 in pseudo-code.

```
struct tpacket4_desc {
        __u64 idx;
        __u32 len;
        __u16 offset;
        __u8  error; /* an errno */
        __u8  flags;
};

struct tpacket4_queue {
        struct tpacket4_desc *ring;

        unsigned int avail_idx;
        unsigned int last_used_idx;
        unsigned int num_free;
        unsigned int ring_mask;
};
```

Figure 3: The descriptor format and queue structure of V4. Note that there is no data header in V4.

needing these fields.

The V4 descriptor ring format shown at the bottom of Figure 3 is a lock-less single producer, single consumer queue heavily inspired by a proposal by Michael Tsirkin [12] and another one by Rusty Russel [10].

RX operations from user space simply check if the TP4_DESC_KERNEL flag is not set for the current entry in the descriptor ring. If it is, it retrieves the descriptor and advances to the next one until it either finds that the flag is not set or the configured packet batch size is reached. Once it has completed processing the packets, they are sent back to the kernel either through the TX descriptor ring (by setting all the info in the descriptor plus setting TP4_DESC_KERNEL) or if the packet should not be sent, by putting it in the RX descriptor ring. So no syscalls are needed for the RX path. However, in the current implementation packets are not guaranteed to be sent until after you have called a send syscall such as sendto().

PACKET_ZEROCOPY

To be able to offer significantly higher performance we propose to introduce a new zero-copy mode for a V4 socket invoked with the setsockopt option called PACKET_ZEROCOPY. It sets up the HW NIC driver so that packets are DMA'ed straight into user space. It is important to note that TX and RX descriptors are still only mapped to kernel space, as seen to the right in Figure 4. The kernel will make sure that anything user space tries to do with these descriptors is validated so that it cannot negatively affect the kernel or other user space processes. This is also a requirement to be able to maintain a HW agnostic API to user space. The price you pay for this performance boost is that the semantics will be different. Since it is zero-copy, it requires support in the driver, and some Linux networking functionality (such as qdisc and netfilter) will have no effect on the packet anymore. All this will be explained in the rest of this section.

Even in ZC mode, it is of course mandatory that the Linux security and isolation rules for user space are followed. Our implementation has the following properties in user-space:

- User-space cannot crash the kernel or another process
- User-space cannot read or write any kernel data or packets

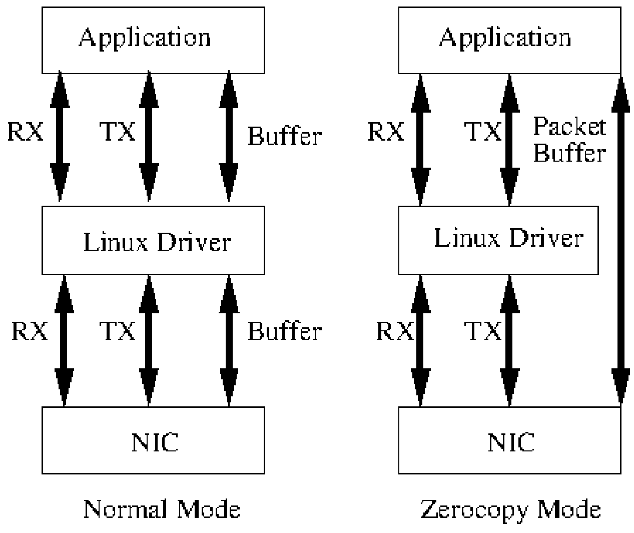

Figure 4: In copy-mode all the three areas go through the kernel, but in ZC mode, the packet buffer area is directly mapped between the device and user space. Note though, for isolation, security, and HW agnostic reasons, the RX and TX HW descriptors are still only mapped to kernel space.

- User-space cannot read or write any data or packets from other user space processes, unless this data is explicitly shared

How do we then make sure that these rules are followed with ZC? To make sure that user space cannot crash the kernel or another process, the HW descriptor rings are only visible and modifiable by the kernel. User space only sees the virtual V4 descriptor rings. To be able to make sure that one process does not see any other processes packets or the kernel's packets we need to have one dedicated RX/TX queue pair in the HW for each user space process. This queue pair is filled only with data buffers from the packet buffer of that application (these are packet buffers explicitly sent down by the application), so any packet arriving on that queue pair will be put in a buffer belonging to that user space process. How do we then make sure that only packets destined for that specific user space process get in the queue in the first place? For this we need to activate HW classification support in the NIC, unless the application should get every single packet arriving on a network interface.

