

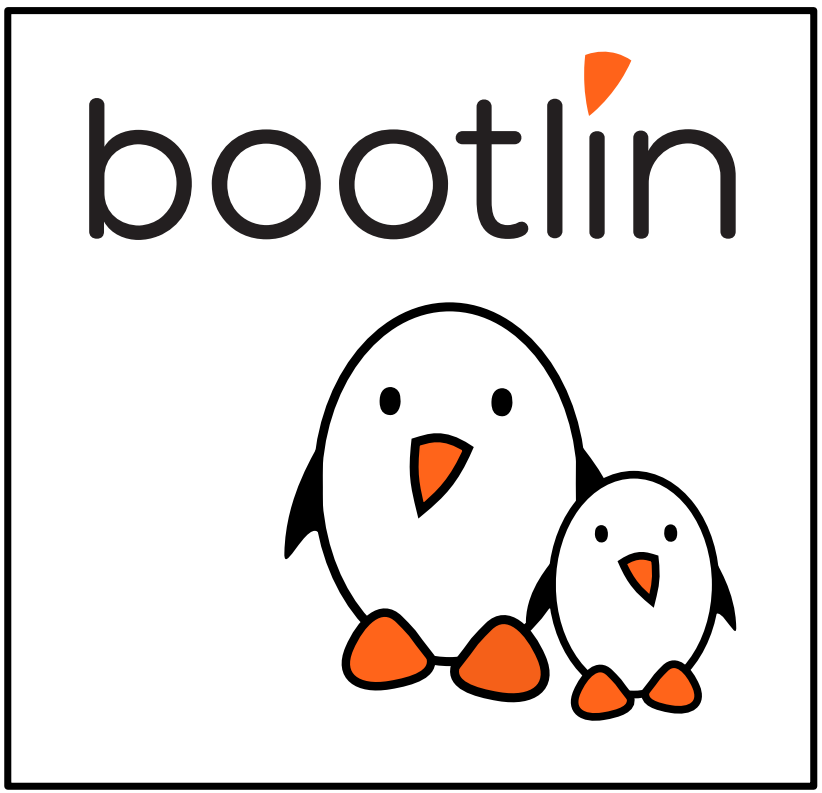
XDP & XSK support from a Linux driver standpoint

Théo Lebrun

theo.lebrun@bootlin.com

ELCE 2026

© Copyright 2004-2026, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!



Introduction





Théo Lebrun

- Embedded Linux engineer & trainer
- Joined Bootlin in 2022, following an internship
- Linux kernel driver work
  - SoC upstreaming of the Mobileye EyeQ MIPS family & maintainer
  - Suspend-to-RAM support for TI Jacinto automotive SoCs
  - Linux driver course trainer
  - Mobileye EyeQ SoCs and **MACB maintainer**
- & more
  - PipeWire ecosystem experience; audio course trainer
  - Current maintainer of Elixir (not the language)
  - Build system work on Buildroot & Yocto
- Living in *Lyon*, not north nor south of France
- I like running
- theo.lebrun@bootlin.com



MACB/GEM Ethernet controller

- Our case study for today
- Embedded Ethernet controller **Cadence IP**
- Platform and PCI driver in drivers/net/ethernet/cadence/
- The IP evolved over time (and it shows in the driver)
  - (Old) RM9200: separate `net_device_ops`, 9 Rx descriptors, 1 Tx slot (or 2?)
  - MACB: 10/100M, single Tx/Rx queue
  - GEM: 100M~10G, multi Tx/Rx queues



MACB/GEM Ethernet controller

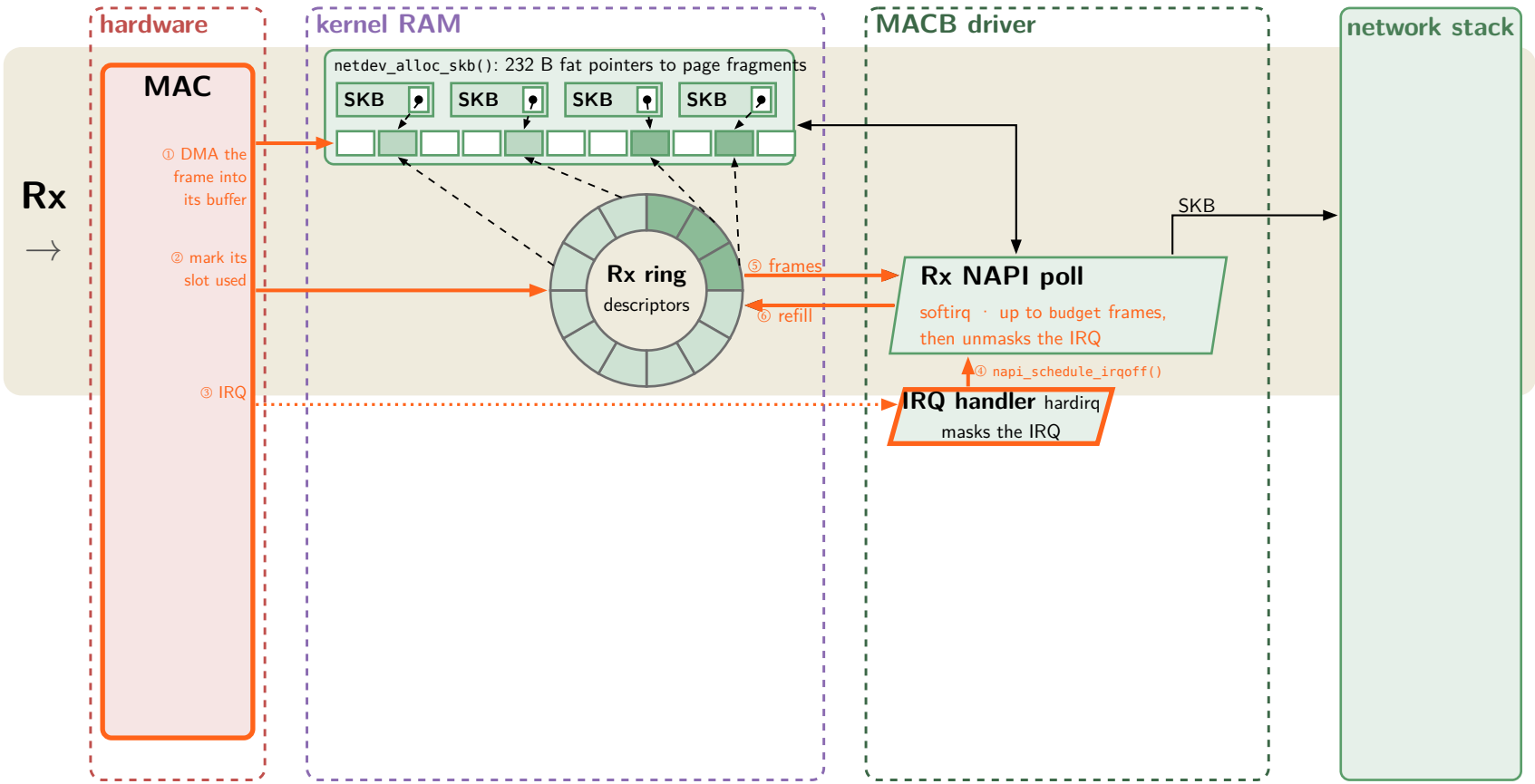
- Our case study for today
- Embedded Ethernet controller **Cadence IP**
- Platform and PCI driver in drivers/net/ethernet/cadence/
- The IP evolved over time (and it shows in the driver)
  - (Old) RM9200: separate `net_device_ops`, 9 Rx descriptors, 1 Tx slot (or 2?)
  - MACB: 10/100M, single Tx/Rx queue
  - GEM: 100M~10G, multi Tx/Rx queues
- Found in many SoCs
  - Microchip SAMA5, SAM9X60, SAMA7G5, PolarFire
  - AMD/Xilinx Zynq, ZynqMP, Versal
  - SiFive HiFive Unleased, Unmatched
  - **Raspberry Pi 5** (RP1 I/O silicon over PCI)
  - **Mobileye EyeQ** family

Rx & Tx: how a driver moves frames



1 **Rx & Tx** — how a driver moves frames
2 XDP — a program in the NAPI poll
3 XSK — frames straight to userspace
4 Context swap — new buffers, no close + open
5 Numbers — why we're here

