AF_XDP copy mode needs more love

Netdev 0x1A Roma

Jason Xing <kernelxing@tencent.com>

Content

1. Brief Introduction
2. Copy Mode
3. Performance Optimization
4. Batch Xmit
5. “Issues”?

1. Brief Introduction

1. What is the AF_XDP?
   a. 4 queues pointing to UMEM serves TX/RX path
   b. bypass kernel technology
2. Known practices in production
   a. mtcp: [A Userspace Transport Stack Doesn't Have to Mean Losing Linux Processing]
   b. Alibaba: [libxudp]
3. References
   a. [Accelerating networking with AF_XDP]

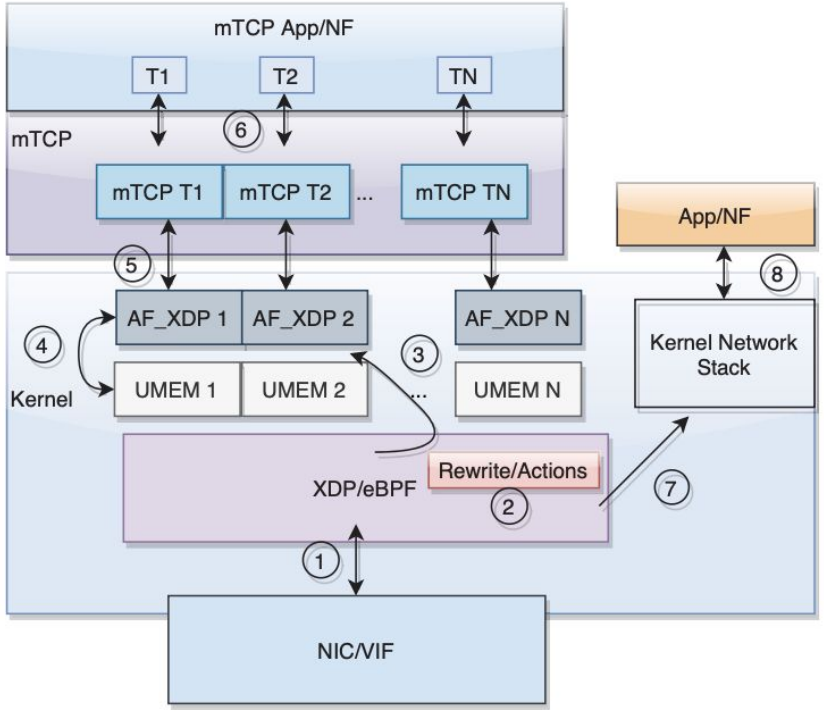


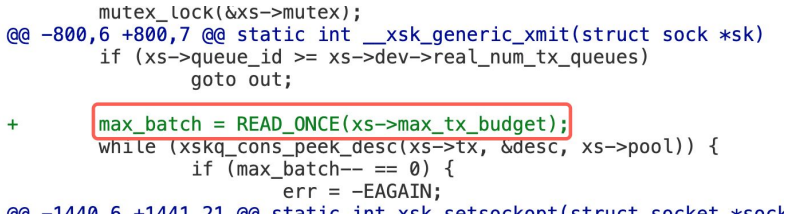
Figure 1: mTCP/AF XDP architecture

2. Copy Mode

1. Applicable Scenarios
    a. veth
    b. virtio_net
    c. some NICs that don’t support ZC mode so far.
        i. More interesting implementations about how to support in a easier/efficient way will be discussed in XDP workshop. Stay tuned!

3. Performance Optimization

1. net: xsk: introduce XDP_MAX_TX_SKB_BUDGET setsockopt
   a. The tunable value allows the maximum number of descriptors in tx queue to be sent in one send syscall.
   b. Prior value (TX_BATCH_SIZE) is 32 which led to excessive -EAGAIN returns that accounts for over 30% of total events. The initial value is 32. Incredible!
   c. In production, we observed a ~10% performance increase and a ~10% cpu% decrease after tuning it to 256.
   d. Usage
      i. setsockopt(...XDP_MAX_TX_SKB_BUDGET…)



```
        mutex_lock(&xs->mutex);
@@ -800,6 +800,7 @@ static int __xsk_generic_xmit(struct sock *sk)
        if (xs->queue_id >= xs->dev->real_num_tx_queues)
                goto out;

+       max_batch = READ_ONCE(xs->max_tx_budget);
        while (xskq_cons_peek_desc(xs->tx, &desc, xs->pool)) {
                if (max_batch-- == 0) {
                        err = -EAGAIN;
@@ -1440,6 +1441,21 @@ static int xsk_setsockopt(struct socket *sock
```