What happens then if the NIC cannot support this due to e.g. some exotic protocol or just old NIC HW? In that case, packets will be DMA'ed into the kernel and copied out to the relevant packet buffer in user space according to the desired packet distribution policy (so not true zero-copy anymore). As will be discussed later, we believe that XDP is a good and flexible candidate for this functionality as it accepts validated programs from user space that will be executed before the packet is sent to the V4 user space.

ZC is enabled with the setsockopt shown in Figure 2. It takes a queue pair as argument which refers to the queue pair in the HW that should be used to provide the zero-copy packet buffers. All traffic arriving on this queue pair will go to this user space process. As an example, let us assume that the system has eight cores. When the NIC driver is loaded, it will assign one queue pair per core, that is eight in this case. RX and TX (in copy mode) will use the queue pair local to the cores that they are executing on. The application programmer then has to pick one of these, or create a new queue pair, on which ZC should be activated. Using ethtool or tc, the HW can be configured so that only relevant packets are sent to that application. This could be something as simple as saying that all packets from an interface should be sent to queue pair X, or that all packets should be mirrored to queue pair X (as in a tcpdump or wireshark scenario), or that application X should only get packets with a certain IP address and application Y should get packets with another IP address.

In order to offer a large performance boost with ZC, we decided to have the packets leave the kernel straight after XDP in the driver (or right after the driver if there is no XDP support), before putting it into an SKB. This is the reason that networking features, such as qdisc and netfilter, present in the socket path will not work when ZC is enabled. ZC support put into the existing SKB path did not offer more than a marginal performance boost.

One option that we considered instead of having an explicit queue pair number in the setsockopt was to just take over the whole netdev to which we were bound. For example, binding to eth0 would provide all the traffic from eth0, while binding to eth0_v1 would give us whatever traffic the eth0 HW has been configured to send to eth0_v1. So if we had multiple apps, we would have to create multiple netdevs on the device with, for example, ethtool, tc, or HW specific procfs or debugfs interfaces. While we believe this would be more elegant in that it does not expose queue pair to the application, it has the drawback of potentially creating a plethora of netdevs. It is also unclear if all HW devices would support creating more netdevs like this.

XDP Support

XDP is very well suited to enhance and work in conjunction with ZC as it is executed right after the driver. XDP can as of now be used for a number of things in conjunction with V4 and ZC as e.g., introspection, debug, and forwarding. All the XDP actions will work as implemented and XDP_PASS has been extended to pass the packet straight to the V4 packet buffer without any copies in ZC mode. But what if the XDP program would like to pass the packet to the kernel stack even when executing in ZC mode? For this we propose to introduce a new action in XDP called XDP_PASS_TO_KERNEL that will copy the packet into an SKB and send it off to the networking stack in the standard way. Note that the packet is copied into the SKB, since user space should not be allowed to see or be able to modify the packet after we have sent it to the kernel stack. The XDP_PASS_TO_KERNEL defaults to XDP_PASS in the non ZC case.

Implementation

Note that this implementation section is a snapshot of what the code looks like right now and will change as we get feedback.

```
enum tp4_netdev_command {
      TP4_ENABLE,
      TP4_DISABLE,
};

struct tp4_netdev_parms {
      enum tp4_netdev_command command;
      void *rx_opaque;
      void *tx_opaque;
      void (*data_ready)(void *);
      void *data_ready_opaque;
      void (*write_space)(void *);
      void *write_space_opaque;
      void (*error_report)(void *, int);
      void *error_report_opaque;
      int queue_pair;
};

int (*ndo_tp4_zerocopy)
          (struct net_device *dev,
           struct tp4_netdev_parms *parms);
int (*ndo_tp4_xmit)
          (struct net_device *dev,
           int queue_pair);
```

Figure 5: The two new netdevice operations required by ZC.