Rx: descriptor ring & NAPI





Rx refill: one SKB per slot

```
/* gem_rx_refill(), to fill up a ring slot */

// allocate SKB and its underlying page frag
skb = netdev_alloc_skb(netdev,
                       rx_buffer_size);

// create mapping in DMA address space
paddr = dma_map_single(dev, skb->data,
                       rx_buffer_size,
                       DMA_FROM_DEVICE);

// fill up the ring slot
queue->rx_skbuff[entry] = skb;
if (entry == bp->rx_ring_size - 1)
  paddr |= MACB_BIT(RX_WRAP);
desc->ctrl = 0;
dma_wmb();
macb_set_addr(bp, desc, paddr);

skb_reserve(skb, NET_IP_ALIGN); // (maybe)
```

```
/* gem_rx(), once the MAC wrote a frame */

// note down the frame length in SKB
len = desc->ctrl & MACB_RX_FRMLEN_MASK;
skb_put(skb, len);

// sync & delete DMA mapping
addr = macb_get_addr(bp, desc);
dma_unmap_single(dev, addr,
                 rx_buffer_size,
                 DMA_FROM_DEVICE);

// pass to the network stack
napi_gro_receive(napi, skb);

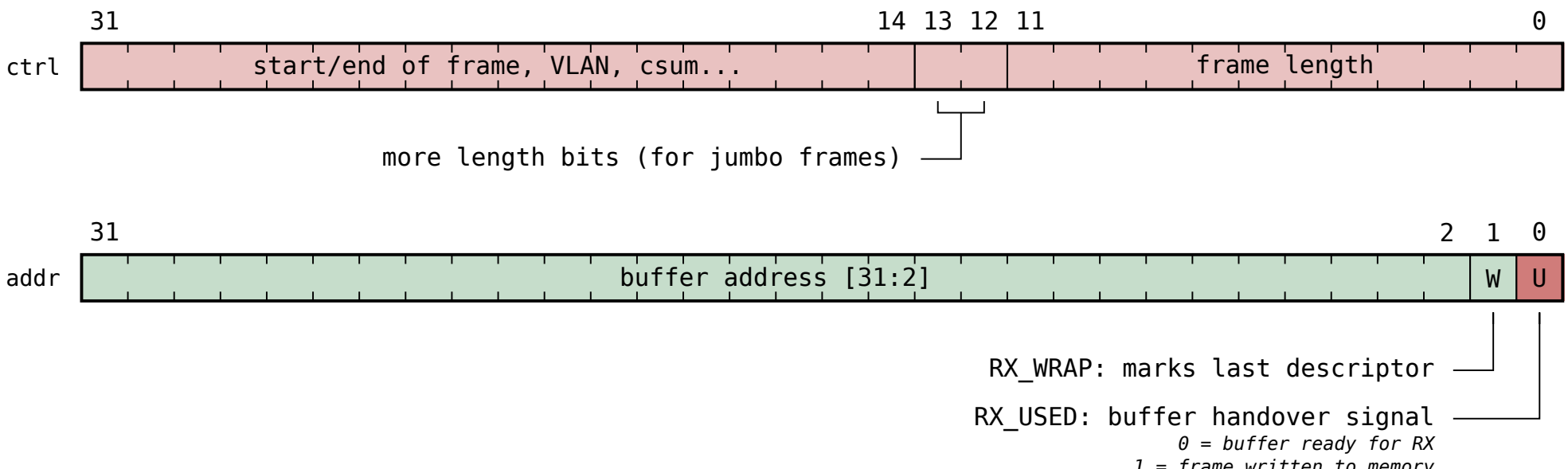
// ... we'll refill the slot ...
```

gem_rx_refill(), gem_rx()

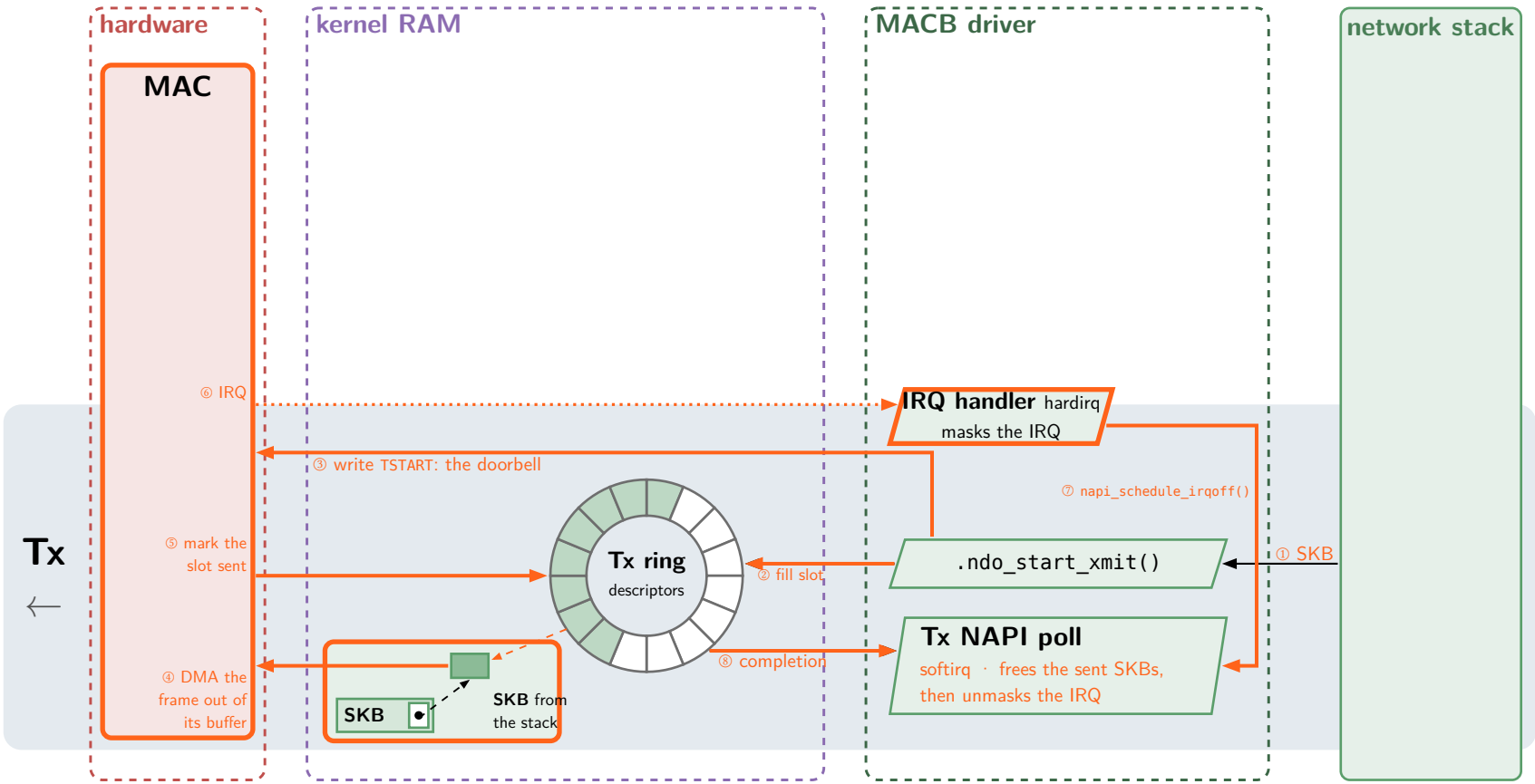


Rx ring descriptor

- Descriptor size is 2, 4 or 6 words
  - addr & control are required words
  - 2 optional words for PTP timestamps
  - 1 more word for 64-bit addresses
- field written by driver, MAC or both



