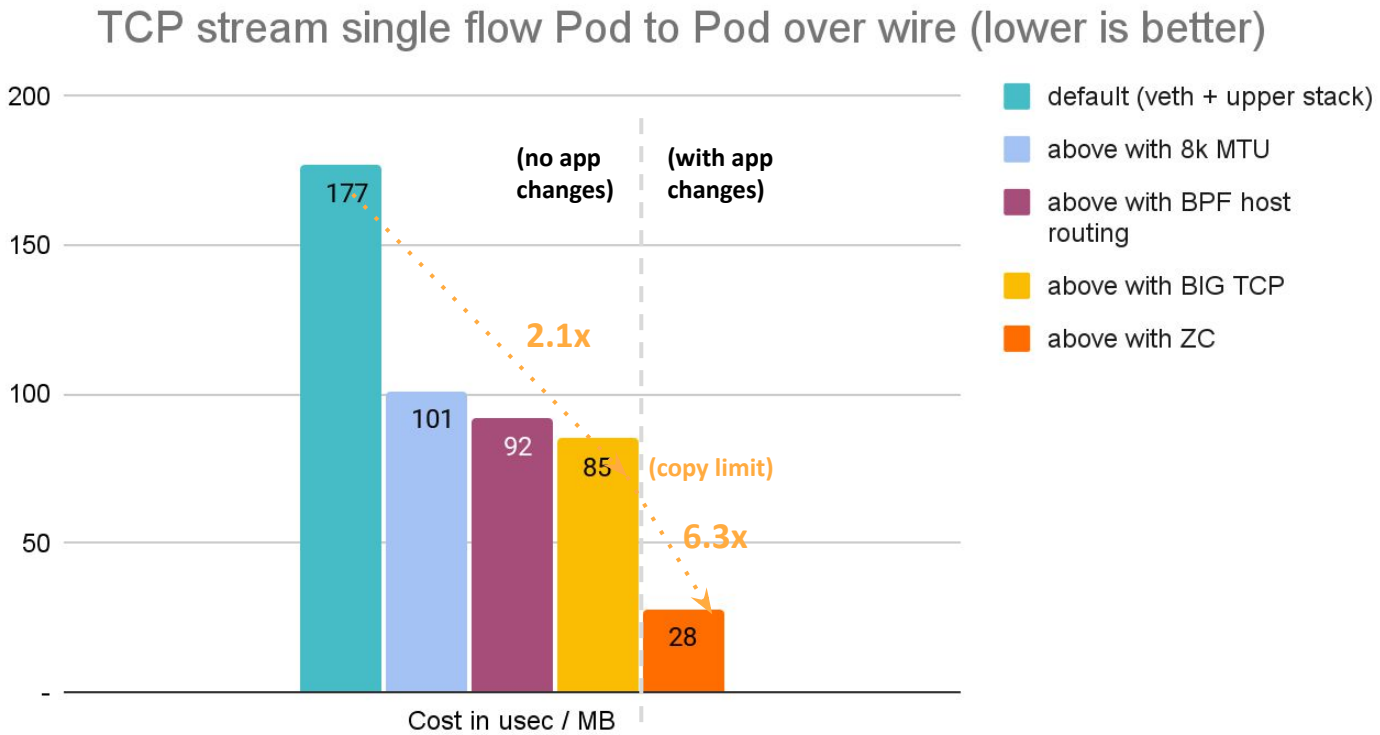
H/D split, BIG TCP & ZC, bpf_mprog for XDP

Daniel Borkmann (Isovalent)