Copy mode is implemented in `af_packet.c` as it uses SKBs. The control path of ZC shares the same code as copy mode in af_packet.c, but it also needs to call the driver to set up or tear down the DMA mappings of the packet buffer. The data path for ZC, on the other hand, is completely implemented in the driver. To support this, we introduced two new netdevice operations in `netdevice.h` as shown in Figure 5. The first one enables or disables zero-copy support. It takes a structure with a number of parameters that is supplied by the code in af_packet.c, among those pointers to packet arrays (described in the next section) for TX and RX, callbacks to use when issuing a poll() from user space, a queue number to bind to, and a callback with which to report errors. The second NDO tells the transmit code to start to send messages that are in the V4 TX queue.

In addition to the implementation supporting the performance enhancing techniques reported earlier, we had three implementation specific goals:

- Making the implementation of ZC as easy as possible
- To abstract away the actual implementation of the V4 format, so that the same driver code could be used without any modifications for SKBs, V2, V3, virtio-net or any other format in ZC mode (see future work for a discussion)
- To get XDP support for free when implementing ZC. Much better to just have one implementation for both functions to facilitate adoption.

In order to support these requirements we have introduced *packet arrays*.

Packet Arrays

A *packet array* is a collection of frames that represents a number of packets. When ZC is enabled, the driver creates at least two packet arrays, one for RX and one for TX, using the first two calls in Figure 6. The first parameter tells the packet array where to get its packets from and where to send them when they are finished, which in our case is one of the V4 descriptor queues. The second parameter decides how many frames the packet array can hold, and then finally a pointer to the device this packet array is associated with. Note that the af_packet.c file also uses packet arrays to implement the copy mode path of V4 as we found these concepts made the implementation much easier.

In the data path in Figure 6, we start by calling `tp4a_populate` to fill the packet array with frames. For an array connected to an RX queue, these are empty buffers that the application wants the kernel to put packets into so it can receive new packets. For TX, these are packets that the process wants us to transmit. Validation of user space data is performed in the populate function according to the policy set by the enum tp4_validation in Figure 8. For RX this is set to TP4_VALIDATION_IDX as we only need to validate that the index field points to an entry inside the packet buffer. All other entries in the descriptor will be overwritten by the kernel. For TX, on the other hand, we need to perform a full validation of the descriptor as this data will be used by the kernel. This is accomplished by setting the validation to be TP4_VALIDATION_DESC, which validates all fields in the descriptor. Invalid entries are not put in the packet array and are immediately sent back to user space with the appropriate errno marked in the error field of the descriptor. Note that these validations and DMA directions are automatically set up by the `tp4a_rx_new` and `tp4a_tx_new` functions.

The driver will then process each packet by calling `tp4a_next_packet` in a loop to retrieve each packet in the packet array and process it. The same operation can be done on frames instead by using `tp4a_next_frame`. Once there are no more packets in the packet array (or the programmer has decided that enough packets have been processed, due to, for example, a NAPI poll weight threshold) we simply call `tp4a_flush` to write the completed descriptors to user space. For RX, these will be packets that user space receives. For TX, these will be completions telling the application that exactly these frames/packets have been successfully transmitted.

Frames and packets are represented by *frame sets*. A frame set can contain one or more frames that could be anything from part of a packet to several packets. The functions operating on frame sets are shown in Figure 7 and are used to get frame properties such as length and data pointers, as well as setting properties on the frame. This way the actual descriptor format used in user space is not exposed to the driver and the same code can be used to support a number of different interfaces. See the future work section for more on what interfaces could potentially be supported in the future.

Example: HW NIC Driver I40E

Intel® Ethernet 700 Series [5] is a range of contemporary network adapters that support 1GbE, 10GbE, 25GbE, and40GbE. The Linux kernel device driver supporting this range of NIC is "i40e", residing at `drivers/net/ethernet/intel/i40e` in the kernel

```
In the control path:
struct tp4_packet_array *tp4a_rx_new(void *rx_opaque, size_t elems,
                                     struct device *dev)
struct tp4_packet_array *tp4a_tx_new(void *tx_opaque, size_t elems,
                                     struct device *dev)

In the data path:
void tp4a_populate(struct tp4_packet_array *a);
while (bool tp4a_next_packet(struct tp4_packet_array *a, struct tp4_frame_set *p) {
        process_packet(struct tp4_frame_set *p);
}
int tp4a_flush(struct tp4_packet_array *a);
```