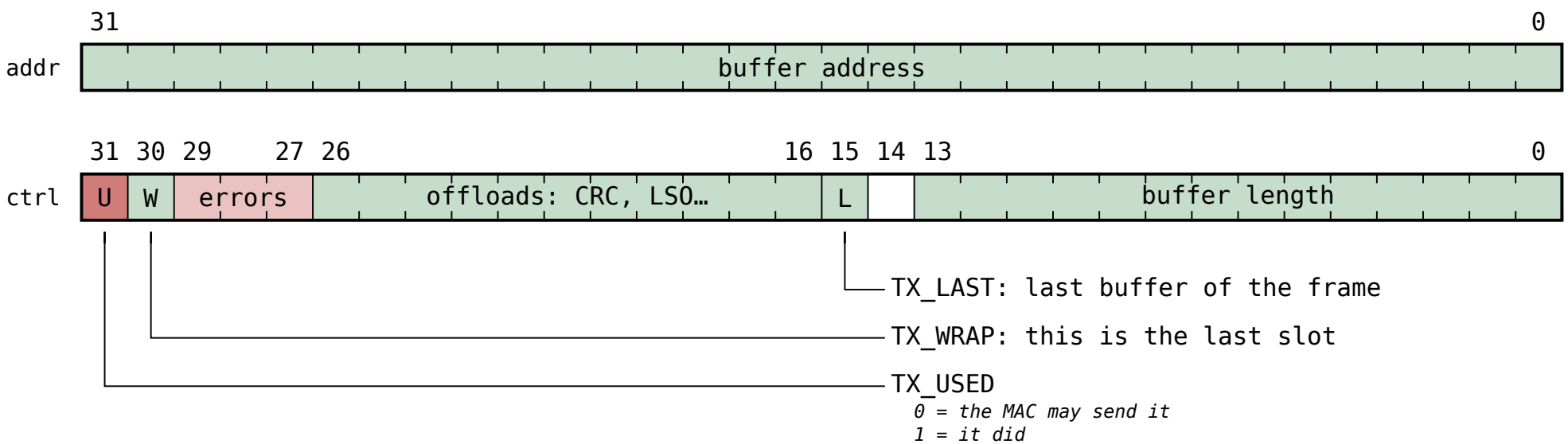
Tx: descriptor ring, doorbell & NAPI



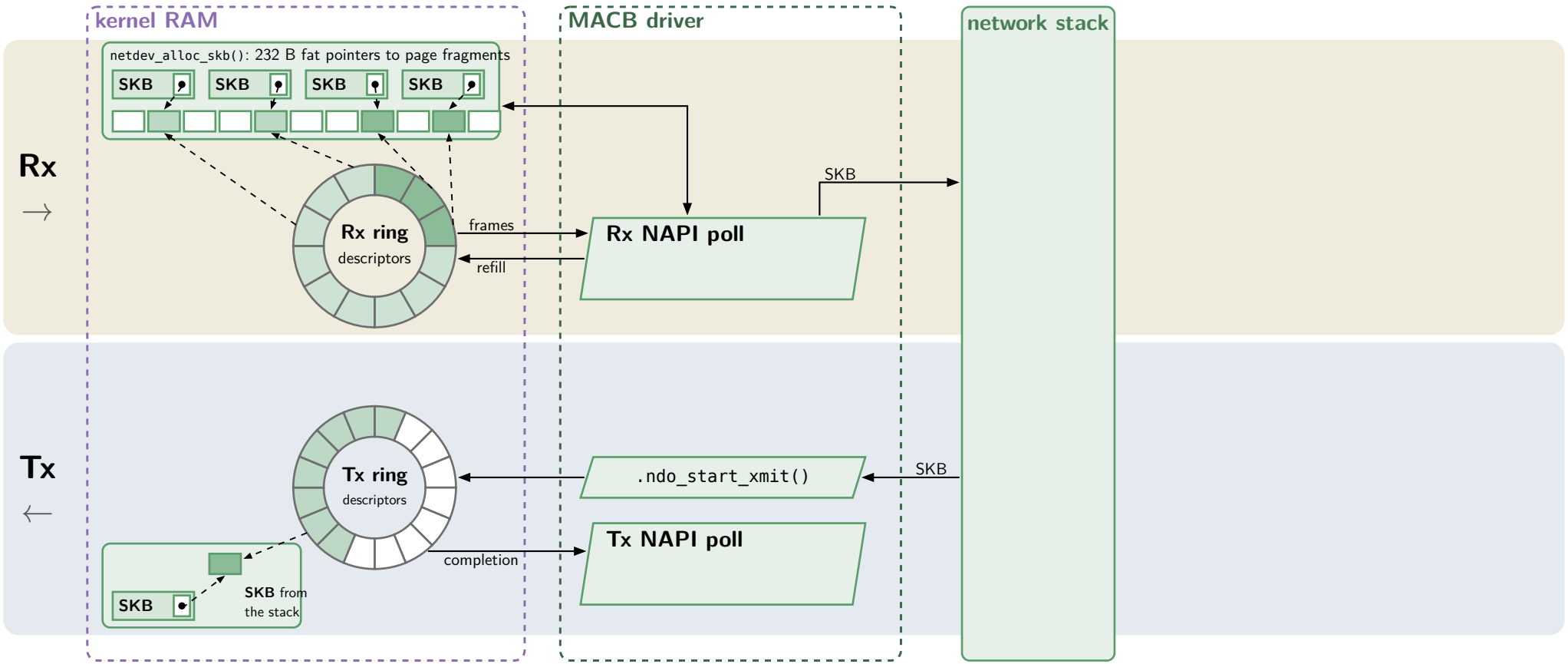


Tx ring descriptor

- Descriptor size is 2, 4 or 6 words
  - address & control are the 2 required words
  - 2 optional words for PTP timestamps
  - 1 more word for 64-bit addresses
- MAC behavior is to emit until it finds a slot with `TX_USED` set
- field written by ■ driver, ■ MAC or ■ both



The driver map



XDP: a program in the NAPI poll



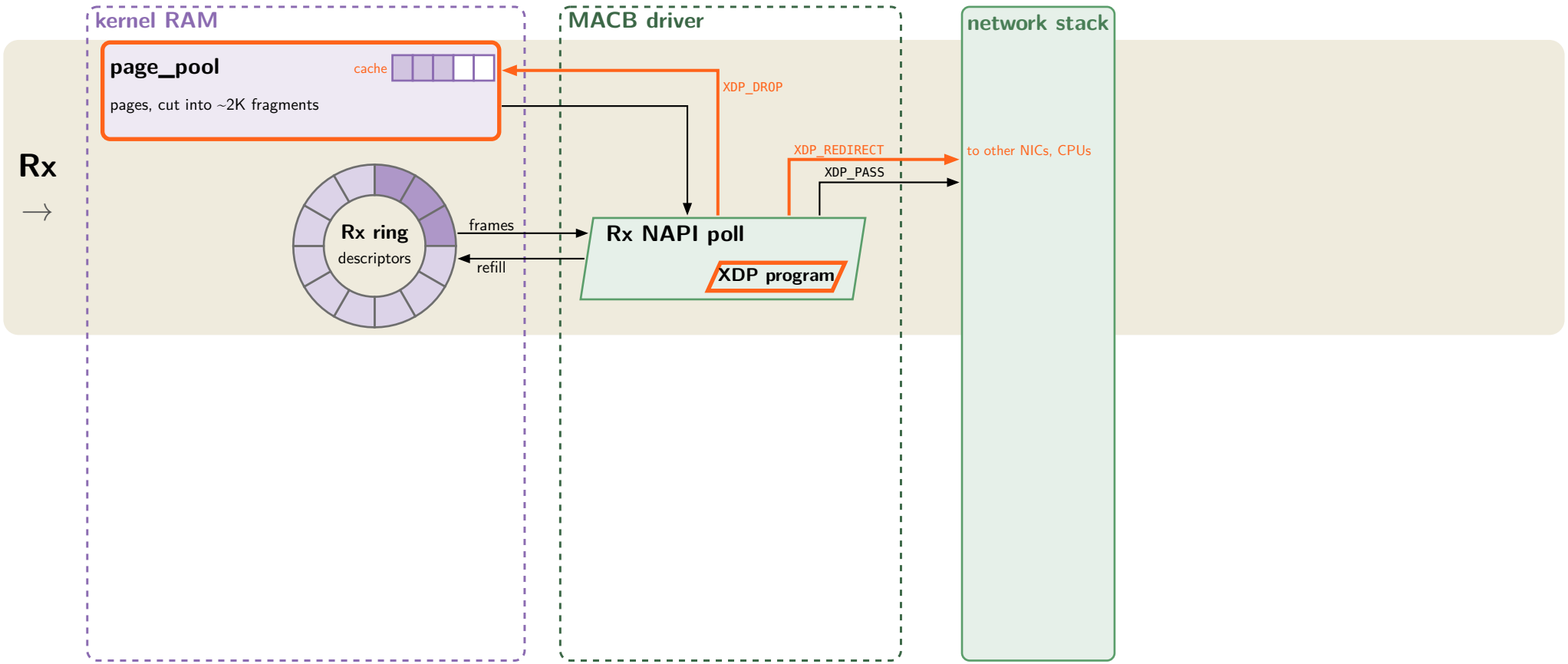
1 Rx & Tx — how a driver moves frames
2 **XDP** — a program in the NAPI poll
3 XSK — frames straight to userspace
4 Context swap — new buffers, no close + open
5 Numbers — why we're here



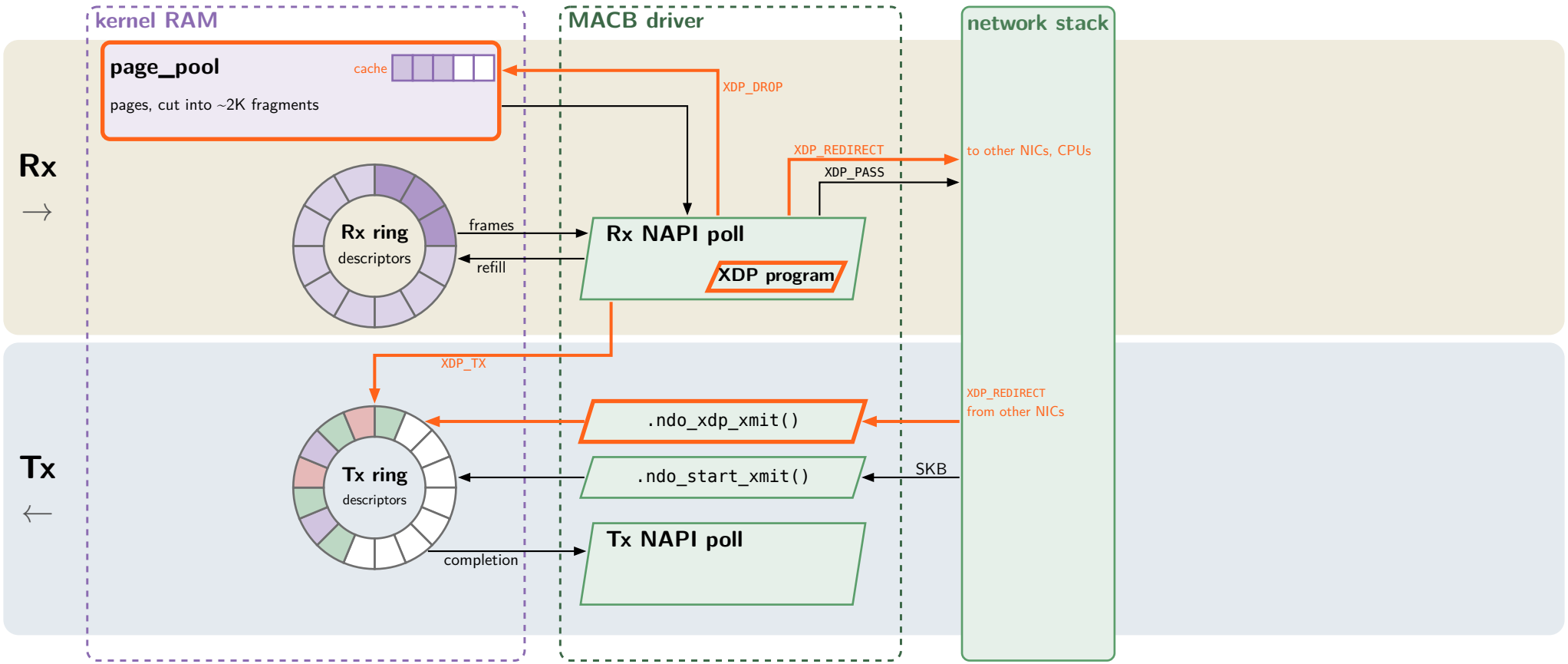
XDP: eXpress Data Path

- Userspace gives us an **eBPF program**
- We run it in a virtual machine, on every received packet
- Main goal is high packet throughput
- It can rewrite the frame, write metadata and **decide the fate of the packet**
  - `XDP_PASS`: give to network stack
  - `XDP_DROP`: silently drop the packet
  - `XDP_ABORTED`: drop the packet (not silently)
  - `XDP_TX`: back out of the same interface
  - `XDP_REDIRECT`: another interface, CPU or userspace AF_XDP socket
- *"Generic XDP"* is when ran from the network stack; let's focus on *"native XDP"*