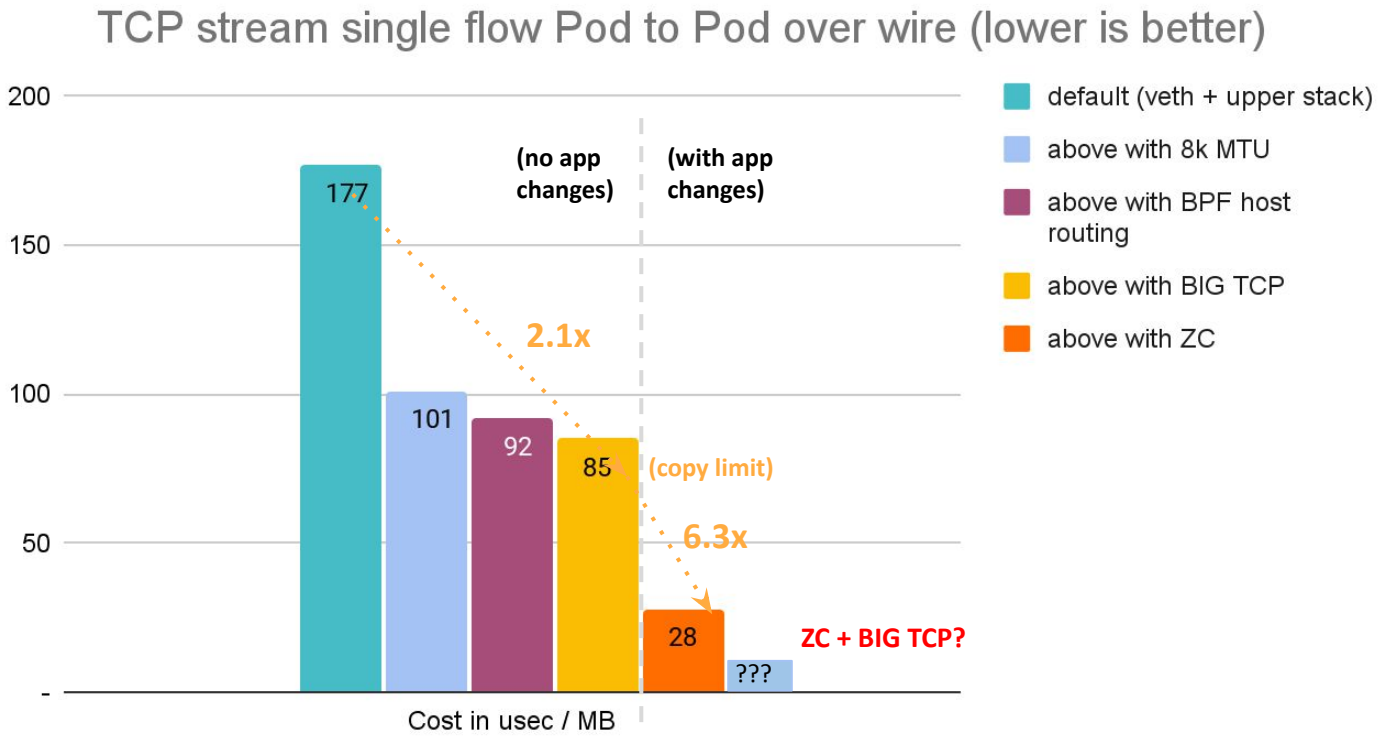
Netconf 2023

Experiment: Reducing cost to transfer large data



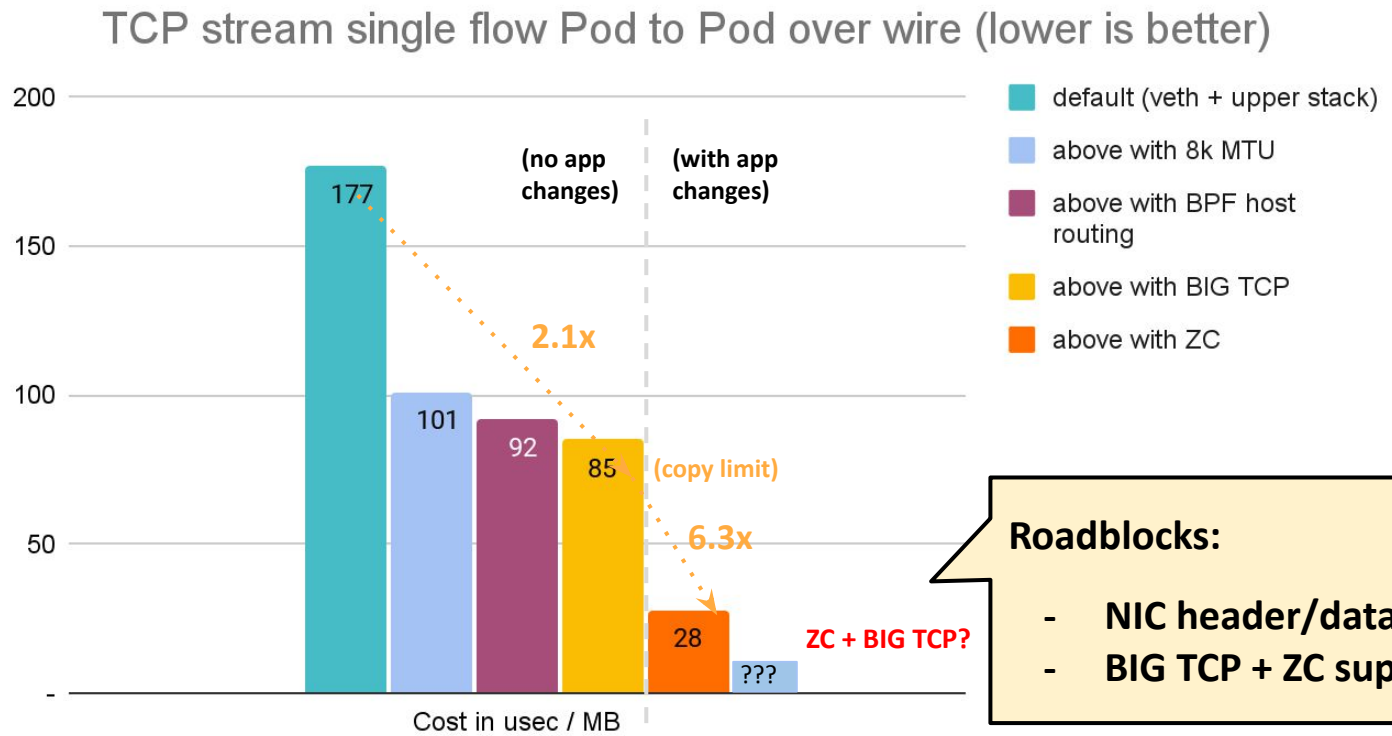
Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

Experiment: Reducing cost to transfer large data



Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

Experiment: Reducing cost to transfer large data



Roadblocks:

- **NIC header/data split for ZC**
- **BIG TCP + ZC support**

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

NIC header/data split

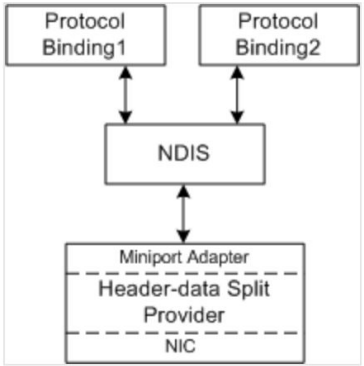
Header/data split: Ability for a NIC to dissect packets and place header and data into separate places

- Currently no generic facility to configure header/data split and not all NICs support it
- From Eric's netdevconf talk: "mlx4 does not have generic header-data split, but can use up to 4 segments to receive a packet. Each segment can be precisely sized."
- For testing we implemented a PoC for mlx5 to have first frag ETH + IPv6 + TCP so that next frags have PAGE_SIZE aligned payload for TCP RX ZC
  - PoC hack: `ethtool --set-priv-flags eth0 rx_striding_rq off`

