Poll mode driver for XDP zero-copy
Why a new DPDK PMD for XDP?

- Hardware agnostic (PMD extension for non DPDK NICs).
- No custom driver (igb_uio) dependency.
- Offload meta data for user space.
- No change to DPDK applications or library (**).
- Support for full and partial buffer replace (**).
AGENDA

“How to make a secure, transparent, low overhead Poll Mode Driver for interfaces ranging from wired, wireless, and accelerators. Where management is on kernel; and Data path with user space.”

1. What is XDP?
2. How XDP Zero Copy DPDK is done?
3. Why External Buff?
4. Possible Use cases.
5. Future work.
6. Q & A
What is XDP?

**Definition:**
XDP (eXpress Data Path) is a technique for offloading packet processing to the network layer, bypassing the user-space application directly.

**Components:**
- **NIC:** Network Interface Card
- **RX Ring:** Receive ring buffer
- **TX Ring:** Transmit ring buffer
- **SKB:** Single-Buffered Kernel
- ** skb shared info:** Shared information within the kernel

**Diagram:**
- **Driver**:
  - `_rcv_ISR()`
  - RX DMA BUFFER
  - TX DMA BUFFER
  - `_hard_start_xmit()`
- **BPF ACTION**:
  - XDP_PASS
  - XDP_DROP
- **MAP for XDP sock**
- **User Buffer**:
- **Network Stack**
- **User Application**

**SKB frame:**
- `head`
- `data`
- `tail`
- `end`
- `len`

**XDP frames (page aligned):**
- XDP_TX
- XDP_REDIRECT
- XDP_PASS
- XDP_DROP
How XDP Zero Copy DPDK is done?
How XDP Zero Copy DPDK PMD is done?

- **eth_dev_probe**
  - Parse user args for XDP

- **eth_dev_configure**
  - `rte_malloc` - `NUM_FRAMES * XDP_FRAME_SIZE & getpagesize()`;
  - `setsockopt` - `XDP_UMEM_REG, XDP_UMEM_FILL_RING, XDP_UMEM_COMPLETION_RING, XDP_MMAP_OFFSETS`
  - `mmap` for `XDP_UMEM_PGOFF_FILL_RING & XDP_UMEM_PGOFF_COMPLETION_RING`

- **eth_rx_queue_setup**
  - `setsockopt` - `XDP_RX_RING`
  - `umem_fill_to_kernel` - `FILL_RING`

- **eth_tx_queue_setup**
  - `setsockopt` - `XDP_TX_RING`
  - `umem_fill_to_kernel` - `COMPLETION_RING`

- **eth_dev_start**
  - `sxdp_family = PF_XDP; sxdp_queue_id = rx_queue; sxdp_flags = XDP_ZEROCOPY; sxdp_ifindex = if_nametoindex(priv_data->k_iface_name);`
  - `xskx_map = bpf_map_get_fd_by_id`
  - `bpf_map_update_elem` - `key & priv_data->xsk_fd`
  - `bind(priv_data->xsk_fd, (struct sockaddr *)&sxdp, sizeof(sxdp))`

- **eth_dev_stop**
  - `xskx_map = bpf_map_get_fd_by_id(priv_data->xskx_map_id);`
  - `bpf_map_delete_elem(xskx_map, &key)`

---

vdev=net_xdpzc0,kiface=ens7,xskxmap="xsks_map", data_offset=256
How XDP Zero Copy DPDK is done?

- eth_xdp_zc_rx
  - xq_deq
  - rte_pktmbuf_alloc_bulk
  - rte_memcpy
  - umem_fill_to_kernel2

- eth_xdp_zc_tx
  - umem_complete_from_kernel
  - rte_memcpy
  - xq_enq
  - sendto(xsk_fd) //event notify
  - rte_pktmbuf_free
Why external mbuf?

memcpy

- **Pros:**
  - XDP Buffer are released to pool immediately after copy.

- **Cons:**
  - Large byte copy is multiple smaller copy.
  - HW is limited to 2 load & 1 store on vector.

mbuf pool

- **Pros:**
  - Buffer is in DPDK buffer format.
  - No copy or external buffer.

