Add XDP support on a NIC driver

Lorenzo Bianconi
Jesper Dangaard Brouer
Ilias Apalodimas

NetDev 0x14
August 2020

Target audience

- target audience is a kernel developer that wants to extend existing driver to support XDP

XDP technical requirements

Quick introduction to XDP technical requirements

Driver XDP RX-processing (NAPI) loop

Code needed in driver for supporting XDP is fairly simple

```
while (desc_in_rx_ring && budget_left--) {
  action = bpf_prog_run_xdp(xdp_prog, xdp_buff);
  /* helper bpf_redirect_map have set map (and index) via this_cpu_ptr */
  switch (action) {
    case XDP_PASS:     break;
    case XDP_TX:       res = driver_local_xmit_xdp_ring(adapter, xdp_buff); break;
    case XDP_REDIRECT: res = xdp_do_redirect(netdev, xdp_buff, xdp_prog); break;
    default:           bpf_warn_invalid_xdp_action(action); /* fall-through */
    case XDP_ABORTED:  trace_xdp_exception(netdev, xdp_prog, action); /* fall-through */
    case XDP_DROP:     page_pool_recycle_direct(pp, page); res = DRV_XDP_CONSUMED; break;
  } /* left out acting on res */
} /* End of NAPI-poll */
xdp_do_flush_map(); /* Bulk chosen by map, can store xdp_frame's for flushing */
driver_local_XDP_TX_flush();
```

Tricky part is changing driver memory model to be compatible with XDP

XDP requirements

- XDP frame in physical contiguous memory
  - BPF Direct-Access for validating correctness
  - No paged frames support, data cannot be split across pages
  - Read and Write access to the DMA buffers
  - Disable jumbo frames (packet < PAGE_SIZE) loading a BPF program
- XDP headroom for xdp_frame area
  - add push/pop header through bpf_xdp_adjust_head()
- Reserve tailroom for skb_shared_info and rely on build_skb() on XDP_PASS
- Cannot allocate page fragments to support it (e.g. through napi_alloc_skb())
- Rx buffers must be recycled to get high speed!

Register your XDP memory model

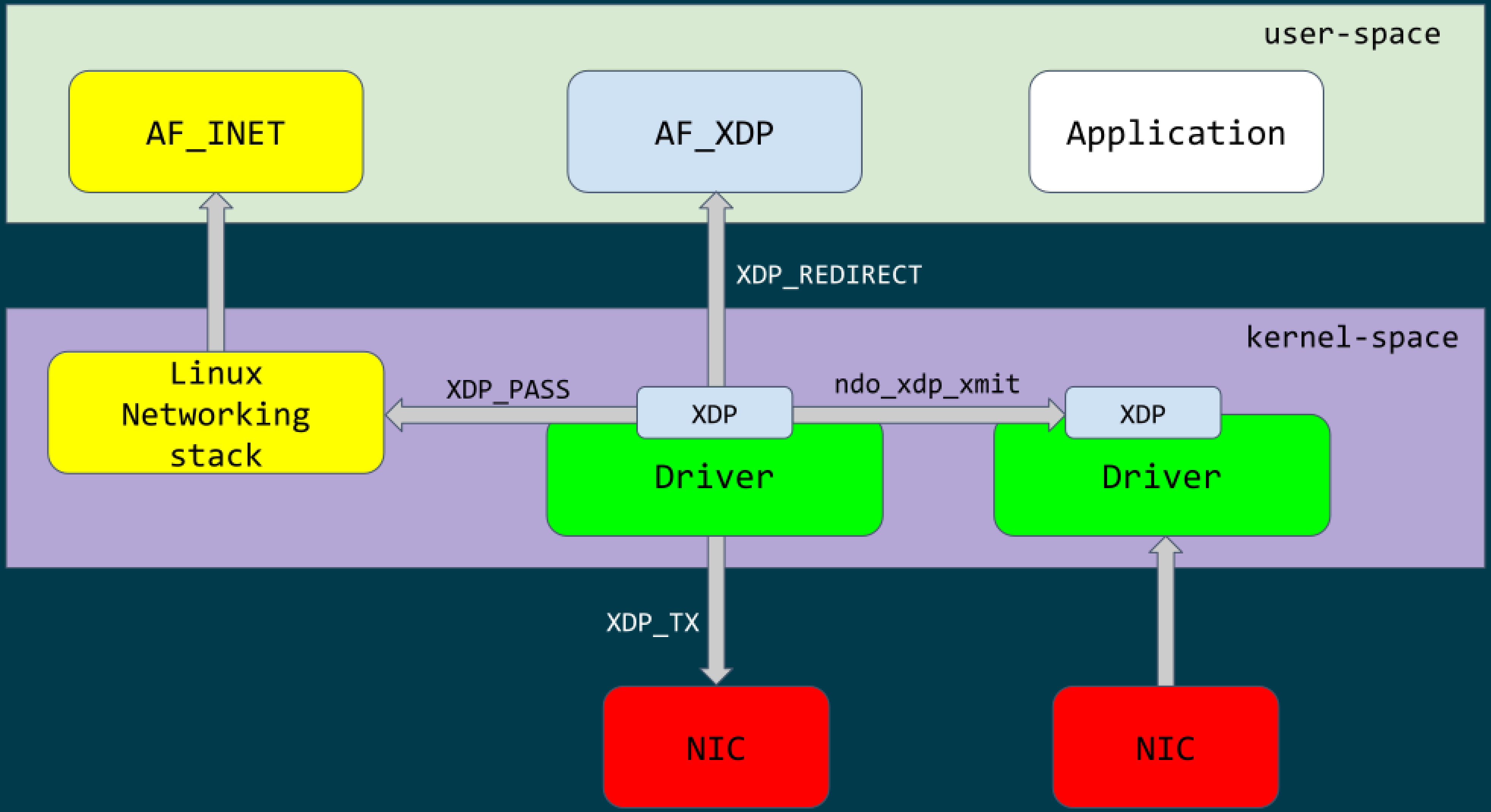
XDP force drivers to register which memory model they use

- See struct xdp_rxq_info and member “mem” struct xdp_mem_info
  - API see: `xdp_rxq_info_reg_mem_model()`

Advantage: Allows inventing new memory models for networking

- `MEM_TYPE_PAGE_SHARED` is the normal refcnt based model
- `MEM_TYPE_PAGE_POOL` is optimized for XDP (more on this later)
- `MEM_TYPE_XSK_BUFF_POOL` is to AF_XDP zero-copy into userspace
- Hope new models will be invented
  - e.g. imagine memory used by NIC belongs to GPU graphics card

XDP architecture



The page_pool allocator

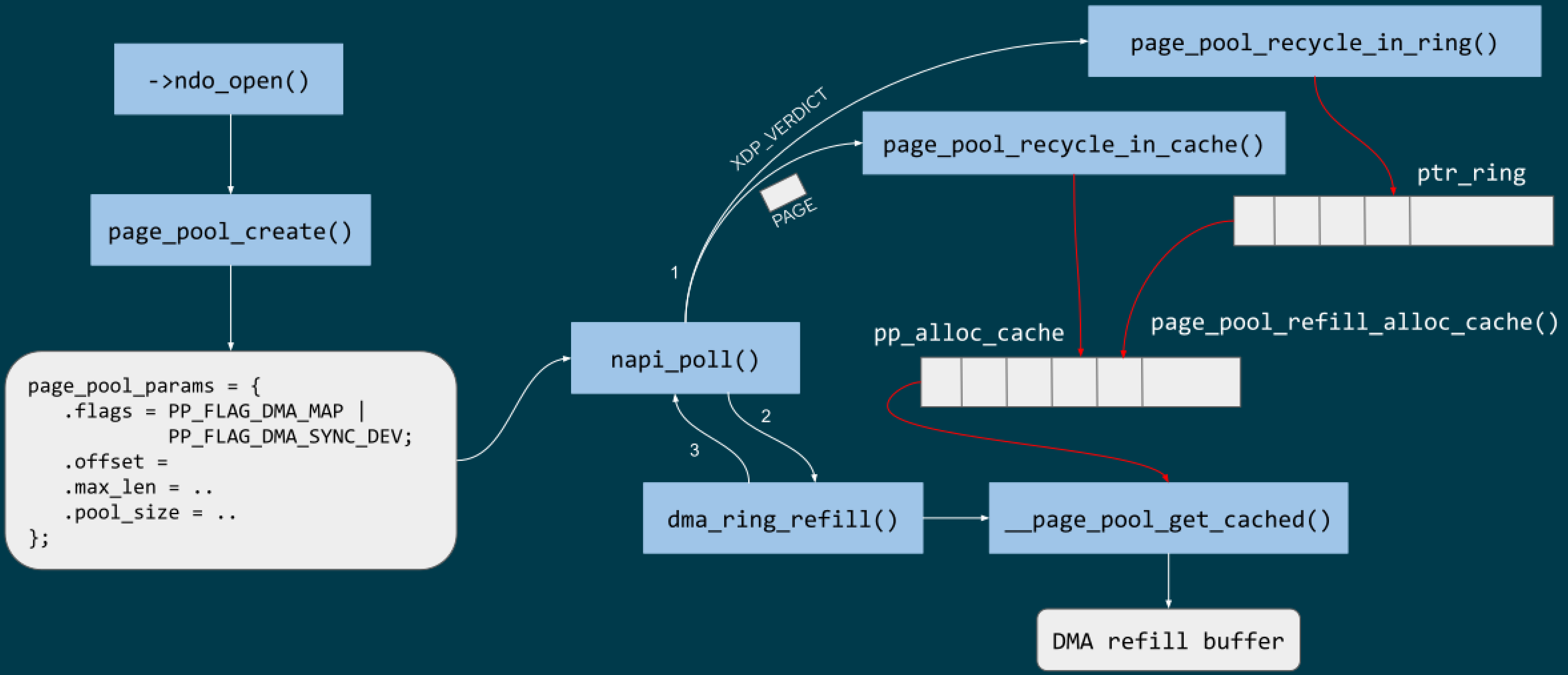
Helping drivers with API to ease transition to new memory model