The driver map, with XDP



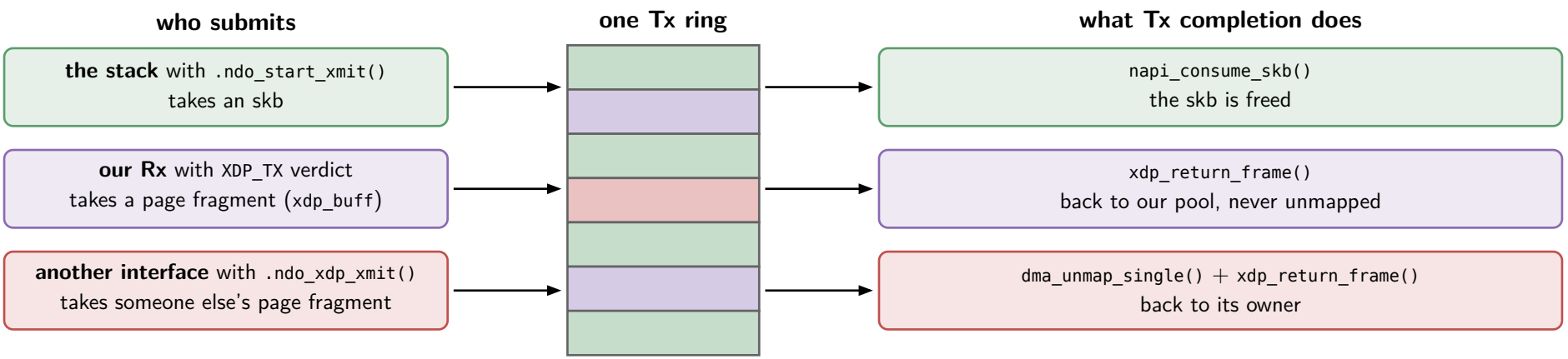
The driver map, with XDP





One Tx ring, three kinds of buffers

- Our Tx ring slots now store more than just SKBs
- And need to remember, to release resources properly





Rx buffers & allocator

Constraints imposed to our Rx buffers:

- **Writable**: program may rewrite the frame and its metadata
- **Headroom**: room in front, to grow headers or store metadata
- **Tailroom**: room behind, to turn the buffer into an skb cheaply
- **Cheap to recycle**: `XDP_DROP` must cost next to nothing
- **Freeable from anywhere**: after `XDP_{TX,REDIRECT}`, someone else frees it

We go from `netdev_alloc_skb()` to the `page_pool` allocator:

- It is a per-queue pool of page fragments, which allows
  - Fixed size buffers
  - DMA map once, then sync on receive
  - Zero contention across queues (spread across CPUs)
- Raw buffers get turned into SKBs on XDP_PASS
- We register on our buffers a *memory model* to signal how to free them
- `page_pool` is the recommended allocator, **even without XDP**

XSK: frames straight to userspace



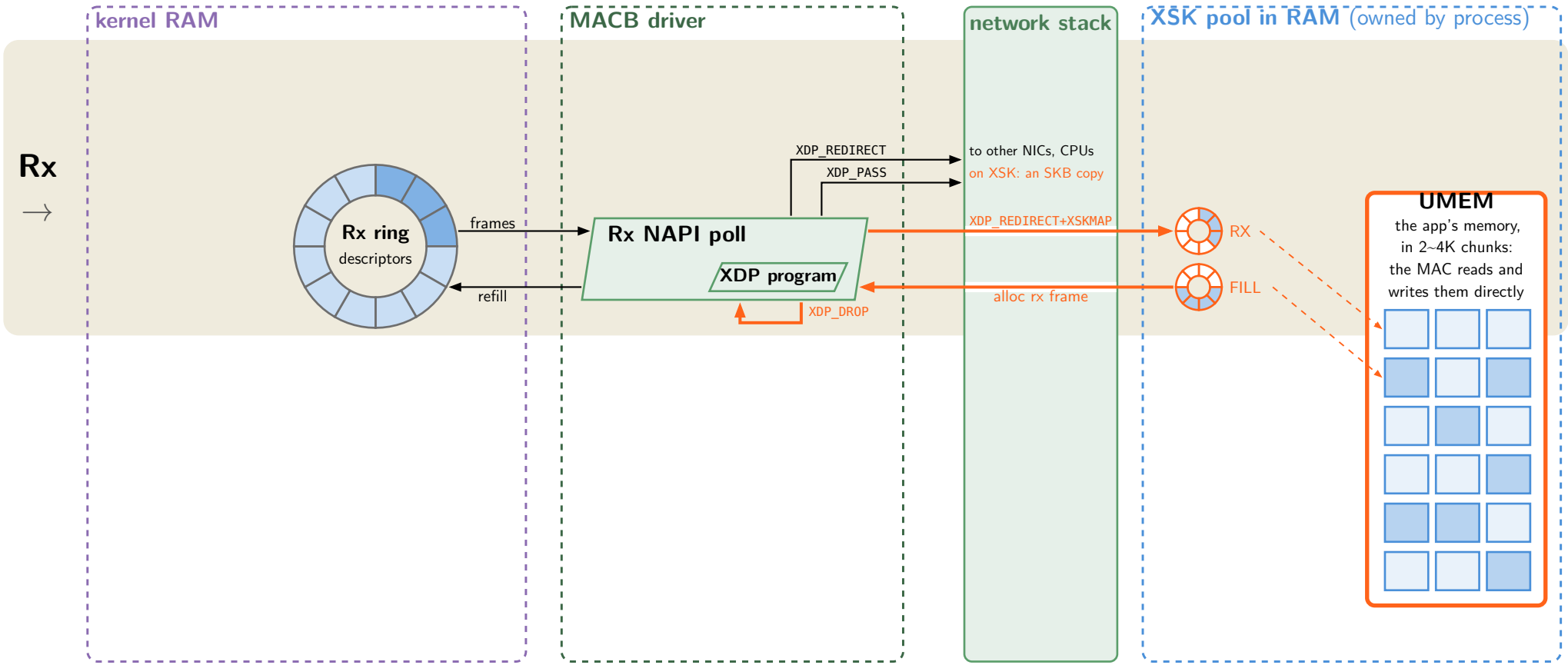
1. Rx & Tx — how a driver moves frames
2. XDP — a program in the NAPI poll
3. **XSK** — frames straight to userspace
4. Context swap — new buffers, no close + open
5. Numbers — why we're here



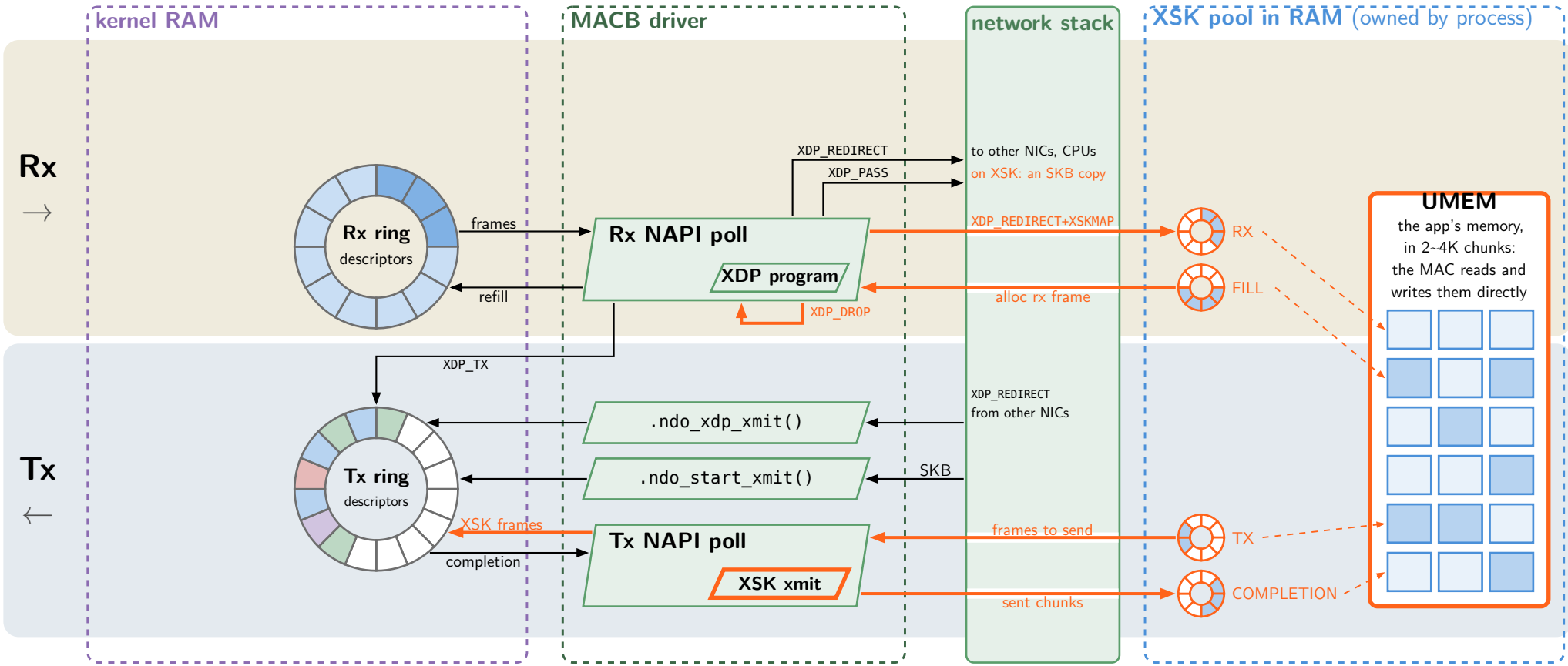
XSK ZC: AF_XDP sockets with zero-copy

- XSK stands for *"AF_**X**DP **s**oc**k**ets"*
- Fully upstream **kernel bypass** strategy for **transmit and/or receive**
  - Zero copy, zero syscall, zero locking
  - Goal: high packet throughput (also called speed)
- Finer grain than userspace driver re-implementations
  - Decide on routing to the network stack using XDP
  - Combines well with RX flow classification rules (one socket per queue)
- Userspace owns and preallocates buffers, then lends them to kernelspace
  - Kernel bulk DMA maps all buffers on attach
- Communication occurs through 4 lockless rings

