SKB creation outside drivers: using metadata and HW-offloads?

Toke Høiland-Jørgensen (Red Hat)
Jesper Dangaard Brouer (Red Hat)

Netconf
Boston, June 2019

Framing XDP

XDP: in-kernel programmable (eBPF) layer before netstack

- (AF_XDP is our selective kernel-bypass to userspace)

XDP ensures that Linux networking stays relevant

- Operates at L2-L3, netstack is L4-L7

XDP is not first mover, but we believe XDP is different and better

- Flexible sharing of NIC resources
- Killer feature: Integration with Linux kernel
  - This talk is about extending this integration further

Move SKB allocations out of NIC drivers

Goal: Simplify driver, via creating SKB inside network-core code

- Happens today via `xdp_frame` in both veth and cpumap

The `xdp_frame` is placed in top of data-frame (`data_hard_start`)

- Currently 32-bytes

Issue: SKB's created this way are lacking HW-offloads like:

- HW checksum info (for `skb->ip_summed` + `skb->csum`)
- HW RX hash (`skb_set_hash(hash, type)`)
- (these are almost always needed... tempted to extend `xdp_frame`)

Other HW-offloads

Other existing offloads, used by SKBs, but not always enabled

- VLAN (`__vlan_hwaccel_put_tag()`)
- RX timestamp
  - HW `skb_hwtstamps()` (stored in skb_shared_info)
  - Earlier XDP software timestamp (for `skb->tstamp`)
- RX mark (`skb->mark` supported by mlx5)

Other potential offloads, which hardware can do (but not used by SKB):

- Unique u64 flow identifier key (mlx5 HW)
- Higher-level protocol header offsets
  - RSS-hash can deduce e.g. IPv4/TCP (as frag not marked as TCP)
  - But NIC HW have full parse info avail

The holy-grail for HW-offloads

The GOAL is to come-up with a Generic Offload Abstraction Layer...

Generic and dynamic way to transfer HW-offload info

- Only enable info when needed
- Both made available for SKB creation and XDP programs

The big questions are:

- Where to store this information?
- How to make it dynamic?

Simple static solution

The simple solution that isn't as dynamic as we want...

Have drivers send along extra struct with info to `xdp_do_redirect()`

- Use info-struct when calling `convert_to_xdp_frame()`

Drivers have to fill-out info-struct every time

- Driver basically transfer info from descriptor to info-struct
- All drivers have to agree on struct layout

The XDP-prog don't have access to info-struct

- As `xdp_do_redirect()` happens after XDP-prog runs
  - (could be solved by also giving info-struct to XDP-prog)

Use NIC frame descriptor directly? (No)

This came up before... why not give NIC frame descriptor directly to BPF?

Why can't we use frame descriptor directly?

- Very compact bit format and union overloaded
  - Even if possible to describe via BTF
  - Prog to decode too specific to vendor HW (+ revision)
- HW revisions have erratas (e.g. ixgbe csum invalid in one HW rev)
  - A driver translation function should handle/hide this
- With cpumap xdp_frame is read on remote CPU, descriptor not-valid

Where to store the offload info?

At least info-struct should be described via BTF

Instead of separate info-struct, store info-struct in data-frame area?

- Two options:
  - Use XDP metadata area (already avail to XDP)
  - Use areas "inside" xdp_frame (or dynamic area after xdp_frame ends)
    - not curr avail to XDP (as `xdp_frame` is created after XDP-prog ran)

Note: Cannot store info-struct inside `xdp_rxq_info`

- Because not a per frame data-structure, and xdp_frame use bulk processing

Background: What is XDP-metadata area?

Background slide, what do we have today...

XDP have 32 bytes metadata in front of payload (`xdp_buff->data_meta`)

- XDP tail-calls can read this (transfer info between tail-calls)
- TC eBPF (cls_bpf) can read this, and update SKB fields
  - E.g. save XDP lookup and use in TC eBPF hook
- AF_XDP raw frames have this metadata avail in front of payload

Safe to allow XDP to update offload info?

Can we allow XDP to update offload info area?

- Happens before SKB field update
- Are there any safety issues? (kernel netstack stability)
- XDP could potentially fix HW-offload fields

Likely need some boundary checks

- Especially for higher-level protocol header offsets

Can verifier tell us

- if XDP prog changed metadata area?