Figure 6: Main operations on packet arrays and conceptual programming flow.

```
bool tp4f_next_frame(struct tp4_frame_set *p)
bool tp4f_prev_frame(struct tp4_frame_set *p)
u64 tp4f_get_frame_id(struct tp4_frame_set *p)
u32 tp4f_get_frame_len(struct tp4_frame_set *p)
u32 tp4f_get_data_offset(struct tp4_frame_set *p)
void *tp4f_get_data(struct tp4_frame_set *p)
bool tp4f_is_last_frame(struct tp4_frame_set *p)
void tp4f_set_error(struct tp4_frame_set *p, int errno)
void tp4f_set_frame(struct tp4_frame_set *p, u32 len, u16 offset, bool is_eop)
```

Figure 7: Example operations on frame sets producing either frames or packets if it has packet in its name.

```
enum tp4_validation {
      TP4_VALIDATION_NONE,
      TP4_VALIDATION_IDX,
      TP4_VALIDATION_DESC
};

struct tp4_packet_array {
      void *queue_opaque;
      struct device *dev;
      enum dma_data_direction direction;
      enum tp4_validation validation;
      u32 start;
      u32 curr;
      u32 end;
      u32 mask;
      void* items[0];
};
```

Figure 8: A packet array represents a batch of frames/packets that the driver can operate on, independent on the format or the final destination of the frames.

tree.

The implementation consists of four parts. First, instead of allocating ingress buffers from the page allocator, we need to "allocate" buffers by dequeuing buffers from the V4 queue. Second, when an Rx buffer is filled, we need to complete the buffer to user space. This is an asynchronous event, so we need to track state from the V4 queue. In the regular SKB path, the ingress HW descriptor ring is augmented by the `i40e_rx_buffer` struct to track meta information about the HW descriptor. Third, `ndo_start_xmit`-like functionality for the V4 queue, i.e. pull Tx V4 buffers from user-space and place them on the HW Tx descriptor ring. Fourth, completing a transmitted Tx frame back to the V4 queue. Again, this requires tracking meta information about the descriptor.

All parts above fit very nicely into the packet array structure described in the Implementation section. The first part, allocation, is exposed to the driver via the packet array function and getting buffers from user space is done by simply calling `tp4a_populate`. Tracking V4 meta information is done implicitly by the packet array, so no additions were needed in the `i40e_tx_buffer` and `i40e_rx_buffer` structures. Completing entries back to user space is simply a `tp4a_flush`. For the transmit path, pulling frames from user space is, again, the populate function, and completion is a simple flush.

The current implementation does all work, i.e. allocate, update Rx/Tx HW descriptors and completion from a NAPI poll function, to minimize explicit locking.

Example: Virtual Driver veth

Veth is a virtual Ethernet device commonly used with the Linux bridge, SW switches and containers. We used it as an example of how ZC could be supported in a virtual device that is usually used to communicate between two processes. The implementation with packet arrays becomes small. Each process that enables zero-copy creates two packet arrays: one for RX and another one for TX. The code in the data path is then what is shown in Figure 9.

Note that this code works even on SKBs. Let us say that the other process does not have ZC enabled, but the calling process does. In this case `tp4a_flush` will allocate an SKB for each packet and transfer the descriptor metadata and the packet to the SKB so that the receiving process will get an

```
tp4a_populate(my_tp4a_tx);
tp4a_populate(other_process_tp4a_rx);
tp4a_copy_packets(other_process_tp4a_rx,
                  my_tp4a_tx);
tp4a_flush(other_process_tp4a_rx);
tp4a_flush(my_tp4a_tx);
```

Figure 9: The data path of ZC in veth written in pseudo-code.

SKB sent to it. The function `tp4a_populate` can also populate the array with packets from SKBs to support the case when we would like to send data from a process without ZC enabled to one that has it enabled. Finally, the code can also be used for the SKB to SKB case, but it is not efficient since (at least the current implementation) will unnecessarily convert the SKB twice.