The driver map, with XSK



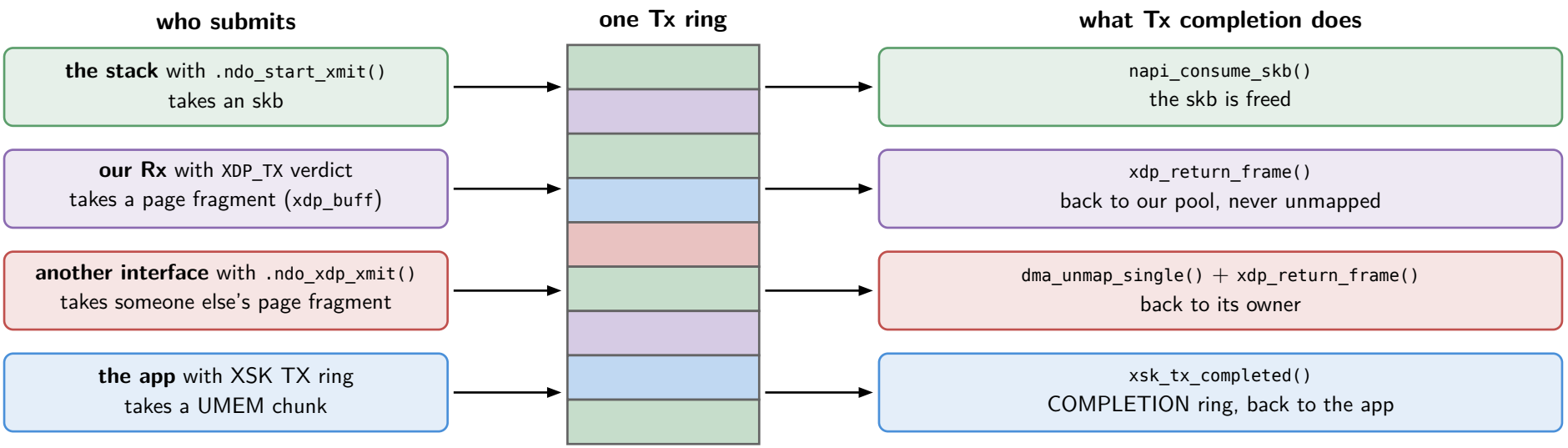
The driver map, with XSK





One Tx ring, four kinds of buffers

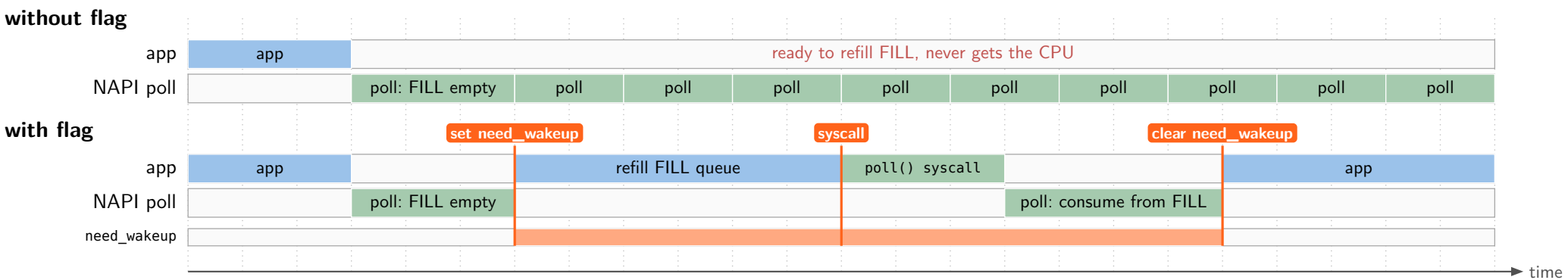
- Tx ring slots can contain one more slot type: `MACB_TYPE_XSK`
- Return it to the COMPLETION ring, no DMA unmap nor kfree





need_wakeup: sleep, don't spin

- NAPI poll will spin when missing buffers, by default
- Acceptable when NAPI CPU ≠ app CPU
- With `XDP_USE_NEED_WAKEUP` flag: instead *synchronise* using two shared memory `need_wakeup` booleans (one per direction)
- Made for core handling all traffic, both NAPI and AF_XDP process
- Commit 77cd0d7b3f25 ("xsk: add support for need_wakeup flag in AF_XDP rings")





Attaching a pool to one queue

```
/* .ndo_bpf(XDP_SETUP_XSK_POOL) */

if (pool && !gem_xdp_valid_mtu(bp, netdev->mtu))
    return -EOPNOTSUPP;    /* 1 frame = 1 buffer */

if (pool)
    err = xsk_pool_dma_map(pool, dev, attrs);

if (running) {
    new_ctx = macb_context_alloc(bp, ...);
    macb_context_swap_start(bp);
}

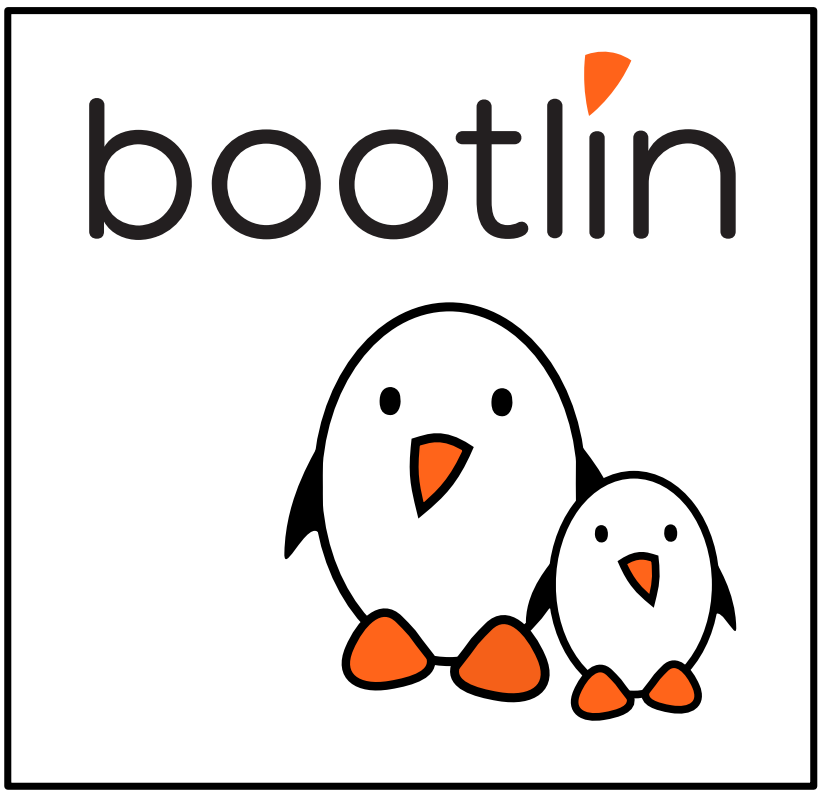
bp->info.xsk_pool[qid] = pool;

if (running)
    macb_context_swap_end(bp, new_ctx);
```

- AF_XDP socket is per queue: other queues keep their page_pool allocator
- The UMEM gets DMA-mapped once, at attach
- One frame, one descriptor: the MTU is capped as for XDP
- No XDP program required: Tx-only sockets work without one
- Advertise `XSK_ZEROCOPY` at probe, or the core refuses zero-copy binds

Context swapping

1. Rx & Tx — how a driver moves frames
2. XDP — a program in the NAPI poll
3. XSK — frames straight to userspace
4. **Context swap** — new buffers, no close + open
5. Numbers — why we're here



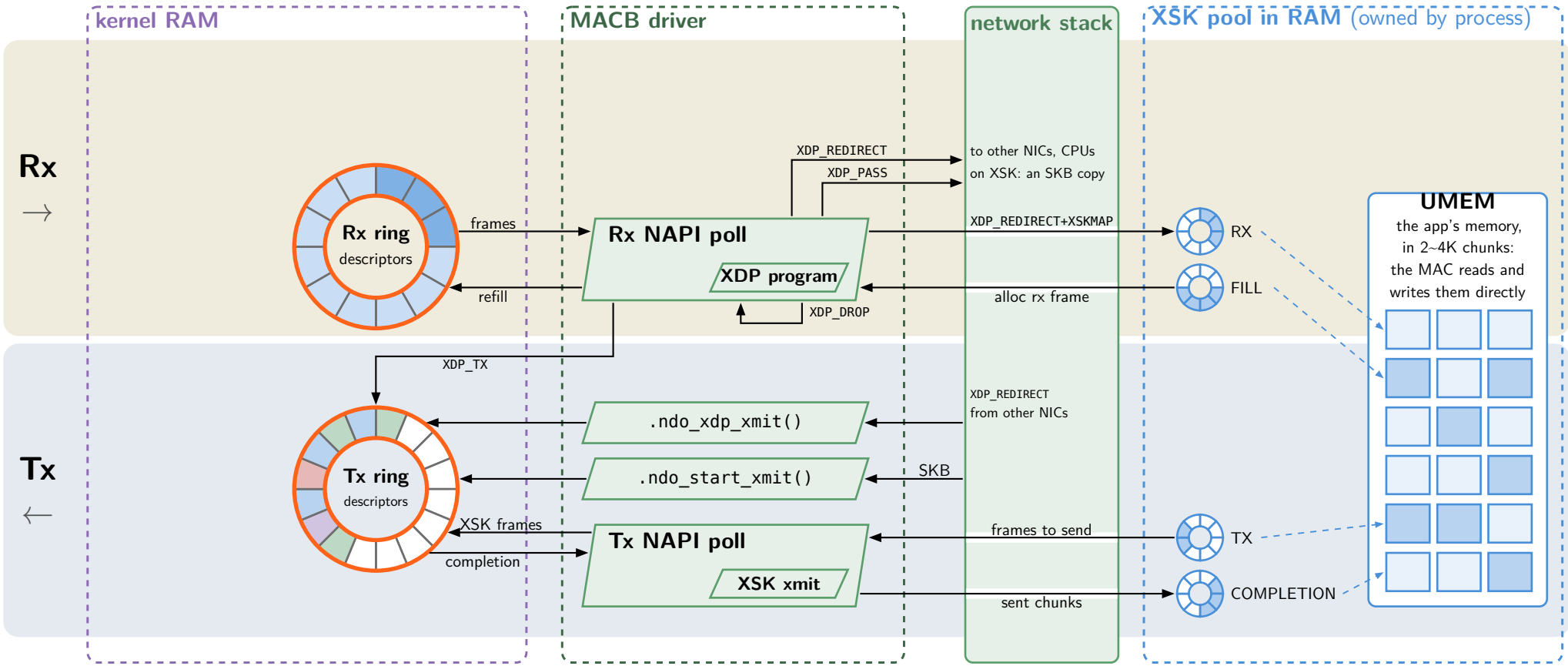


Context swapping

- A **context swap** occurs when an operation needs to change buffer attributes: size, headroom, mapping, owner, etc.
- It implies reallocating all Rx buffers
- And freeing pending Tx buffers