- page_pool design principles
- page_pool architecture
- page_pool APIs
- code examples

page_pool: design principles

- optimized for XDP memory-model
  - ideally one page per frame model
  - supports split-page model, but the recycling is usually in-driver
- one page_pool for each hardware rx queue
  - run in NAPI context, no locking penalties
  - some hw will impose exceptions (i.e. currently cpsw)
- native buffer recycling for XDP
  - two caches available
    - in-softirq cache
    - ptr-ring cache
- API supports DMA mapping and syncing

page_pool: architecture



```
page_pool_params = {
   .flags = PP_FLAG_DMA_MAP |
            PP_FLAG_DMA_SYNC_DEV;
   .offset =
   .max_len = ..
   .pool_size = ..
};
```

page_pool: APIs

- Main APIs
  - page_pool_create(): create the pool object
  - page_pool_put_page(): recycle or unmap the page
  - page_pool_release_page(): unmap the page
  - page_pool_dev_alloc_pages(): get a new page from cache or alloc a new one
  - page_pool_get_dma_addr(): retrieve the stored DMA address
  - page_pool_get_dma_dir(): retrieve the stored DMA direction
  - page_pool_recycle_direct(): recycle the page immediately
- kernel documentation available @ Documentation/networking/page_pool.rst

Code examples (1/2)

- pool registration opening net_device

```
struct page_pool_params pp_params = { 0 };
struct xdp_rxq_info xdp_rxq;
int err;

pp_params.order = 0;
/* internal DMA mapping in page_pool */
pp_params.flags = PP_FLAG_DMA_MAP;
pp_params.pool_size = DESC_NUM;
pp_params.nid = NUMA_NO_NODE;
pp_params.dev = priv->dev;
pp_params.dma_dir = xdp_prog ? DMA_BIDIRECTIONAL : DMA_FROM_DEVICE;
page_pool = page_pool_create(&pp_params);

err = xdp_rxq_info_reg(&xdp_rxq, ndev, 0);
if (err)
    goto err_out;

err = xdp_rxq_info_reg_mem_model(&xdp_rxq, MEM_TYPE_PAGE_POOL, page_pool);
if (err)
    goto err_out;
```

Code examples (2/2)

- NAPI poller

```
dma_dir = page_pool_get_dma_dir(dring->page_pool);
...
while (done < budget) {
    if (some error)
        page_pool_recycle_direct(page_pool, page);
    if (packet_is_xdp) {
        if XDP_DROP:
            page_pool_recycle_direct(page_pool, page);
    } else (packet_is_skb) {
        page_pool_release_page(page_pool, page);
        new_page = page_pool_dev_alloc_pages(page_pool);
    }
}
```

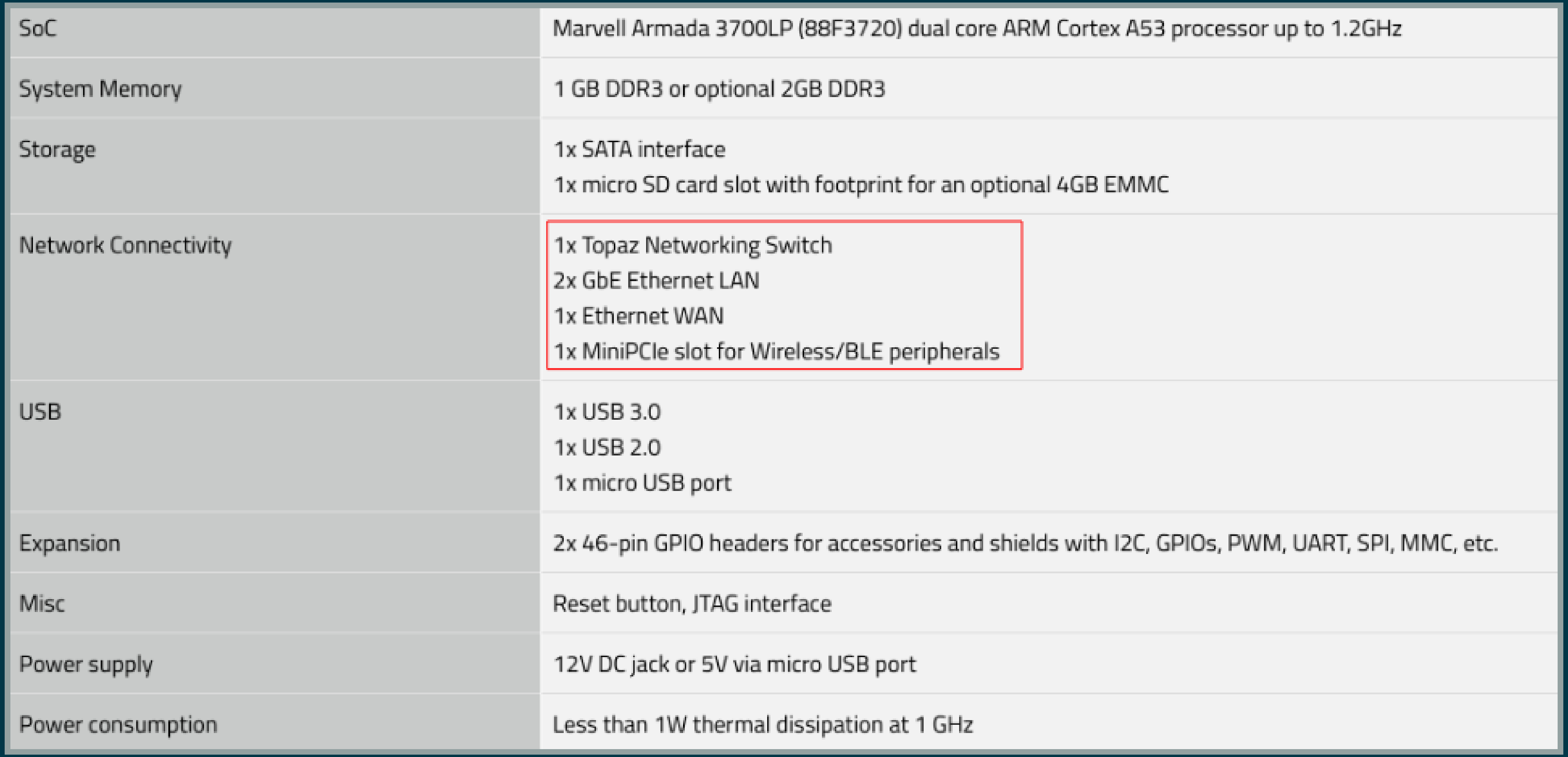
- module unloading

```
page_pool_put_page(page_pool, page, false);
xdp_rxq_info_unreg(&xdp_rxq);
```

Add XDP support on a NIC driver: mvneta

- page_pool lifecycle
  - create/destroy the pool
  - DMA ring refill
- XDP architecture
  - XDP main loop
  - XDP verdicts
  - XDP new features

Marvell ESPRESSObin - mvneta



SoC	Marvell Armada 3700LP (88F3720) dual core ARM Cortex A53 processor up to 1.2GHz
System Memory	1 GB DDR3 or optional 2GB DDR3
Storage	1x SATA interface 1x micro SD card slot with footprint for an optional 4GB EMMC
Network Connectivity	1x Topaz Networking Switch 2x GbE Ethernet LAN 1x Ethernet WAN 1x MiniPCIe slot for Wireless/BLE peripherals
USB	1x USB 3.0 1x USB 2.0 1x micro USB port
Expansion	2x 46-pin GPIO headers for accessories and shields with I2C, GPIOs, PWM, UART, SPI, MMC, etc.
Misc	Reset button, JTAG interface
Power supply	12V DC jack or 5V via micro USB port
Power consumption	Less than 1W thermal dissipation at 1 GHz