XDP Implementation Support

An XDP program is executed by `tp4a_flush` on each frame in the packet array that is ready to be flushed. If the XDP program returns XDP_DROP, the packet will simply be dropped from the packet array. XDP_TX will send the packet out again and then remove the frame from the packet array, while XDP_PASS will leave it as is so that it will be sent to user space.

Note that XDP does currently only support packets up to PAGE_SIZE in size and if you enable XDP you will not be able to use packets that are larger than PAGE_SIZE with ZC.

Discussion: Unifying XDP and ZC Support

Both XDP and ZC require driver additions to work in both the RX and TX paths. Would it not be good if we only had to implement this once and get both XDP and ZC support? Here is a high-level discussion on how to achieve this with packet arrays, so if you implement ZC support you get XDP support for free. Note that this is a discussion topic and not part of the RFC nor the experimental evaluation.

In the previous section we showed how XDP support is integrated into the packet arrays when we have ZC on, but we also have to support XDP with packet arrays when ZC is not turned on. We believe this could be achieved by introducing a source/sink concept in the packet array. (In the current implementation, the source and sink are always a V4 queue.) The source that `tp4a_populate` will get packets/buffers from will be (when ZC is not used) the standard buffer allocator used by the driver. The sink used by `tp4a_flush` to send packets will then be the already present “fill in the SKB and send it up the stack” path. With something like this, we believe we could support both non-ZC and ZC with the same packet array code. One key challenge here will be to deliver the same performance (or close enough) with packet arrays in the non-ZC case as the original RX and TX code.

What to do then with the two sets of NDOs? The ndo_xdp_flush is used to kick the TX path to send the packets that got the action XDP_TX. Similarly, ndo_tp4_xmit is used to kick the TX path to send the packets in the V4 TX ring. Neither of them take the RTNL lock. As packets now reside in a single packet array independent if they are ZC and/or had XDP executed, we believe it should be possible just to have one single NDO that simply flushes the packet array with a `tp4a_flush`. ndo_tp4_zerocopy enables and disables ZC while ndo_xdp enables and disables XDP. For these it probably makes the most sense just to keep them separate as they pass down data that is quite different.

Experimental Methodology

We run on a dual socket system with two Broadwell E5-2660 @ 2.0 GHz with hyperthreading turned off. Each socket has 14 cores which gives a total of 28. The memory is DDR4 @ 1067 MT/s and the size of each DIMM is 8192MB and with 8 of those DIMMs in the system we have 64 GB of total memory. We run Linux version 4.14-rc4, the distribution we use is Ubuntu 16.04.2 LTS, and the compiler used is gcc version 5.4.0 20160609. The NIC is an Intel I40E 40Gbit/s networking card version 2.1.14-k with firmware version 5.05. Only a single interface is used on the card. The BIOS is from Intel and has version number GRRFCRB1.86B.0261.R01.1507240936. Turbo boost has been turned off as well as power save to provide more stable performance numbers. All the four types of HW prefetchers are turned on. The V4 and ZC patch set is the limited distribution RFC v2 but our goal is to update this paper with results from the first public RFC sometime after it is released. Packets are generated by commercial packet generator HW that is generating packets at full 40 Gbit/s line rate.

Benchmark	**Description**
rxdrop	RX only without packet data touch
txpush	TX only without packet data touch
l2fwd	RX + swaps MAC headers + TX
tcpdump	Captures packets. Uses libpcap

Table 1: The micro-benchmarks used in this paper.

The micro-benchmarks used in this study for the I40E NIC are shown in Table 1. The first four are part of the RFC while libpcap is version 1.9.0-PRE-GIT_2017_10_09 extended to support V4 and ZC with tcpdump version 4.10.0-PRE-GIT on top of it extended to be able to pick V3, V4 and/or ZC support from the commandline. For veth, we use l2fwd with a configurable amount of packets being ping-ponged between two processes. These are generated by one of the processes before the start of the benchmarking run. Each benchmark runs for 60 seconds and each application process executes on its own core with cpu affinity. One core for the I40E benchmarks and two for the veth benchmarks. The RX and TX interrupts are processed by another core by setting the interrupt mask to that core. All processes are run on the same socket.