Operation	Driver callback	Command
Configure # of active queues	`ethtool_ops::set_channels()`	`ethtool -L eth0 combined 1`
change ring slots count	`ethtool_ops::set_ringparam()`	`ethtool -G eth0 rx 42`
change MTU	`.ndo_change_mtu()`	`ip link set dev eth0 mtu 999`
Attach XDP program	`.ndo_bpf(XDP_SETUP_PROG, ...)`	`xdp-loader load eth0 prog.o`
Attach XSK pool	`.ndo_bpf(XDP_SETUP_XSK_POOL, ...)`	`xdp-bench xsk-drop eth0`

The driver map: what gets swapped





Context swapping

- The old way is to fully close then re-open
  - It forces a `phylink_stop()`
  - Most importantly: it does not resist memory pressure

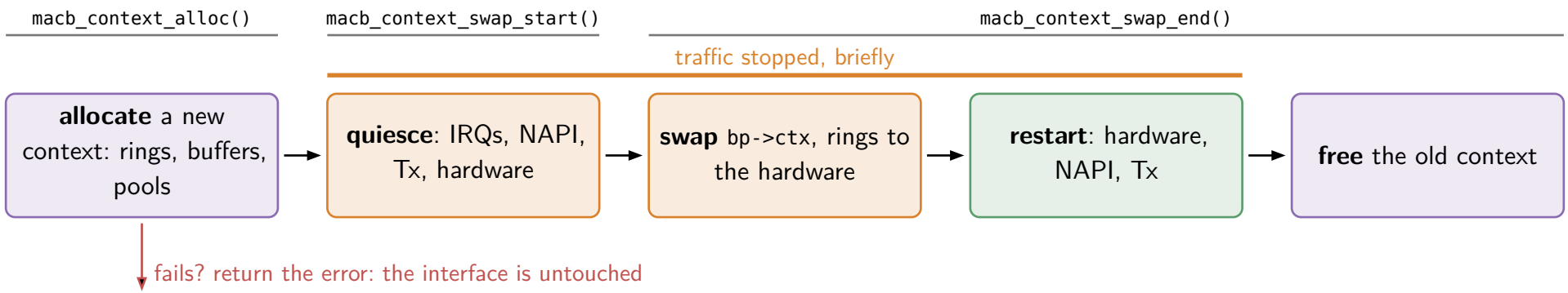
```
static int gem_xdp_setup(struct net_device *netdev, struct bpf_prog *prog, ...)
{
    bool need_update = !!bp->prog != !!prog;
    bool running = netif_running(netdev);

    if (running && need_update)
        macb_close(netdev); // free everything, shutdown MAC & PHY

    old_prog = rcu_replace_pointer(bp->prog, prog, lockdep_rtnl_is_held());

    if (running && need_update)
        err = macb_open(netdev); // restart it all
    if (old_prog)
        bpf_prog_put(old_prog);
    return 0;
}
```

Context swapping



- Allocate first: a failure costs nothing and **board stays online**
- Everything that depends on the configuration lives in the context: ring & buffer sizes, pointers, DMA mappings, pending SKB, etc.
- For a short moment, we own two different contexts
- Careful with function prototypes, to avoid touching the wrong context



Quiescing, in the right order

IRQs first, NAPI second

```
queue_writel(queue, IDR, -1);
macb_queue_isr_clear(bp, queue, -1);

for (q = 0; q < bp->num_queues; ++q)
    synchronize_irq(queues[q].irq);

// <!!!> here, NAPI might unmask IRQs <!!!>

for (q = 0; q < bp->num_queues; ++q) {
    napi_disable(&queues[q].napi_rx);
    napi_disable(&queues[q].napi_tx);
}
```



Quiescing, in the right order

IRQs first, NAPI second

```
queue_writel(queue, IDR, -1);
macb_queue_isr_clear(bp, queue, -1);

for (q = 0; q < bp->num_queues; ++q)
    synchronize_irq(queues[q].irq);

// <!!!> here, NAPI might unmask IRQs <!!!>

for (q = 0; q < bp->num_queues; ++q) {
    napi_disable(&queues[q].napi_rx);
    napi_disable(&queues[q].napi_tx);
}
```

NAPI first, IRQs second

```
for (q = 0; q < bp->num_queues; ++q) {
    napi_disable(&queues[q].napi_rx);
    napi_disable(&queues[q].napi_tx);
}

// <!!!> here, IRQs can schedule NAPI <!!!>

queue_writel(queue, IDR, -1);
macb_queue_isr_clear(bp, queue, -1);

for (q = 0; q < bp->num_queues; ++q)
    synchronize_irq(queues[q].irq);
```



Quiescing, in the right order

IRQs first, NAPI second

```
queue_writel(queue, IDR, -1);
macb_queue_isr_clear(bp, queue, -1);

for (q = 0; q < bp->num_queues; ++q)
    synchronize_irq(queues[q].irq);

// <!!!> here, NAPI might unmask IRQs <!!!>

for (q = 0; q < bp->num_queues; ++q) {
    napi_disable(&queues[q].napi_rx);
    napi_disable(&queues[q].napi_tx);
}
```

NAPI first, IRQs second

```
for (q = 0; q < bp->num_queues; ++q) {
    napi_disable(&queues[q].napi_rx);
    napi_disable(&queues[q].napi_tx);
}

// <!!!> here, IRQs can schedule NAPI <!!!>

queue_writel(queue, IDR, -1);
macb_queue_isr_clear(bp, queue, -1);

for (q = 0; q < bp->num_queues; ++q)
    synchronize_irq(queues[q].irq);
```

One solution: IRQs active but self-disarming

Numbers

1. Rx & Tx — how a driver moves frames
2. XDP — a program in the NAPI poll
3. XSK — frames straight to userspace
4. Context swap — new buffers, no close + open
5. **Numbers** — why we're here





Numbers

- **!!! Not your workload !!!**



Numbers

- **!!! Not your workload !!!**
- Only throughput measurements, don't forget the latency distribution



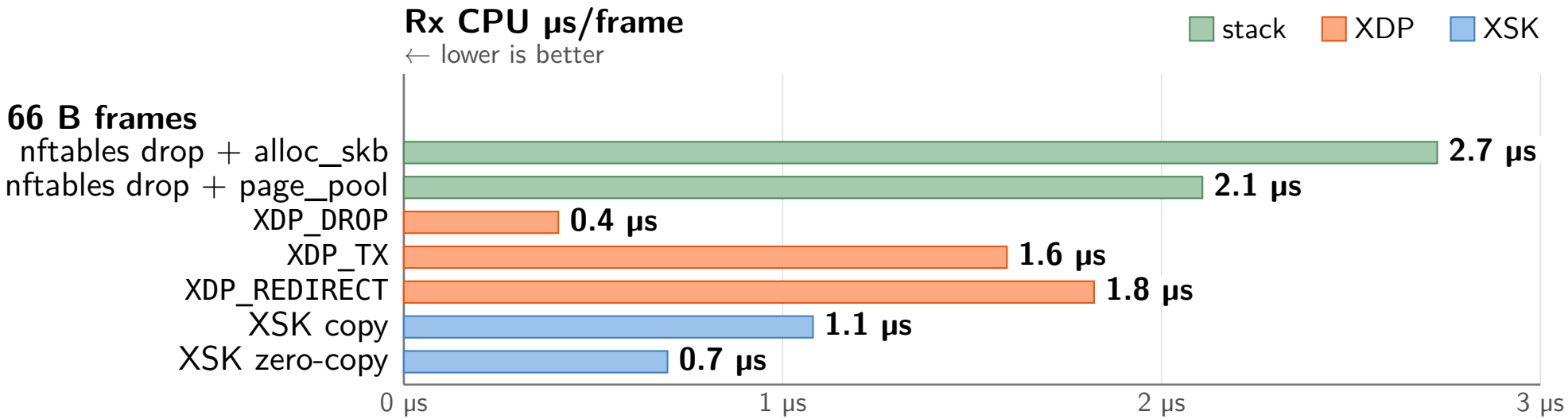
Numbers

- **!!! Not your workload !!!**
- Only throughput measurements, don't forget the latency distribution
- Driver has not been profiled nor optimised (talk to its maintainer)