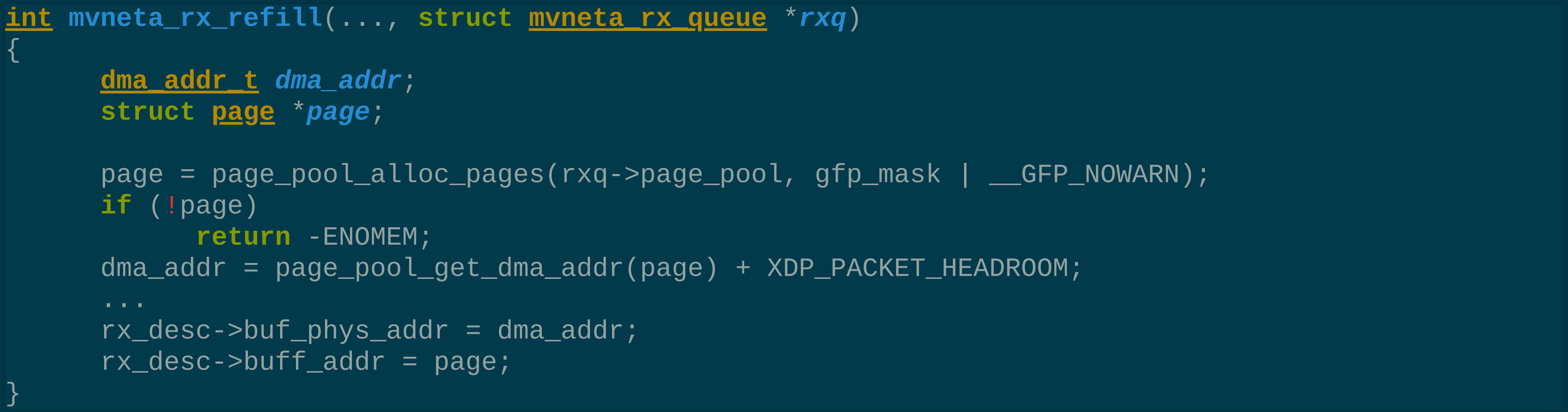
mvneta: page_pool lifecycle (1/3)

- the page_pool is usually associated to a hw rx queue
  - the page_pool is created opening or reconfiguring the net_device

```c
int mvneta_create_page_pool(..., struct mvneta_rx_queue *rxq, ...)
{
      struct page_pool_params pp_params = {
            .order = 0,
            .flags = PP_FLAG_DMA_MAP | PP_FLAG_DMA_SYNC_DEV,
            .pool_size = size,
            .nid = NUMA_NO_NODE,
            .dma_dir = xdp_prog ? DMA_BIDIRECTIONAL : DMA_FROM_DEVICE,
            .offset = XDP_PACKET_HEADROOM,
            .max_len = PAGE_SIZE - SKB_DATA_ALIGN(sizeof(struct skb_shared_info) +
                                                  XDP_PACKET_HEADROOM),
      };
      rxq->page_pool = page_pool_create(&pp_params);
      ...
      xdp_rxq_info_reg(&rxq->xdp_rxq, ..., rxq->id);
      ...
      xdp_rxq_info_reg_mem_model(&rxq->xdp_rxq, MEM_TYPE_PAGE_POOL, rxq->page_pool);
}
```

mvneta: page_pool lifecycle (2/3)

- mvneta_rx_refill() relies on page_pool APIs to refill the hw DMA rx ring
  - get pages from page_pool caches and avoid the page allocator
  - the page is dma_sync_*_for_device() relying on page_pool APIs in page_pool_put_page()



```
int mvneta_rx_refill(..., struct mvneta_rx_queue *rxq)
{
      dma_addr_t dma_addr;
      struct page *page;

      page = page_pool_alloc_pages(rxq->page_pool, gfp_mask | __GFP_NOWARN);
      if (!page)
            return -ENOMEM;
      dma_addr = page_pool_get_dma_addr(page) + XDP_PACKET_HEADROOM;
      ...
      rx_desc->buf_phys_addr = dma_addr;
      rx_desc->buff_addr = page;
}
```

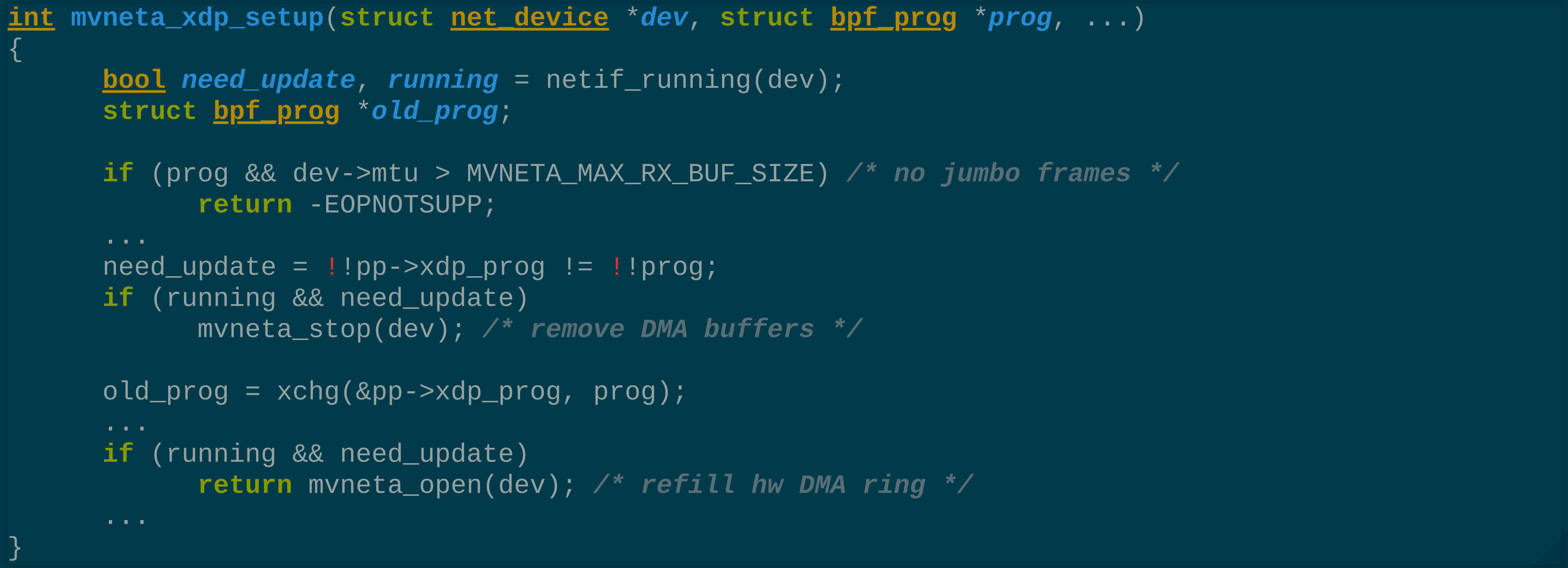
mvneta: page_pool lifecycle (3/3)

- pages allocated to the NIC are released closing the net_device
    - pages are released to the page_pool
    - the page_pool is destroyed whenever there are no inflight pages

```
void mvneta_rxq_drop_pkts(..., struct mvneta_rx_queue *rxq)
{
      for (i = 0; i < rxq->size; i++) {
            ...
                page_pool_put_full_page(rxq->page_pool, page, false);
        }
        if (xdp_rxq_info_is_reg(&rxq->xdp_rxq))
                xdp_rxq_info_unreg(&rxq->xdp_rxq);
        page_pool_destroy(rxq->page_pool);
      ...
}
```

mvneta: loading an eBPF program

- mvneta_xdp_setup() is used to load or remove an eBPF program from the NIC
  - it reconfigures the DMA buffers - XDP memory model