Experimental Results

Table 2 shows the results for the I40E NIC with 64 byte packets. As can be seen, ZC is between 20x to 40x faster than V2 and V3 for the micro-benchmarks. This is a significant improvement that we believe should translate to a sizeable performance increase in real life workloads. An indication of this can be seen with tcpdump, as the performance increase with ZC is a good 20x compared to V2 and V3, though tcpdump only has an RX component. By comparing V2 and V3 with V4 in copy-mode, we focus more on just the user space interfaces. These results show more of a mixed bag for 64 byte packets. For RX, V4 is slightly better, but for the micro-benchmarks with a TX component, V3 is better. The reason for this is that the current implementation of the data path of V4 in copy mode takes one more lock than V3 and V2. In order to get to the same performance as V3 in these benchmarks, we need to optimize away this extra lock in some way. This is future work.

Benchmark	V2	V3	V4	V4 + ZC
rxdrop	0.67	0.73	0.74	33.7
txpush	0.98	0.98	0.91	19.6
l2fwd	0.66	0.71	0.67	15.5
tcpdump	-	0.74	0.74	14.1

Table 2: The results of the I40E NIC benchmark runs in Mpps for 64 byte packets.

But when we take a look at the results for larger packets (1500 bytes) in Table 3, V4 is always better than V2 and V3 even in copy mode. The reason for this is that V4 does not need to copy any packet data from the RX descriptor ring to the TX one as all packets are in the packet buffer. To send a packet that was received, only the id of the packet has to be written into the TX descriptor ring. With V2 and V3, the whole packet has to be copied. V4 in copy mode is between 10% to 15% faster than V2 and V3 in copy mode. For ZC, we reach 40 Gbit/s line rate for rxdrop and tcpdump but somewhat less for txpush and l2fwd.

Benchmark	V2	V3	V4	V4 + ZC
rxdrop	0.56	0.58	0.66	3.3
txpush	0.81	0.81	0.88	3.1
l2fwd	0.55	0.56	0.62	2.9
tcpdump	-	0.62	0.64	3.3

Table 3: The results of the I40E NIC benchmark runs in Mpps for 1500 byte packets.

The results for veth can be seen in Table 4 and here ZC offers a performance increase compared to V2 and V3 by up to 8 times. But with V4 in copy mode we get less than half the performance of V2 and V3, so what is happening here? By running perf on the system, we have seen that the two processes involved in the application are not doing much at all in V4. They are just spinning on the flag in the data structure to check if there is a new packet for it. And this kind of spinning when one of the entities has much less to do than the other, is detrimental to the performance of the ring structure we are using. By implementing some rudimentary, hackish back off, we have seen that the performance gets in line with or better than V2 and V3. It seems people have already discovered this and Michael Tsirkin has sent out a proposal to improve the ring structure [13] that we should include to improve the situation. There might be other problems lurking here also. More examination is needed.

Benchmark	V2	V3	V4	V4 + ZC
l2fwd-1	0.65	0.76	0.32	0.90
l2fwd-64	0.85	0.93	0.65	5.6

Table 4: The results of the veth benchmark runs in Mpps for 64 byte packets. l2fwd-1 refers to one packet being ping-ponged and l2fwd-64 is with 64 packets.

Overall for the micro-benchmarks and tcpdump, ZC can be up to 40 times faster than V2 and V3, which we think is a quite significant performance increase that should hopefully translate into good sized boost in real life workloads too. While V4 in copy mode is better than V2 and V3 in many scenarios, it does lag behind in performance in some circumstances. In the development of the code base, up until this paper, performance optimizations have not been a focus, but it is clear that we need some focused efforts in the data path of V4 in copy mode as well as making the ring structure more resilient to contention. The latter effort will likely also boost the performance of ZC even further.

Related Work

A high-performance packet IO comparison paper by Gallenmüller et al [4] compared PF_RING, DPDK, and netmap and came to the same conclusions as we outlined in the introduction. While DPDK and PF_RING can offer higher performance than netmap that uses a kernel module, the former requires application rewriting and can crash the whole system as it does not have a security model.

Future Work and Discussion

First and foremost is to send out an RFC for this on the mailing list that is hopefully followed by a proper patch set. The only thing we can promise is that after review on the mailing list, things will not look exactly like the code presented in this paper.