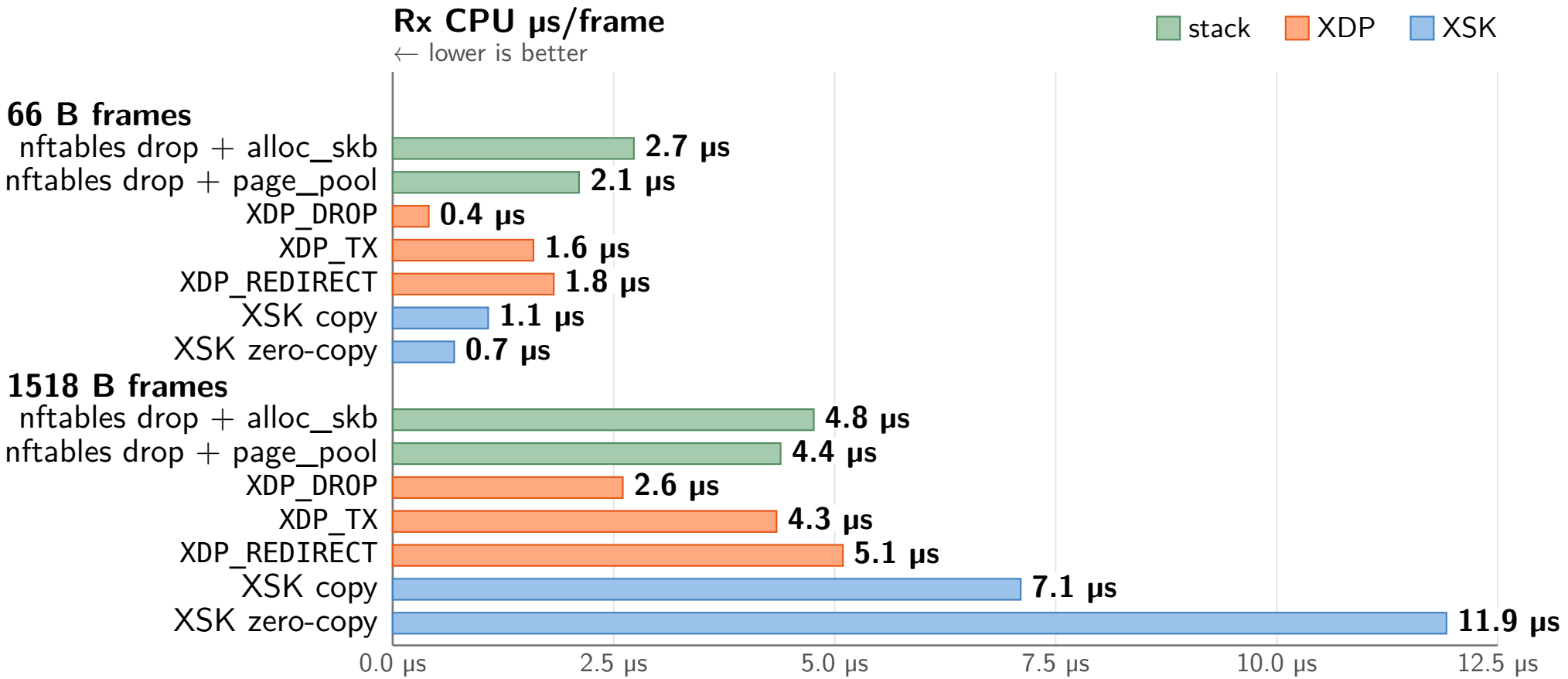
Rx CPU time per frame

- EyeQ5 eth1→eth0 1G, 2 CPU Tx, 1 CPU Rx, 4×I6500 cores @ 1.5GHz, 3×30s



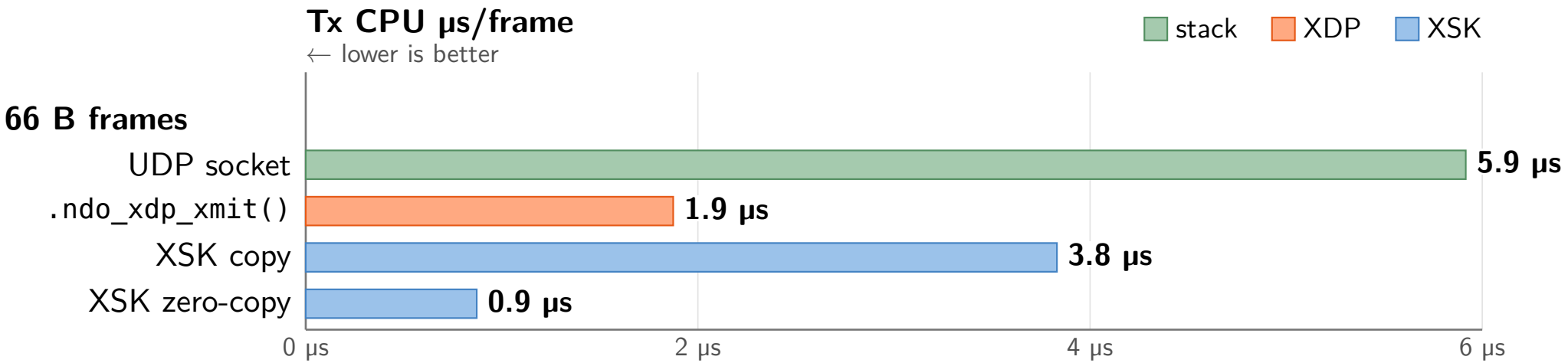
Rx CPU time per frame

- EyeQ5 eth1→eth0 1G, 2 CPU Tx, 1 CPU Rx, 4×I6500 cores @ 1.5GHz, 3×30s



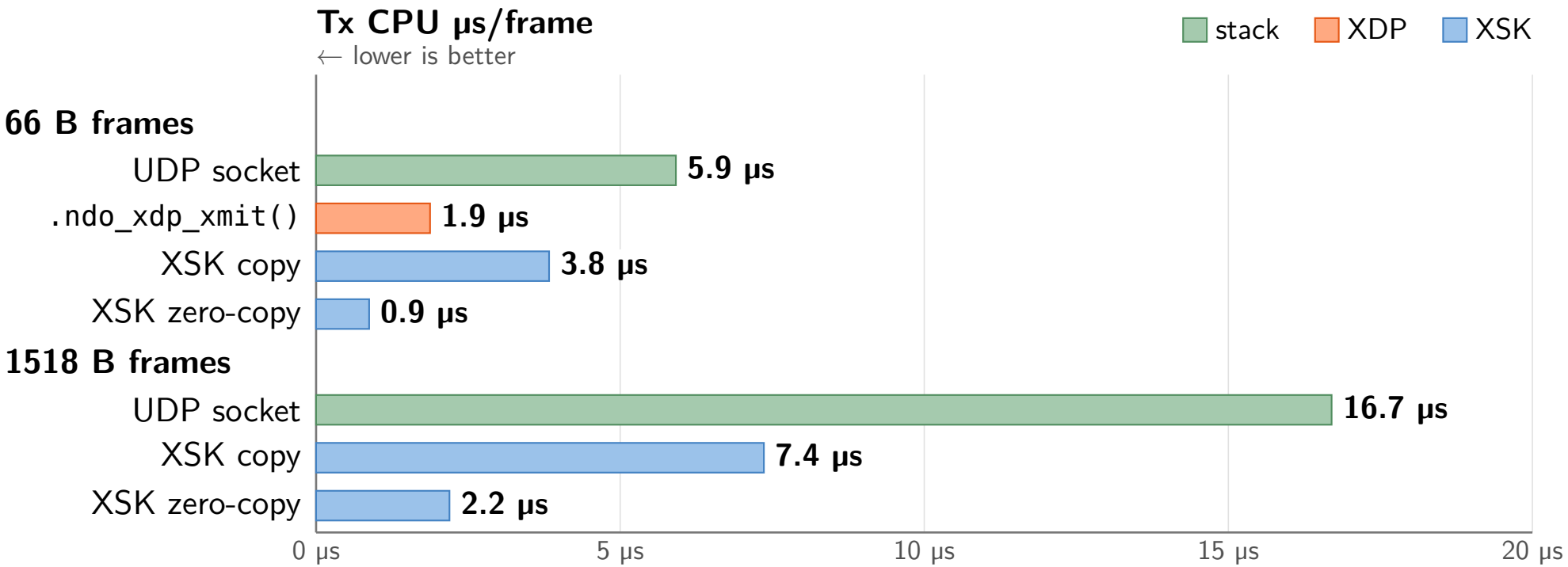
Tx CPU time per frame

- EyeQ5 eth1→eth0 1G, 1 CPU Tx, 4×I6500 cores @ 1.5GHz, 3×30s



Tx CPU time per frame

- EyeQ5 eth1→eth0 1G, 1 CPU Tx, 4×I6500 cores @ 1.5GHz, 3×30s

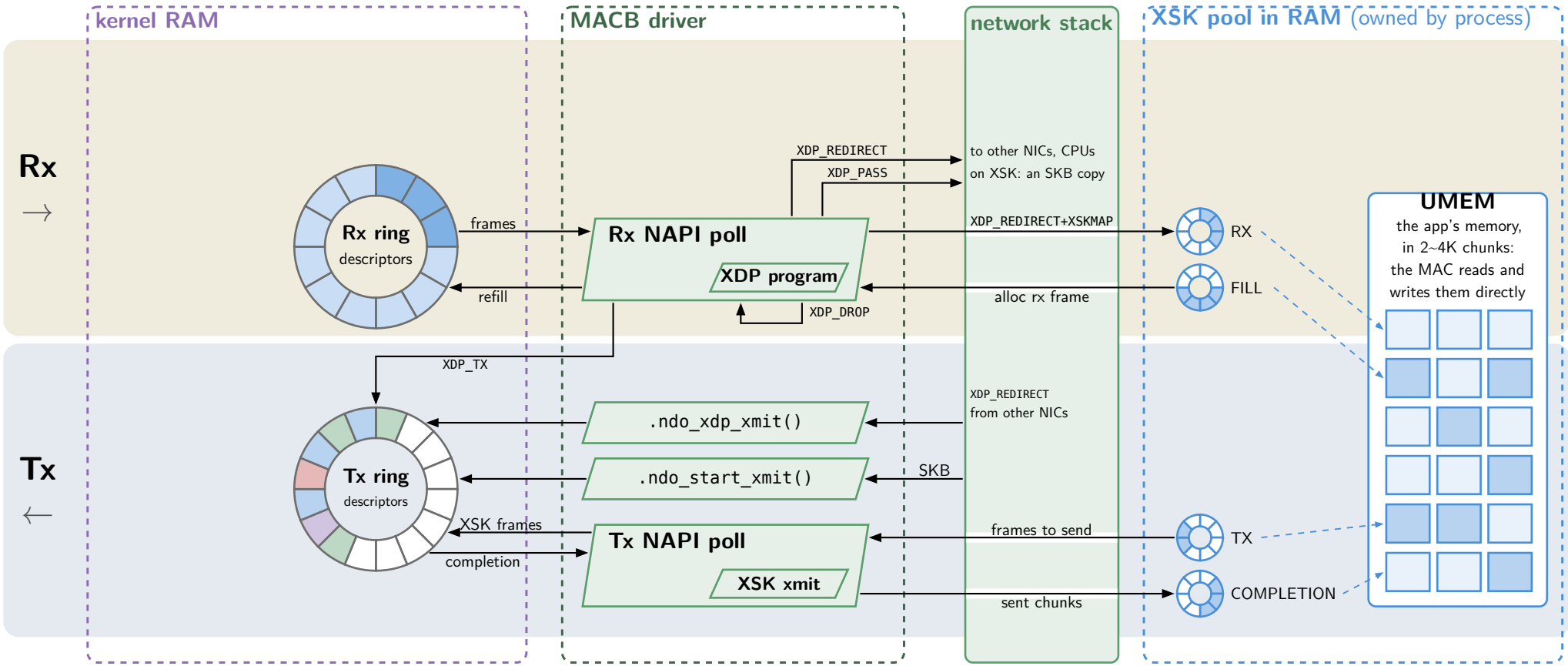


Conclusion

1. Rx & Tx — how a driver moves frames
2. XDP — a program in the NAPI poll
3. XSK — frames straight to userspace
4. Context swap — new buffers, no close + open
5. Numbers — why we're here