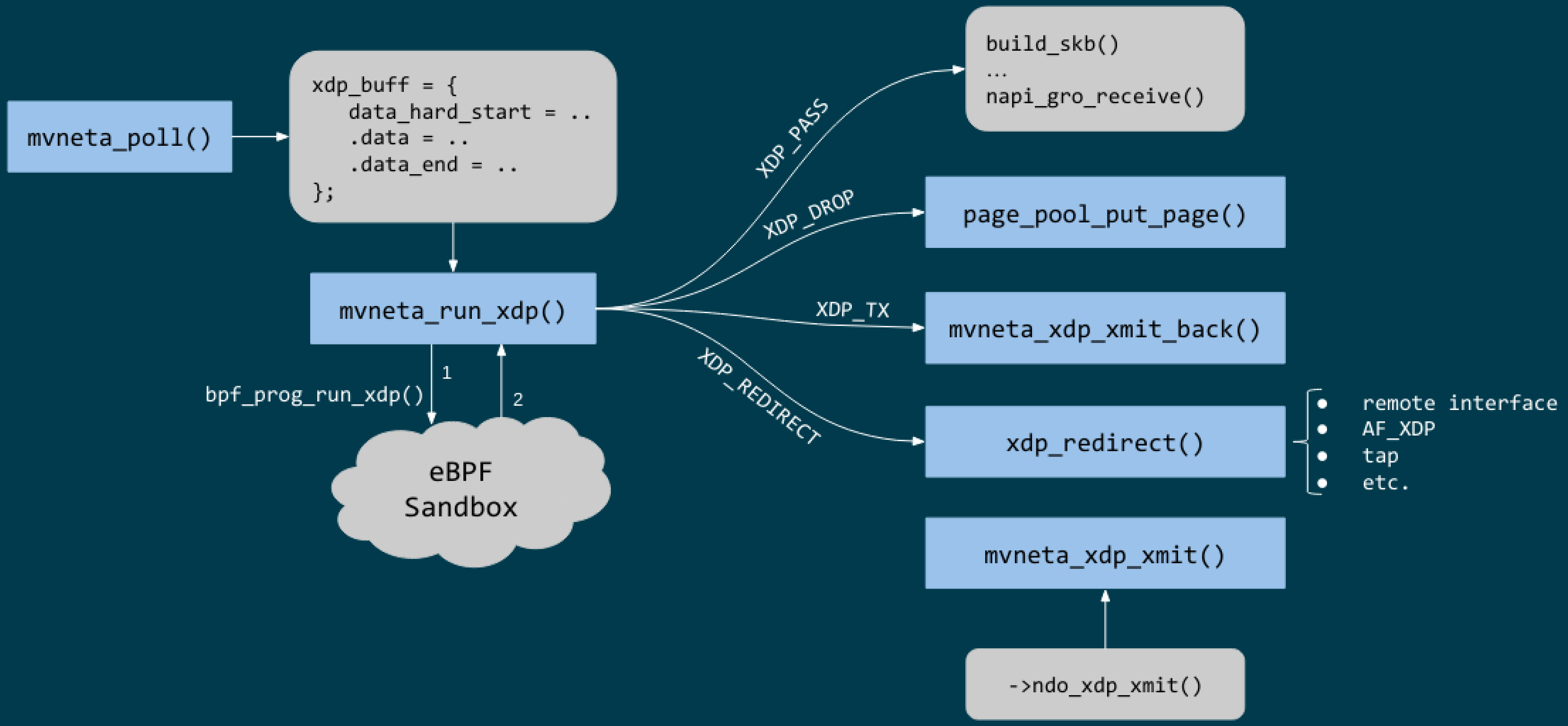


```
int mvneta_xdp_setup(struct net_device *dev, struct bpf_prog *prog, ...)
{
      bool need_update, running = netif_running(dev);
      struct bpf_prog *old_prog;

      if (prog && dev->mtu > MVNETA_MAX_RX_BUF_SIZE) /* no jumbo frames */
            return -EOPNOTSUPP;
      ...
      need_update = !!pp->xdp_prog != !!prog;
      if (running && need_update)
            mvneta_stop(dev); /* remove DMA buffers */

      old_prog = xchg(&pp->xdp_prog, prog);
      ...
      if (running && need_update)
            return mvneta_open(dev); /* refill hw DMA ring */
      ...
}
```

mvneta XDP architecture



mvneta XDP: main loop - mvneta_rx_swbm()

```
struct bpf_prog *xdp_prog = READ_ONCE(pp->xdp_prog);
struct xdp_buff xdp;
for (i = 0, i < budget; i++) {
  ...
  if (rx_desc->status & MVNETA_RXD_FIRST_DESC) { /* XDP is single buffer */
      enum dma_data_direction dma_dir = page_pool_get_dma_dir(rxq->page_pool);
      dma_sync_single_for_cpu(..., rx_desc->buf_phys_addr, rx_desc->data_size,
                              dma_dir); /* invalid CPU caches */
      ...
      xdp->data_hard_start = rx_desc->buff_addr; /* init xdp_buff */
      xdp->data = rx_desc->buff_addr + XDP_PACKET_HEADROOM + MVNETA_MH_SIZE;
      xdp->data_end = xdp->data + rx_desc->data_size;
      ...
      ret = mvneta_run_xdp(.., xdp_prog, xdp, ...);
      if (ret != MVNETA_XDP_PASS)
            goto refill;
      /* send the packet to the networking stack */
      ...
refill:
      mvneta_rx_refill(.., rxq);
  }
}
```

mvneta XDP: main loop - mvneta_run_xdp()

```
int mvneta_run_xdp(struct bpf_prog *prog, struct xdp_buff *xdp, ...)
{
      int len = xdp->data_end - xdp->data_hard_start - XDP_PACKET_HEADROOM;
      int act = bpf_prog_run_xdp(prog, xdp);
      ...
      switch (act) {
      case XDP_PASS:
            return MVNETA_XDP_PASS;
      case XDP_REDIRECT:
            ...
            xdp_do_redirect(..., xdp, prog);
            return MVNETA_XDP_REDIR;
      case XDP_TX:
            mvneta_xdp_xmit_back(..., xdp);
            return MVNETA_XDP_TX;
      case XDP_ABORTED:
            trace_xdp_exception(..., prog, act);
      /* fall through */
      case XDP_DROP:
            page_pool_put_page(rxq->page_pool, virt_to_head_page(xdp->data), len, true);
            return MVNETA_XDP_DROPPED;
      }
}
```

mvneta XDP: XDP_DROP (1/3)

- the driver is running in NAPI context and page refcount is 1
  - page_pool_put_page() will recycle the page in in-softirq page_pool cache
- the page is synced for device using optional size in page_pool_dma_sync_for_device()

```
int mvneta_run_xdp(struct bpf_prog *prog, struct xdp_buff *xdp, ...)
{
	int len = xdp->data_end - xdp->data_hard_start - rx_offset;
	int act = bpf_prog_run_xdp(prog, xdp);
	...
	switch (act) {
	...
	case XDP_DROP:
		page_pool_put_page(rxq->page_pool, virt_to_head_page(xdp->data), len, true);
		stats->xdp_drop++;
		return MVNETA_XDP_DROPPED;
	}
}
```

mvneta XDP: XDP_DROP (2/3)

mvneta XDP: XDP_DROP (3/3)

- DDoS performance:
  - packet size: 64B
  - DSA: disabled
- XDP_DROP:

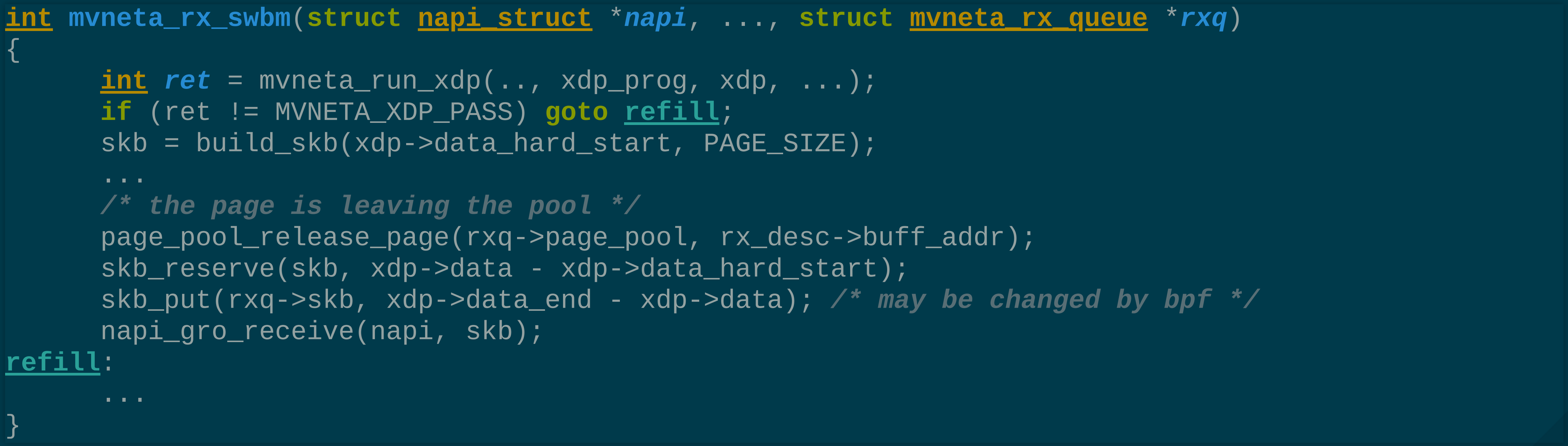
```
$ip link set dev eth0 xdp obj xdp-drop.o
585273 pkt/s
585159 pkt/s
585050 pkt/s
```