- **Cons:**
  - All buffers needs to be page aligned.
  - Applications needs to be adapted.
  - Buffer held in application till packet is dropped or tx complete.
How XDP Zero Copy DPDK is done?

- **eth_xdp_full_zc_rx**
  
  xq_deq
  
  rte_pktmbuf_alloc_bulk
  
  rte_pktmbuf_ext_shinfo_init_helper - xdp_buf_release_to_fq
  
  if (success) rte_pktmbuf_attach_extbuf
  
  else
  
  rte_memcpy

- **eth_xdp_full_zc_tx**
  
  umem_complete_from_kernel
  
  If (RTE_MBUF_HAS_EXTBUF(bufs[i]))
  
  { rtexx_pktmbuf_ext_shinfo_init_helper;
    rte_atomic64_exchange - mbuf->buf_addr with umem_buf }
  
  else
  
  {rte_memcpy}
  
  xq_enq, sendto
Performance

Throughput for 64B

<table>
<thead>
<tr>
<th>SCENARIOS</th>
<th>rx-tx (l2fwd)</th>
<th>tx drop</th>
<th>rx drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdp_zc_ (no offload)</td>
<td>9.7</td>
<td>10.5</td>
<td>10.8</td>
</tr>
<tr>
<td>xdp_zc</td>
<td>8.9</td>
<td>9.8</td>
<td>10.2</td>
</tr>
<tr>
<td>xdp_memcpy</td>
<td>6.9</td>
<td>7.9</td>
<td>8.5</td>
</tr>
<tr>
<td>igb_uio</td>
<td>14.8</td>
<td>14.9</td>
<td>14.9</td>
</tr>
</tbody>
</table>

Host CPU: Xeon Skylake
Guest CPU: 4 cores pinned
Guest OS: 4.20.0-042000-generic
DPDK: 18.11
NIC: 82599ES pass through
Version: 5.1.0k
Firmware: 0x000118f6
RX kernel thread on core 4
DPDK on core 2
Possible Use Cases

- No device specific user space driver (generic)
  - Allow more non DPDK NICs to be part of eco-system.
  - Support for wireless, and other accelerators.
- Using existing Kernel + XDP alternative to igb_uio.
  - Port ownership on Kernel.
  - PMD is a pipe for XDP.
- Security fixes found in driver is
  - Available as patches.
  - Not dependent on DPDK release cycles.
  - In tree build for driver.
- To use minimal change for userspace or DPDK applications.
  - No complicated FDIR rule sets for VF.
  - No change to mbuf or mempool library.
- Event dev offload
  - NIC HW RSS to be used as flow hash.
- Application Acceleration
  - OVS or vSwitch for Guest OS and Dockers
  - User space PTP and SYNCE with time stamp offload meta-data.
Future work

• In progress:
  • Integration to mainline. (after 19.05)
  • Stability and regression.

• Need to explore:
  • Support for Jumbo Frames (multi – segmented driver buffer)

• To Do:
  • Support for multiple queues. (19.11)
  • Extended stats. (19.05)
  • Applications enhanced for XDP ZC offload meta-data. (need help from community)
“Thanks All, Q & A?”

SCREEN SHOTS AS BACKUP
How AF_XDP works?

- sfd = socket(PF_XDP, SOCK_RAW, 0);
  buffs = calloc(num_buffs, FRAME_SIZE);
  setsockopt(sfd, SOL_XDP, XDP_MEM_REG, buffs);
  setsockopt(sfd, SOL_XDP, XDP_{RX|TX|FILL|COMPLETE}_RING, ring_size);
  mmap(..., sfd, ......); /* map kernel rings */
  bind(sfd, "/dev/eth0", queue_id,....);
  for (;;) {
    read_process_send_messages(sfd);
  };
How does DPDK PMD works?

- **_dev_configure**
  - Check user parameters.
  - Create private area.
  - Apply default or user configuration.

- **_queue_setup**

- **_dev_start | _dev_stop**
  - Per queue
    - buffer cache for DMA address.
    - Clean-up.
  - Destroy private area.

- **_rx_burst | _tx_burst**
  - Scalar or vector.
  - SW offload.
  - House keeping for DMA buffers
    - Buffer cache for next DMA regions for RX.
    - Clean up TX packet