The Path To TCP 4K MTU and RX ZeroCopy: https://legacy.netdevconf.info/0x14/session.html?talk-the-path-to-tcp-4k-mtu-and-rx-zerocopy

NIC header/data split

Header/data split: Brief look at Windows (NDIS 6.1+) - docs link / developer link



The miniport driver receives configuration information from NDIS to set up the NIC for header-data split receive operations. Also, the miniport driver exposes the NIC's services to NDIS for run-time operations such as send and receive operations.

A NIC that is capable of header-data split operations receives Ethernet frames and splits the headers and data into separate receive buffers.

[souce]

NIC header/data split

Header/data split: Brief look at Windows (NDIS 6.1+) - docs link / developer link

- Reporting of NIC capabilities:
  - NDIS_HD_SPLIT_CAPS_SUPPORTS_HEADER_DATA_SPLIT
  - NDIS_HD_SPLIT_CAPS_SUPPORTS_IPV4_OPTIONS
  - NDIS_HD_SPLIT_CAPS_SUPPORTS_IPV6_EXTENSION_HEADERS
  - NDIS_HD_SPLIT_CAPS_SUPPORTS_TCP_OPTIONS

- Current NIC config:
  - NDIS_HD_SPLIT_ENABLE_HEADER_DATA_SPLIT

NIC header/data split

Header/data split: Could potentially be realized as an ethtool setting?