- tc drop:

```
$tc qdisc add dev eth0 clsact ; tc filter add dev eth0 ingress matchall action gact drop
185237 pkt/s
185557 pkt/s
185670 pkt/s
```

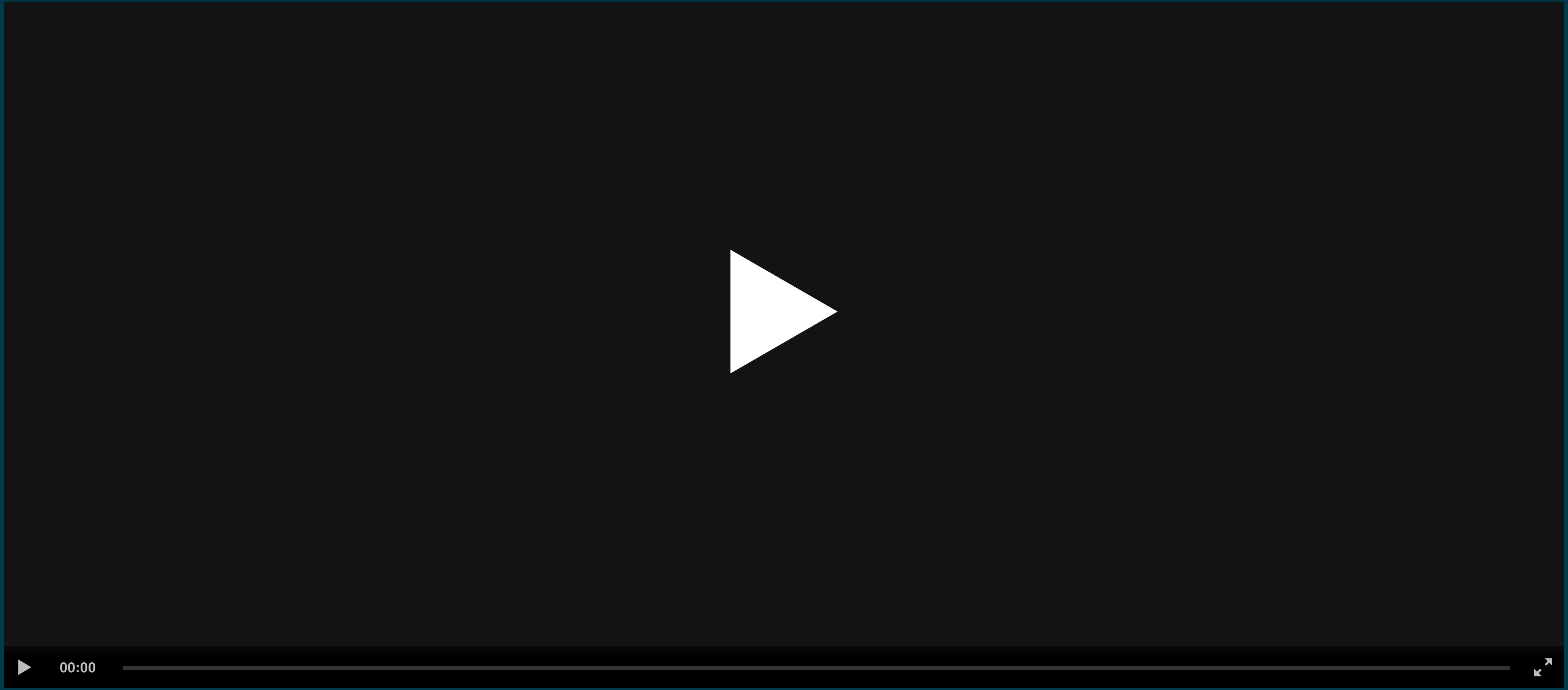
mvneta XDP: XDP_PASS (1/2)

- XDP_PASS to forward the frame to the networking stack
- mvneta_swbm_rx_frame() relies on build_skb() for zero-copy
  - get rid of original copy-break approach
  - take into account skb_shared_info in the buffer tailroom



```
int mvneta_rx_swbm(struct napi_struct *napi, ..., struct mvneta_rx_queue *rxq)
{
      int ret = mvneta_run_xdp(.., xdp_prog, xdp, ...);
      if (ret != MVNETA_XDP_PASS) goto refill;
      skb = build_skb(xdp->data_hard_start, PAGE_SIZE);
      ...
      /* the page is leaving the pool */
      page_pool_release_page(rxq->page_pool, rx_desc->buff_addr);
      skb_reserve(skb, xdp->data - xdp->data_hard_start);
      skb_put(rxq->skb, xdp->data_end - xdp->data); /* may be changed by bpf */
      napi_gro_receive(napi, skb);
refill:
      ...
}
```

mvneta XDP: XDP_PASS (2/2)



mvneta XDP: XDP_TX (1/3)

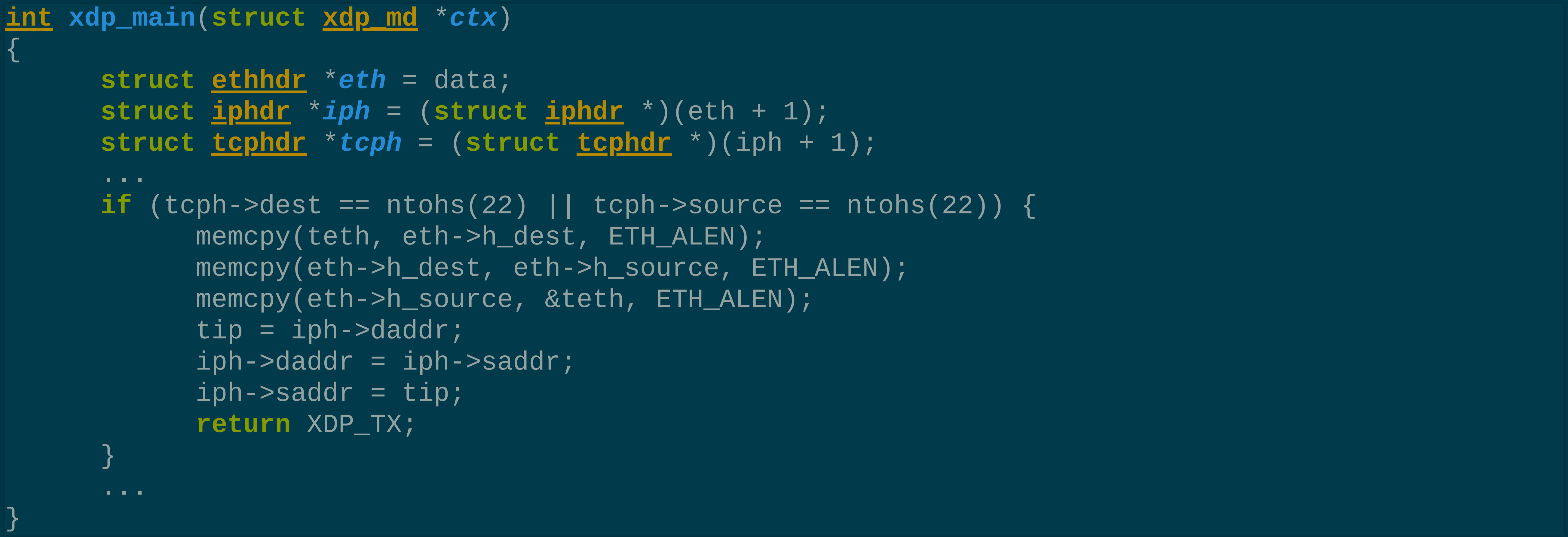
- XDP_TX = frame transmitted back out interface where packet was received
  - no need to DMA remap the page, only to DMA-sync/flush CPU caches

```
int mvneta_xdp_xmit_back(..., struct xdp_buff *xdp)
{
        struct xdp_frame *xdpf = convert_to_xdp_frame(xdp);
        struct page *page = virt_to_page(xdpf->data);
        dma_addr_t dma_addr;

        dma_addr = page_pool_get_dma_addr(page) +
                   sizeof(*xdpf) + xdpf->headroom;
        dma_sync_single_for_device(..., dma_addr, xdpf->len,
                                   DMA_BIDIRECTIONAL);
        tx_desc->buf_phys_addr = dma_addr;
        tx_desc->data_size = xdpf->len;
        /* update DMA tx registers */
        ...
}
```