Conclusion


kernel RAM
MACB driver
network stack
XSK pool in RAM (owned by process)
Rx
Tx
Rx ring
descriptors
frames
refill
Rx NAPI poll
XDP program
XDP_REDIRECT
XDP_PASS
XDP_DROP
to other NICs, CPUs
on XSK: an SKB copy
XDP_REDIRECT+XSKMAP
alloc rx frame
RX
FILL
UMEM
the app's memory,
in 2~4K chunks:
the MAC reads and
writes them directly
XDP_TX
Tx ring
descriptors
.ndo_xdp_xmit()
.ndo_start_xmit()
XDP_REDIRECT
from other NICs
SKB
XSK frames
completion
Tx NAPI poll
XSK xmit
frames to send
sent chunks
TX
COMPLETION



Takeaways

- **Network device:** talk to HW using descriptor rings, polling rings using NAPI
- **XDP:** on Rx poll, run eBPF program against each packets
- **XSK:** allow XDP_REDIRECT from/to NAPI poll to/from userspace
- **XSK zero-copy:** MAC directly writes to or reads from userspace-owned buffers



Takeaways

- **Network device:** talk to HW using descriptor rings, polling rings using NAPI
- **XDP:** on Rx poll, run eBPF program against each packets
- **XSK:** allow XDP_REDIRECT from/to NAPI poll to/from userspace
- **XSK zero-copy:** MAC directly writes to or reads from userspace-owned buffers

- **Upstream status:**
  - Context swapping V10
  - Page pool migration + XDP, V6, pending, From: Paolo Valerio
  - AF_XDP zero-copy V1 RFC, pending



References

- Bootlin's Embedded Linux networking trainings, material under CC-BY-SA 3.0
  - Also see Real-time Linux training
  - And Linux kernel and driver development training
- Kernel documentation
  - Documentation/networking/af_xdp.rst
  - Documentation/networking/page_pool.rst
- Kernel code
  - include/net/xdp.h
  - include/net/xdp_sock_drv.h
  - include/net/page_pool/helpers.h
  - net/core/filter.c (comment about how XDP_REDIRECT works)
- Netdev series
  - [PATCH net-next v6 0/7] net: macb: Add XDP support and page pool integration
  - [PATCH net-next 0/8] net: macb: add XSK support
  - [PATCH net-next v9 00/17] net: macb: implement context swapping

Thank you!

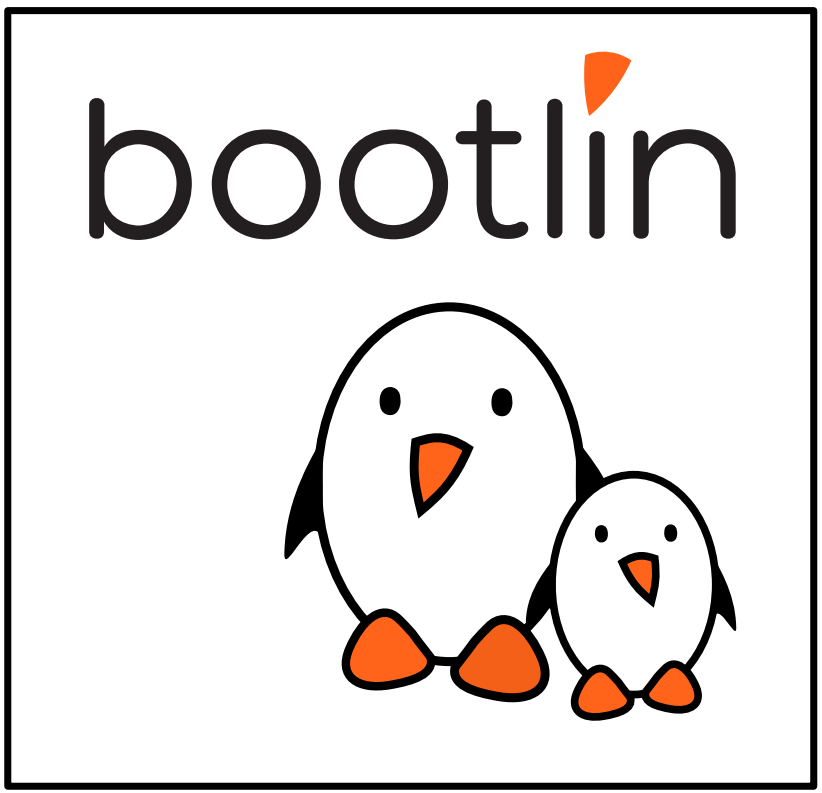
Questions?

Théo Lebrun

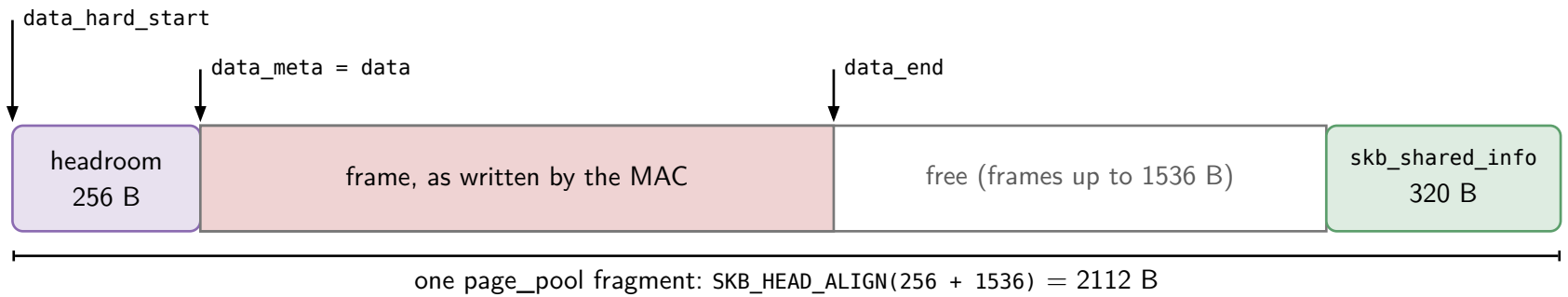
Slides under CC-BY-SA 3.0

https://bootlin.com/pub/conferences/2026/

Annex



Rx buffer layout



MTU 1500, drawn to scale

- The program may move `data` and `data_meta` into the headroom using bpf_xdp_adjust_head() & bpf_xdp_adjust_meta()
- `data_end` is set by the driver, on receive, based on descriptor frame length
- napi_build_skb() allocates an SKB pointing to the same memory for XDP_PASS



How many buffers fit in a page?

MTU 1500	cache line	headroom	frame	shinfo	total	per page
RPi5, no program	64 B	64	1536	320	1920	2 per 4K
RPi5, XDP	64 B	256	1536	320	2112	**1** per 4K
EyeQ5, no program	128 B	128	1536	384	2048	8 per 16K
EyeQ5, XDP	128 B	256	1536	384	2176	**7** per 16K

- RPi5 page size is configurable, see `CONFIG_ARM64_16K_PAGES=y`, default is 4K
- Headroom size
  - Without XDP $\Longrightarrow$ one cache line
  - With XDP $\Longrightarrow$ `XDP_PACKET_HEADROOM` = 256 bytes
- As size changes, `page_pool` reallocated on !XDP $\leftrightarrow$ XDP transitions



XDP_REDIRECT, step by step

1. The program calls `bpf_redirect()` or `bpf_redirect_map()`: the target is looked up and stored in a per-CPU struct bpf_redirect_info
2. It returns `XDP_REDIRECT`; the driver calls `xdp_do_redirect()`, which queues the frame in a bulk queue of that map type
3. Before leaving its NAPI poll, the driver calls `xdp_do_flush()`: bulk queues are flushed, the redirect completes
   - Before `napi_complete_done()`: RCU protection relies on staying inside one NAPI instance

- DEVMAP: another interface, through its `.ndo_xdp_xmit()`
- CPUMAP: another CPU builds the skb
- XSKMAP: an AF_XDP socket

From the comment in net/core/filter.c



Becoming a XDP native driver

```
// macb_probe()

netdev->xdp_features = NETDEV_XDP_ACT_BASIC |
                       NETDEV_XDP_ACT_REDIRECT |
                       NETDEV_XDP_ACT_NDO_XMIT;

err = register_netdev(netdev);
```

- Steps:
  1. Advertise at probe
  2. Register XDP program callback
- `bp->prog` is RCU-protected: Rx reads, attach swaps
- !prog ↔ prog: new buffer layout means new pool

```
// .ndo_bpf(netdev, xdp) & xdp.cmd=XDP_SETUP_PROG

running = netif_running(dev);
need_update = !!bp->prog != !!prog;
if (running && need_update)
  macb_close(dev);

old_prog = rcu_replace_pointer(bp->prog, prog,
                               lockdep_rtnl_is_held());

if (running && need_update)
  macb_open(dev);

if (old_prog)
  bpf_prog_put(old_prog);
```