- mlx4/mlx5 based NICs: precise sizing
- ice/i40e based NICs: docs indicate h/d split possible
- Ideally users have a facility to more easily consume RX ZC without changing driver code
- ethtool configuration?
    - Get NIC capabilities wrt h/d split
        - none/generic/static sizing
        - Are there commonalities between vendor caps wrt generic?
    - Get current info on how skb(/xdpbuff) is assembled
    - Set NIC h/d split with optional sizing/cutoff point

Thoughts/opinions/experience from practice/various NICs?

Other settings/caveats

Various settings need to be considered for RX ZC:

- mlx5 (mainly just our PoC): `ethtool --set-priv-flags eth0 rx_striding_rq off`
- MTU is set to 4168 (4k) or 8264 (8k), implicitly affects TCP ADVMSS
- For pinning TCP WSCALE the TCP rmem/wmem must be adapted e.g. `"4096 67108864 134217728"`
- For TX zero-copy optmem needs tuning: `sysctl net.core.optmem_max=1048576`
- Contention/overhead in IOMMU and page clearing: `iommu=off, init_on_alloc=0 init_on_free=0`
- Page recycling from page pool cannot be reused anymore

Other caveats:

- TCP zero-copy benefits might be limited if application needs to pull data into cache

Good example application for RX & TX TCP zero-copy is tcp_mmap in networking selftests.

BIG TCP & ZC

net: introduce a config option to tweak MAX_SKB_FRAGS

```
Currently, MAX_SKB_FRAGS value is 17.

For standard tcp sendmsg() traffic, no big deal because tcp_sendmsg()
attempts order-3 allocations, stuffing 32768 bytes per frag.

But with zero copy, we use order-0 pages.

For BIG TCP to show its full potential, we add a config option
to be able to fit up to 45 segments per skb.

This is also needed for BIG TCP rx zerocopy, as zerocopy currently
does not support skbs with frag list.

We have used MAX_SKB_FRAGS=45 value for years at Google before
we deployed 4K MTU, with no adverse effect, other than
a recent issue in mlx4, fixed in commit 26782aad00cc
("net/mlx4: MLX4_TX_BOUNCE_BUFFER_SIZE depends on MAX_SKB_FRAGS")
```

BIG TCP & ZC

MAX_SKB_FRAGS is a Kconfig option with range from 17 (default) upto 45

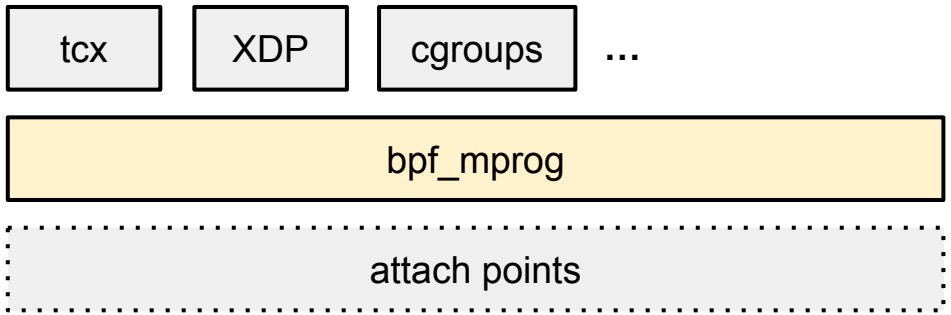
Regular distro users currently need to rebuild their kernel to bump MAX_SKB_FRAGS.

Potential options to lower the barrier?

- Bump MAX_SKB_FRAGS default
- Make MAX_SKB_FRAGS e.g. run/boot-time configurable
  - Lots of churn e.g. used in many places as array[MAX_SKB_FRAGS] inside a struct
- Support BIG TCP + ZC with frag list
- Other ideas?