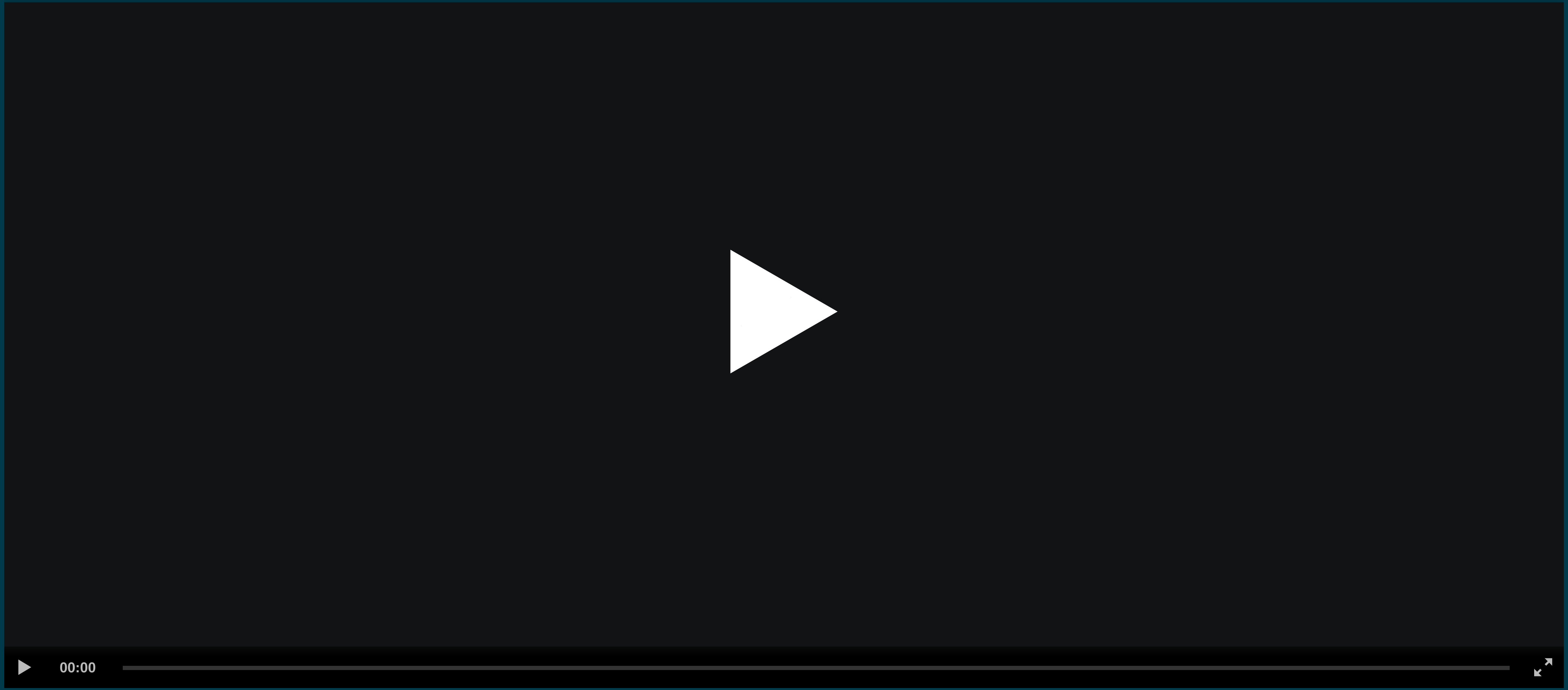
mvneta XDP: XDP_TX (2/3) - ssh-mirror.c

- swap ethernet and ip addresses for ssh connections
  - by Matteo Croce <mcroce@microsoft.com>



```
int xdp_main(struct xdp_md *ctx)
{
        struct ethhdr *eth = data;
        struct iphdr *iph = (struct iphdr *)(eth + 1);
        struct tcphdr *tcph = (struct tcphdr *)(iph + 1);
        ...
        if (tcph->dest == ntohs(22) || tcph->source == ntohs(22)) {
                memcpy(teth, eth->h_dest, ETH_ALEN);
                memcpy(eth->h_dest, eth->h_source, ETH_ALEN);
                memcpy(eth->h_source, &teth, ETH_ALEN);
                tip = iph->daddr;
                iph->daddr = iph->saddr;
                iph->saddr = tip;
                return XDP_TX;
        }
        ...
}
```

mvneta XDP: XDP_TX (3/3)



mvneta XDP: XDP_REDIRECT (1/3)

- xdp_do_redirect() forwards the frame to:
  - remote interface - ndo_xdp_xmit()
  - remote cpu - cpu_map
  - AF_XDP socket

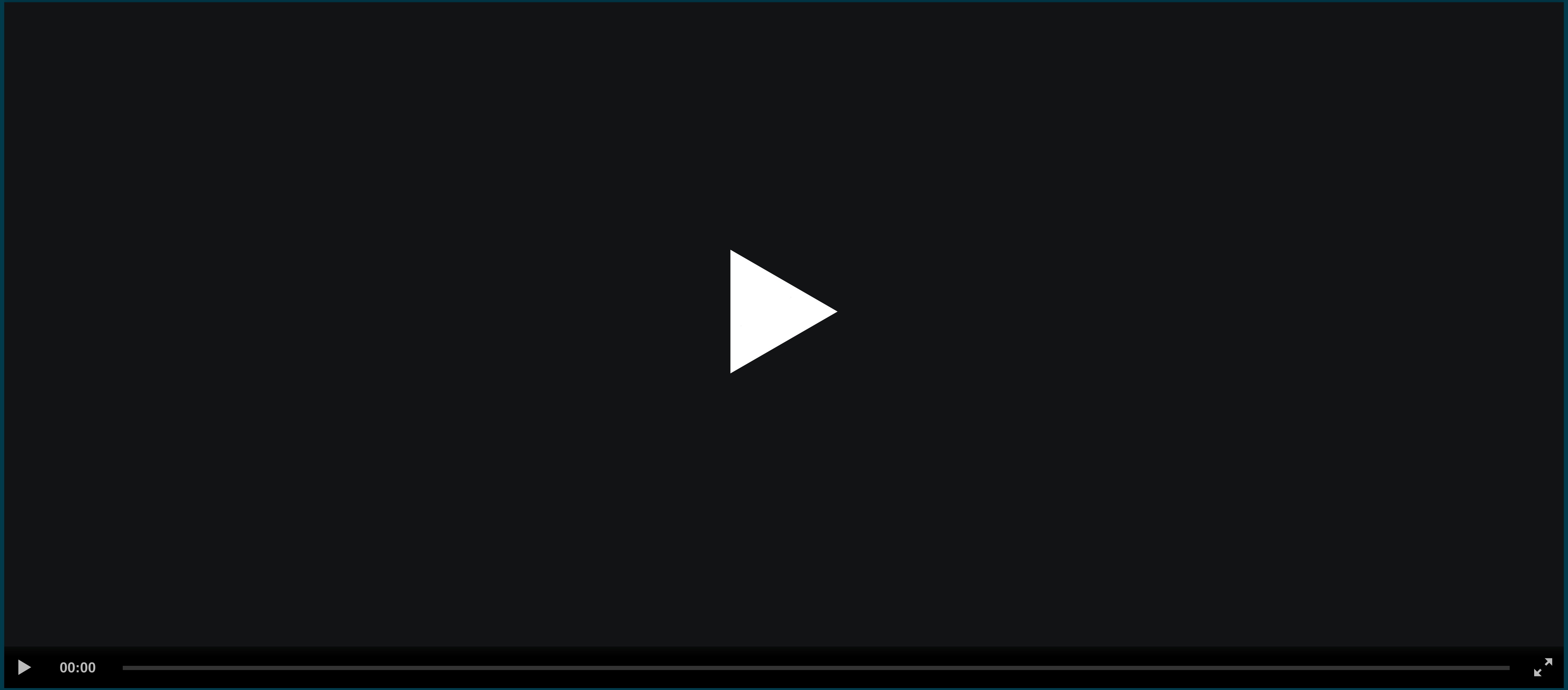
```
int mvneta_run_xdp(struct bpf_prog *prog, struct xdp_buff *xdp, ...)
{
        int act = bpf_prog_run_xdp(prog, xdp);
        ...
        switch (act) {
        ...
        case XDP_REDIRECT:
                xdp_do_redirect(..., xdp, prog);
                ...
                stats->xdp_redirect++;
                return MVNETA_XDP_REDIR;
        }
}
```

mvneta XDP: XDP_REDIRECT (2/3)

- mvneta_xdp_xmit() - mvneta is the destination of XDP_REDIRECT
  - the page is mapped to DMA hw tx ring

```
int mvneta_xdp_xmit(struct net_device *dev, int num_frame,
                    struct xdp_frame **frames, u32 flags)
{
      ...
      for (i = 0; i < num_frame; i++) {
            struct xdp_frame *xdpf = frames[i];
            dma_addr_t dma_addr = dma_map_single(.., xdpf->data,
                                                 xdpf->len, DMA_TO_DEVICE);
            ...
            tx_desc->buf_phys_addr = dma_addr;
            tx_desc->data_size = xdpf->len;
      }
      if (flags & XDP_XMIT_FLUSH) {
            /* update DMA tx registers */
      }
      ...
}
```

mvneta XDP: XDP_REDIRECT (3/3)



mvneta sw RPS: CPUMAP (1/4)