Lacking knowledge about BTF

When info-struct is described via BTF

- Can kernel code understand BTF and act dynamically???
  - In `convert_to_xdp_frame()` code
  - And in xdp_frame to SKB update fields code?

Hack: if driver knows order struct-members can appear in

- Walk BTF format and create bitmap with enabled members
- When member is matched, increment iterator with member size
- (Fear this is slow, due to data dependency on iterator)

Driver call-back function

Driver fill-out “info-struct”, thus knows layout

- xdp_frame to SKB conversion, use driver call-back to update SKB fields?

One step further

- Could driver call-back be a BPF-prog, that update SKB fields?

How to configure driver for this?

Next challenge: What is the interface for configuring this?

- Extending `ndo_bpf` seems obvious
- But there is a dependency between
  - info-struct, driver populate, and SKB-update call-back
  - If XDP-prog use BTF-metadata layout
    - how to handle (or lock) BTF-layout changes runtime

Driver static approach

Steps for static driver

- Step#A: Driver define static info-struct for metadata area
  - Create BTF-format (via macros) and register with BPF (?)
  - Adjust `xdp_buff->data_meta` with info-struct size
- Step#B: Driver function populates metadata with offloads from descriptor
  - It knows about HW offloads curr enabled, revisions and quirks
- XDP-prog is called (how does user get BTF-format?)
- Step#C: Driver static SKB-update call-back
  - Via XDP-redirect (either cpumap or veth) call-back is invoked with SKB

More dynamic approach

Same steps: Step#A + Step#B

- Step#A: Driver defines static info-struct for metadata area
  - Create BTF-format (via macros) and register with BPF
- Step#B: Driver function populates metadata with offloads from descriptor

Dynamic BPF call-back

- Step#C: Driver SKB-update call-back is a BPF-prog
- Validation trick:
  - This SKB-update BPF-prog, must have map named 'metadata'
  - map must have BTF-format that matches driver BTF-format
    - checked on attach via `ndo_bpf`, else reject

When to enable populate metadata

The populate metadata function is not enabled by default

- Driver creates real BPF-map with BTF-format for metadata (as value)
  - (Key is driver "id" for this map, allow for more maps per driver)
- Add `ndo_bpf` query for metadata-map, return map-fd
  - Both XDP-prog and SKB-update prog can use map

Trigger to enable/disable, when map-user gets attached/detached

- (1) `ndo_bpf` attach SKB-update BPF-prog that uses this map,
- and/or when (2) `ndo_bpf` XDP-prog being attached (that uses map)
  - Both cases, check BTF-format match or reject attach
- The map-refcnt, determines when to disable populate metadata again

Selecting metadata layouts

Driver can have multiple metadata-maps

- Identified via map-key as id
- (the map-value define metadata layout via BTF-format)
- Each map (likely) have different driver populate function associated

End

Disclaimer

- These slides are only design ideas and suggestions
- Non of this is actually implemented

Main purpose was getting a discussion going

- which were hopefully successful...

Slides: Extra

Layout of xdp_frame

If layout needs to be discussed...

```
struct xdp_frame {
        void *                  data;           /*     0     8 */
        u16                     len;            /*     8     2 */
        u16                     headroom;       /*    10     2 */
        u16                     metasize;       /*    12     2 */
        /* XXX 2 bytes hole, try to pack */
        struct xdp_mem_info     mem;            /*    16     8 */
        struct net_device *     dev_rx;         /*    24     8 */

        /* size: 32, cachelines: 1, members: 6 */
        /* sum members: 30, holes: 1, sum holes: 2 */
        /* last cacheline: 32 bytes */
};
```

Layout of xdp_buff

If layout needs to be discussed...

```
struct xdp_buff {
        void *                  data;               /*     0     8 */
        void *                  data_end;           /*     8     8 */
        void *                  data_meta;          /*    16     8 */
        void *                  data_hard_start;    /*    24     8 */
        long unsigned int       handle;             /*    32     8 */
        struct xdp_rxq_info *   rxq;                /*    40     8 */

        /* size: 48, cachelines: 1, members: 6 */
        /* last cacheline: 48 bytes */
};
```