We strongly believe that XDP, with some future extensions, can be beneficial in the context of ZC by for example doing load balancing between processes when the HW cannot do this, or the programmer would like to have a more flexible way of distributing the load. It could potentially also be used to rewrite the descriptor format to something that is not V4. How about having a really fast data path for virtio-net in which an XDP program transforms the descriptors in the packet array to virtio-net descriptors? For this we also need the source/sink extension to packet arrays that has been discussed earlier.

We would also like to encourage companies with real applications to give V4 and ZC a try. The feedback from these porting efforts and evaluation are invaluable. We need to show that this brings value to real applications, not just micro-benchmarks and tcpdump. Equally important is that other NIC vendors make some PoC implementation of ZC in their drivers to see if the packet array concept holds over more than the two drivers we tried out in this paper.

Another feature that we would like to support is metadata handling. Today's NICs and other accelerators produce a lot of metadata that could be used to speed up processing of the packets. Some of this will be used by the kernel while other metadata will only be used by user space. In the future, we should implement support in V4 and ZC to be able to pipe this metadata up to user space in an opaque manner, and to use whatever metadata that makes sense in the kernel path of ZC.

Conclusions

This paper presented AF_PACKET V4 and PACKET_ZEROCOPY (ZC) designed to offer high-performance packet delivery to Linux user-space while still being HW agnostic and abiding by the security and isolation rules of Linux. The approach has been implemented as a continuation to the popular AF_PACKET socket interface with the addition of a new true zero-copy mode that can be activated. This mode requires some additions to the device driver, but we have made an effort to make this as simple as possible to help adoption. One goal of our implementation approach was to include XDP support in our implementation, so if you implement support for ZC, you get XDP support for free.

The performance evaluation shows that V4 with ZC outperforms V2 and V3 in all cases by up to 40x, which makes us hopeful that this increase in our micro-benchmarks will translate into a performance increase in real life applications. For V4 in copy-mode, the performance is more mixed which signifies a need to start performance optimization work in that area, something that has not been our focus yet. Overall, we think that the approach shows a very promising performance boost and that it can serve as a foundation to some exiting future work in conjunction with XDP.

Acknowledgments

We would really like to thank the reviewers of the limited distribution RFC for all their comments that have helped improve the interfaces and the code significantly: Alexei Starovoitov, Alexander Duyck, and Jesper Dangaard Brouer. The internal team at Intel that has been helping out reviewing code, writing tests, reviewing this paper and sanity checking our ideas: Rami Rosen, Jeff Shaw, Ferruh Yigit, and Qi Zhang, your participation has really helped. Thank you all for your time. Highly appreciated.

References

[1] Deri, L. 2004. Improving passive packet capture: Beyond device polling. In *International System Administration and Network Engineering Conference (SANE)*.

[2] DPDK - dataplane development kit. http://www.dpdk.org.

[3] ExaBlaze. ExaNIC. http://exablaze.com.

[4] Gallenmüller, S.; Emmerich, P.; Wohlfart, F.; Raumer, D.; and Carle, G. 2015. Comparison of frameworks for high-performance packet io. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, 29–38. IEEE Computer Society.

[5] I40E - intel 40gb/s ethernet network adapter. https://www.intel.com/content/www/us/en/products/network-io/ethernet/10-25-40-gigabit-adapters.html.

[6] Libpcap and tcpdump. http://www.tcpdump.org.

[7] Odintsov, P. Capturing packets in linux at a speed of millions of packets per second without using third party libraries. https://kukuruku.co/post/capturing-packets-in-linux-at-a-speed-of-millions-of-packets-per-second-without-using-third-party-libraries/.

[8] RDMA - remote direct memory access. http://www.rdmaconsortium.org.

[9] Rizzo, L. 2012. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 101–112. Boston, MA: USENIX Association.

[10] Russel, R. vringfd(). https://lwn.net/Articles/276856/.

[11] SolarFlare. OpenOnload. http://openonload.org.

[12] Tsirkin, M. packed ring layout proposal v2. https://lists.oasis-open.org/archives/virtio-dev/201702/msg00010.html.

[13] Tsirkin, M. ptr_ring: batch ring zeroing. https://lkml.org/lkml/2017/4/7/19.