- ESPRESSObin does not support hw Receive Packet Steering (RPS)
  - all the packets are received on cpu0
- With CPUMAPs we can move the processing on a remote cpu
  - CPUMAPs are used to build the skb and forward it to legacy stack
- We extended CPUMAPs to execute an eBPF program on a remote cpu
  - we can now attach an eBPF program on CPUMAP entries
- XDP_REDIRECT and CPUMAP: sw RPS
  - on cpu0 mvneta performs XDP_REDIRECT on a CPUMAP entry
  - on the remote cpu we run an eBPF program
    - e.g. XDP_REDIRECT to another device

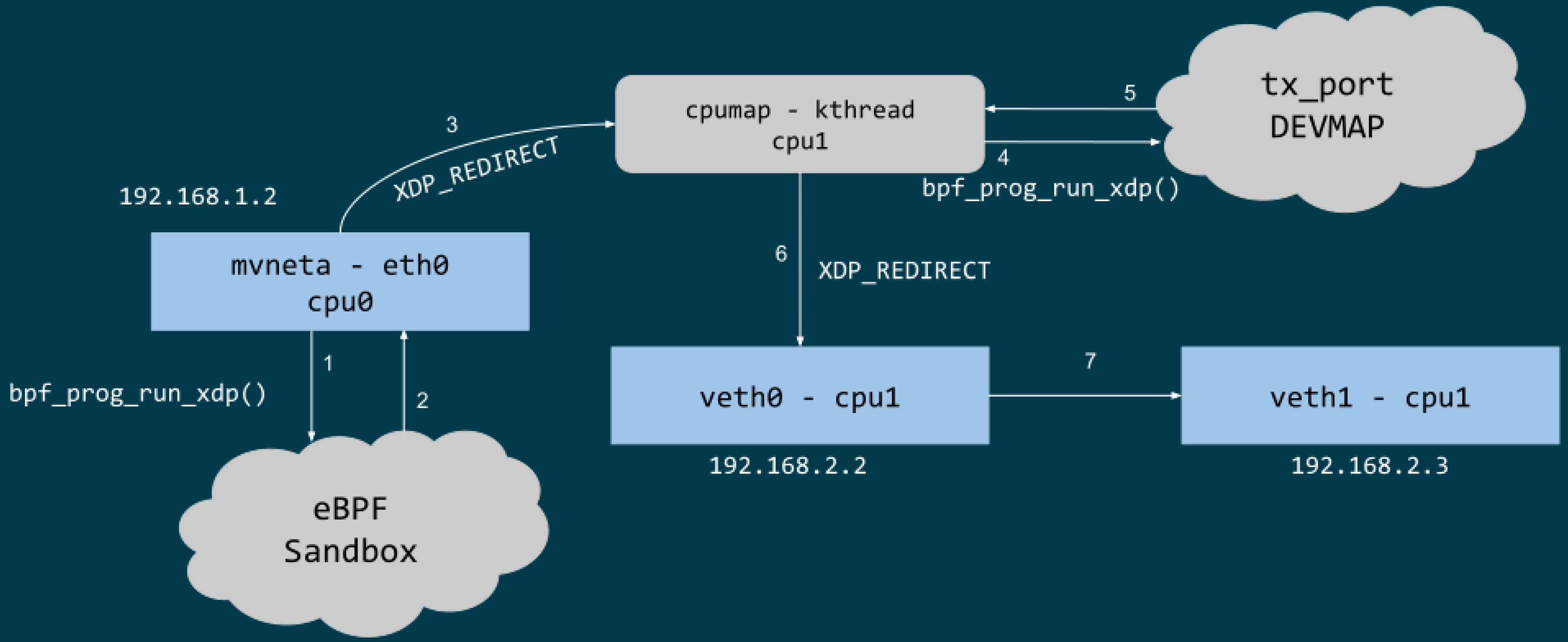
mvneta sw RPS: CPUMAP (2/4)

- kthread bound to run on the remote CPU

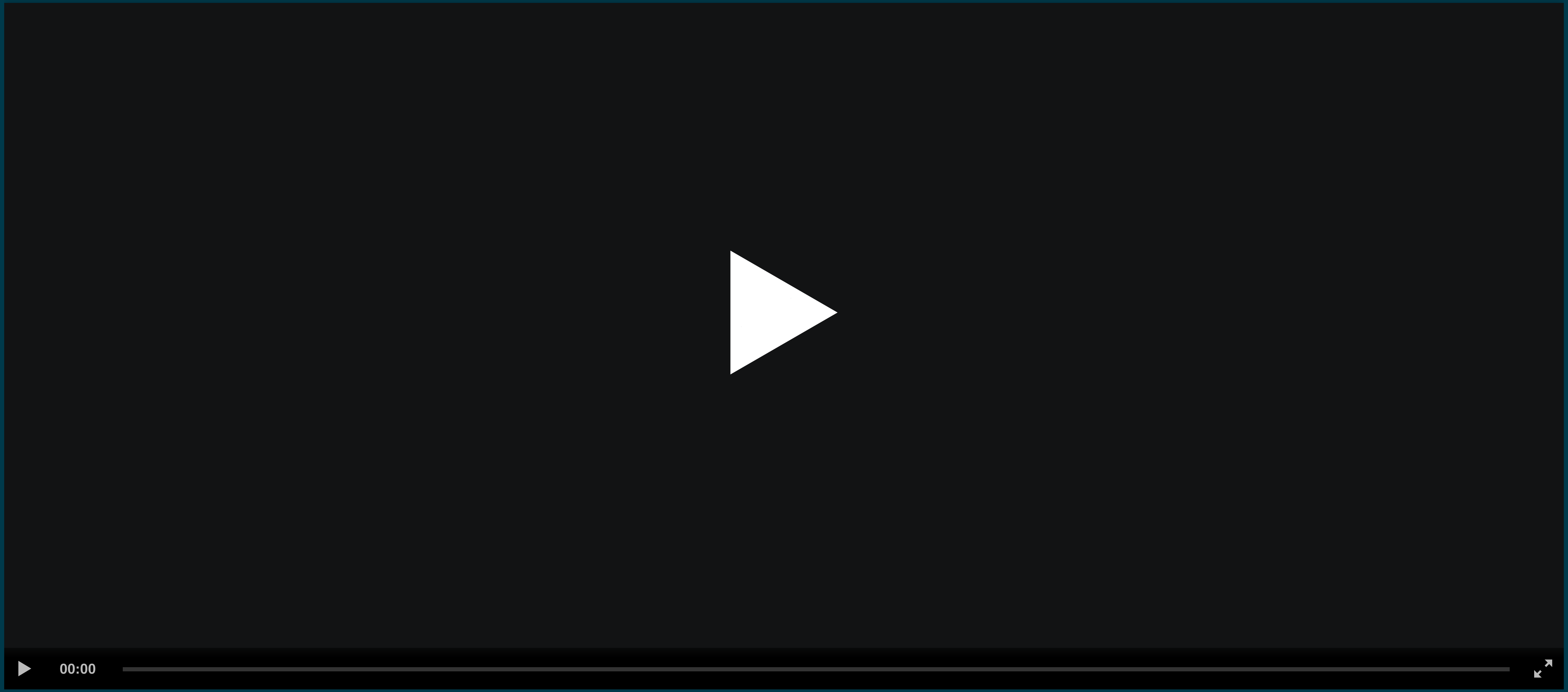
```
int cpu_map_kthread_run(void *data)
{
       n = __ptr_ring_consume_batched(); /* consume redirected frames */
       ...
       for (i = 0; i < n; i++) {
            ...
            act = bpf_prog_run_xdp(rcpu->prog, &xdp);
            switch (act) {
            case XDP_PASS:
                  skb = build_skb_around();
                  ...
            case XDP_REDIRECT:
                  ...
            case XDP_DROP:
                  xdp_return_frame(xdpf);
                  ...
            }
      }

}
```

mvneta sw RPS: CPUMAP (3/4)



mvneta sw RPS: CPUMAP (4/4)



mvneta XDP stats

- proper stats accounting is essential for XDP success
  - allow the sys-admin to understand what is going on
- netdev stats:
  - always increment rx packets counters even for XDP_DROP
- fine grained stats through ethtool

```
root@espresso-bin:~# ethtool -S eth0 | grep xdp
     rx_xdp_redirect: 0
     rx_xdp_pass: 0
     rx_xdp_drop: 0
     rx_xdp_tx: 0
     rx_xdp_tx_errors: 0
     tx_xdp_xmit: 0
     tx_xdp_xmit_errors: 0
```

- even stats name matters!!