XDP + bpf_mprog

- Dependency directives (can also be combined):
  - BPF_F_{BEFORE,AFTER} with relative_{fd,id} which can be {prog,link}
    - BPF_F_ID flag as {fd,id} toggle
    - BPF_F_LINK flag as {prog,link} toggle
- Support prog-based attach/detach and link API
- Internal revision counter and optionally being able to pass expected_revision
  - Daemon can query current state with revision, and pass it along for attachment to assert current state
- Common layer/API which deals which deals with all the details for state update



XDP + bpf_mprog (sketch)

- UAPI PoV:
  - Only native XDP for now
  - BPF links can be nicely integrated
  - What about netlink API?

```
1645  1645                        };
1646  1646                        __u64           expected_revision;
1647  1647                } tcx;
      1648  +             struct {
      1649  +                     union {
      1650  +                             __u32   relative_fd;
      1651  +                             __u32   relative_id;
      1652  +                     };
      1653  +                     __u64           expected_revision;
      1654  +             } xdp;
1648  1655                struct {
1649  1656                        __aligned_u64   path;
1650  1657                        __aligned_u64   offsets;
      ↓↑    @@ -6303,6 +6310,7 @@ enum xdp_action {
6303  6310          XDP_PASS,
6304  6311          XDP_TX,
6305  6312          XDP_REDIRECT,
      6313  +       XDP_NEXT,
6306  6314  };
6307  6315
6308  6316  /* user accessible metadata for XDP packet hook
```

XDP + bpf_mprog (sketch)

- Fast path:

```
493 + static __always_inline u32
494 + xdp_run(const struct bpf_mprog_entry *entry, struct xdp_buff *xdp)
495 + {
496 +         const struct bpf_mprog_fp *fp;
497 +         const struct bpf_prog *prog;
498 +         int ret = XDP_NEXT;
499 +
500 +         bpf_mprog_foreach_prog(entry, fp, prog) {
501 +                 ret = __bpf_prog_run(prog, xdp, BPF_DISPATCHER_FUNC(xdp));
502 +                 if (ret != XDP_NEXT)
503 +                         break;
504 +         }
505 +         if (static_branch_unlikely(&bpf_master_redirect_enabled_key)) {
506 +                 if (ret == XDP_TX && netif_is_bond_slave(xdp->rxq->dev))
507 +                         ret = xdp_master_redirect(xdp);
508 +         }
509 +         return ret;
510 + }
```

XDP + bpf_mprog (sketch, mlx5)

```
@@ -715,10 +715,11 @@ struct mlx5e_rq {
715 715         struct dim         dim; /* Dynamic Interrupt Moderation */
716 716
717 717         /* XDP */
718     -       struct bpf_prog __rcu *xdp_prog;
    718 +       struct bpf_mprog_entry __rcu *xdp_active;
719 719         struct mlx5e_xdpsq    *xdpsq;
720 720         DECLARE_BITMAP(flags, 8);
721 721         struct page_pool      *page_pool;
    722 +       struct bpf_mprog_bundle bundle;
722 723
723 724         /* AF_XDP zero-copy */
724 725         struct xsk_buff_pool  *xsk_pool;
```

```
264 264   /* returns true if packet was consumed by xdp */
265     - bool mlx5e_xdp_handle(struct mlx5e_rq *rq,
266     -                       struct bpf_prog *prog, struct mlx5e_xdp_buff *mxbuf)
    265 + bool mlx5e_xdp_handle(struct mlx5e_rq *rq, struct mlx5e_xdp_buff *mxbuf)
267 266   {
268 267           struct xdp_buff *xdp = &mxbuf->xdp;
269 268           u32 act;
270 269           int err;
271 270
272     -         act = bpf_prog_run_xdp(prog, xdp);
    271 +         act = xdp_run(rcu_dereference(rq->xdp_active), xdp);
273 272           switch (act) {
274 273           case XDP_PASS:
275 274                   return false;
```

XDP + bpf_mprog

- Every native XDP driver needs to be changed :/
  - Potentially gives opportunity to refactor more deeply and move common code into core
  - bpf_mprog rework might get more complex for drivers supporting also af_xdp in .ndo_bpf
- Ideally new XDP base support should become: XDP base actions + redirect + bpf_mprog
- Control plane needs to lock down bpf_mprog with regards to XDP mbuf vs linear mode to only being able to attach one flavor