XDP multi-buffers

Work-in-progress

- Adding XDP multi-buffers support
- Design document under XDP-project

Joint work between Amazon and Red Hat

- Future credit to:
  - Samih Jubran, Lorenzo Bianconi, Eelco Chaudron

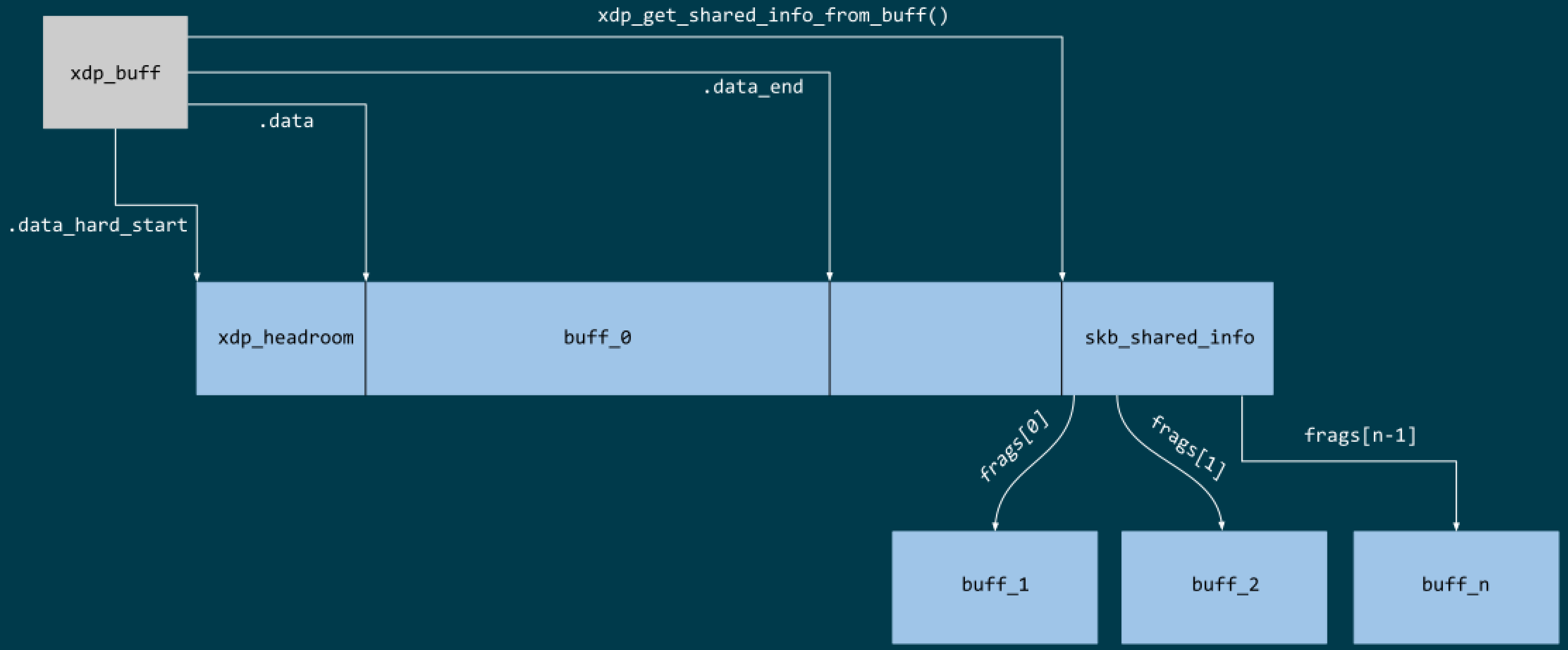
Multi-buffers support for XDP (1/4)

- XDP multi-buffer use cases
  - Jumbo frames, TSO, LRO, Packet header split
  - Handling GRO SKBs in veth/cpumap/generic-XDP
- How to satisfy eBPF Direct-Access (DA) design?
- Proposal: eBPF can access only to the first packet buffer
  - Storage space for multi-buffer segments references
    - (like skb_shared_info) at the end of the first segment (in tailroom)
  - This “xdp_shared_info” area provide metadata
    - for-each buffer: page-pointer, offset, length (see skb_frag_t frags[])
    - also metadata for e.g. number of segments, full packet length
  - Only need single “mb” (multi_buffer) bit indicator in xdp_buff and xdp_md

Multi-buffers support for XDP (2/4)

- Modify drivers rx NAPI loop
  - Process all RX descriptor segments building xdp_buff
    - `mvneta_swbm_rx_frame()`
    - `mvneta_swbm_add_rx_fragment()`
  - Run the eBPF program when all descriptors are processed
  - Change XDP_TX and ndo_xdp_xmit to map non-linear buffers
    - `mvneta_xdp_submit_frame()`
  - Remove MTU check loading the eBPF program
    - `mvneta_xdp_setup()`

Multi-buffers support for XDP (3/4)



(mvneta) Multi-buffers support for XDP (4/4)

```
void mvneta_swbm_add_rx_fragment(struct xdp_buff *xdp, ...)
{
        struct skb_shared_info *sinfo = xdp_get_shared_info_from_buff(xdp);
        ...
        if (data_len > 0 && sinfo->nr_frags < MAX_SKB_FRAGS) {
                skb_frag_t *frag = &sinfo->frags[sinfo->nr_frags++];
                skb_frag_off_set(frag, offset);
                skb_frag_size_set(frag, data_len);
                __skb_frag_set_page(frag, page);
        }
}
struct sk_buff *mvneta_swbm_build_skb(struct xdp_buff *xdp, ..)
{
        struct skb_shared_info *sinfo = xdp_get_shared_info_from_buff(xdp);
        ...
        skb = build_skb(xdp->data_hard_start, PAGE_SIZE);
        memcpy(frags, sinfo->frags, sizeof(skb_frag_t) * num_frags);
        for (i = 0; i < num_frags; i++) {
                skb_add_rx_frag(skb, skb_shinfo(skb)->nr_frags, page,
                                skb_frag_off(frag), skb_frag_size(frag), PAGE_SIZE);
                page_pool_release_page(..., page);
        }
}
```

How to test a XDP driver

- XDP_PASS
- XDP_DROP
- XDP_TX
- XDP_REDIRECT
- ndo_xdp_xmit

test a XDP driver (1/4)

- XDP_PASS:
  - load a program that returns XDP_PASS on the host
    - verify the packets are delivered to the networking stack
- XDP_DROP:
  - load a program that returns XDP_DROP on the host
    - verify traffic is dropped

```
make M=samples/bpf -j24
sudo ./samples/bpf/xdp1 eth0
proto 17:      324874 pkt/s
proto 17:      324557 pkt/s
proto 17:      324650 pkt/s

sudo ./pktgen_sample02_multiqueue.sh -i enp2s0 -d 192.168.200.1 -s 64 \
      -m e0:d5:5e:65:ac:83 -t4 -n0
```

test a XDP driver: XDP_TX (2/4)

- load a program that returns XDP_TX on the host

```
make M=samples/bpf -j24
sudo ./samples/bpf/xdp2 eth0
proto 17:      55231 pkt/s
proto 17:      55971 pkt/s
proto 17:      55617 pkt/s
proto 17:      55103 pkt/s
```

- send a specific amount of packets to the host and capture the re-injected traffic with wireshark/tcpdump

```
sudo tcpdump -ni enp2s0 -s0 -w test.pcap
for i in {1..1500000}; do echo "This is my data" > /dev/udp/192.168.200.1/3000; done
```

- open the trace and verify packets are correctly received (1500000 Rx packets)

test a XDP driver: XDP_REDIRECT (3/4)

- redirect packets to an AF_XDP socket

```
make M=samples/bpf -j24
sudo ./samples/bpf/xdpsock -i eth0
sock0@eth0:0 rxdrop
                pps         pkts        1.00
rx              324,596     869,646
tx              0           0
sock0@eth0:0 rxdrop
                pps         pkts        1.00
rx              324,235     1,194,260
tx              0           0
```

- start sending traffic to that interface

```
sudo ./pktgen_sample02_multiqueue.sh -i enp2s0 -d 192.168.200.1 -s 64 \
      -m e0:d5:5e:65:ac:83 -t4 -n0
```

test XDP: ndo_xdp_xmit (4/4)

- create a veth pair and move one peer to a “remote” namespace

```
ip netns add remote
ip link add v0 type veth peer name v1 netns remote
```

- run xdp_redirect sample from kernel tree to redirect traffic from v0 to eth0
  - start sending traffic from v1

```
make M=samples/bpf -j24
sudo ./samples/bpf/xdp_redirect v0 eth0
```

- start injecting traffic into v1
- check outgoing traffic from eth0 with wireshark/tcpdump

Q&A:

- https://github.com/xdp-project
- https://xdp-project.net