3. Performance Optimization

1. net: xsk: introduce XDP_MAX_TX_SKB_BUDGET setsockopt
2. xsk: use a smaller new lock for shared pool case
   a. Split cq_lock into two smaller locks: cq_prod_lock and cq_cached_prod_lock
   b. The core idea is to avoid frequently turning on/off irq in the hot path that is xsk_cq_reserve_locked()
   c. It yields a ~5% performance gain.

```
--- a/net/xdp/xsk.c
+++ b/net/xdp/xsk.c
@@ -548,12 +548,11 @@ static int xsk_wakeup(struct xdp_sock

 static int xsk_cq_reserve_locked(struct xsk_buff_pool *pool
 {
-       unsigned long flags;
        int ret;

-       spin_lock_irqsave(&pool->cq_lock, flags);
+       spin_lock(&pool->cq_cached_prod_lock);
        ret = xskq_prod_reserve(pool->cq);
-       spin_unlock_irqrestore(&pool->cq_lock, flags);
+       spin_unlock(&pool->cq_cached_prod_lock);

        return ret;
 }
@@ -566,7 +565,7 @@ static void xsk_cq_submit_addr_locked(st
        unsigned long flags;
        u32 idx;

-       spin_lock_irqsave(&pool->cq_lock, flags);
+       spin_lock_irqsave(&pool->cq_prod_lock, flags);
        idx = xskq_get_prod(pool->cq);
```

3. Performance Optimization

1. net: xsk: introduce XDP_MAX_TX_SKB_BUDGET setsockopt
2. xsk: use a smaller new lock for shared pool case
3. xsk: add indirect call for xsk_destruct_skb
   a. Add an indirect call for xsk
   b. The patch is inspired by Eric's optimization in UDP to minimize the impact of mitigation
   c. The performance number varies from 1%-2%

```
#include <trace/events/skb.h>
@@ -1140,12 +1141,13 @@ void skb_release_head_state(struct sk_buff *skb)
        if (skb->destructor) {
                DEBUG_NET_WARN_ON_ONCE(in_hardirq());
 #ifdef CONFIG_INET
-               INDIRECT_CALL_3(skb->destructor,
+               INDIRECT_CALL_4(skb->destructor,
                                tcp_wfree, __sock_wfree, sock_wfree,
+                               xsk_destruct_skb,
                                skb);
 #else
-               INDIRECT_CALL_1(skb->destructor,
-                               sock_wfree,
+               INDIRECT_CALL_2(skb->destructor,
+                               sock_wfree, xsk_destruct_skb,
                                skb);
```

3. Performance Optimization

1. net: xsk: introduce XDP_MAX_TX_SKB_BUDGET setsockopt
2. xsk: use a smaller new lock for shared pool case
3. xsk: add indirect call for xsk_destruct_skb
4. xsk: move cq_cached_prod_lock to avoid touching a cacheline in sending path
   a. move the position to avoid touch/write pool structure which brings performance affect.
   b. The idea is to avoid writing the structure that is mostly read-only.
   c. It improves a bit performance.

```
-   21.00%     1.06%  xudp_app        [kernel.
   - 19.94% __xsk_generic_xmit
      + 7.60% xsk_build_skb
      - 6.40% xsk_cq_reserve_locked
         - 6.26% _raw_spin_lock
              + 0.72% asm_common_interrupt
      + 4.75% __dev_direct_xmit
   + 0.78% __libc_sendto
```

```
@@ -90,11 +90,6 @@ struct xsk_buff_pool {
         * destructor callback.
         */
        spinlock_t cq_prod_lock;
-       /* Mutual exclusion of the completion
-        * Protect: when sockets share a singl
-        * and queue id is shared.
-        */
-       spinlock_t cq_cached_prod_lock;
        struct xdp_buff_xsk *free_heads[];
 };
```

```
@@ -46,6 +46,11 @@ struct xsk_queue {
        u64 invalid_descs;
        u64 queue_empty_descs;
        size_t ring_vmalloc_size;
+       /* Mutual exclusion of the completion
+        * Protect: when sockets share a singl
+        * and queue id is shared.
+        */
+       spinlock_t cq_cached_prod_lock;
 };
```

3. Performance Optimization

1. net: xsk: introduce XDP_MAX_TX_SKB_BUDGET setsockopt
2. xsk: use a smaller new lock for shared pool case
3. xsk: add indirect call for xsk_destruct_skb
4. xsk: move cq_cached_prod_lock to avoid touching a cacheline in sending path
5. net: advance skb_defer_disable_key check in napi_consume_skb
   a. net.core.skb_defer_max = 128
   b. net: add skb_defer_disable_key static key adds a static key to solve the performance regression of net: fix napi_consume_skb() with alien skbs. The idea is to not to free the skb on the cpu where it's allocated, but to free it on the current skb when the sysctl knob is set to 0.
   c. This patch avoid a second irq disabling/enabling operation.

```
void napi_consume_skb(struct sk_buff *skb, int budget)
{
        if (unlikely(!budget || !skb)) {
                dev_consume_skb_any(skb);
                return;
        }

        DEBUG_NET_WARN_ON_ONCE(!in_softirq());

        if (!static_branch_unlikely(&skb_defer_disable_key) &&
            skb->alloc_cpu != smp_processor_id() && !skb_shared(skb)) {
                skb_release_head_state(skb);
                return skb_attempt_defer_free(skb);
        }

        if (!skb_unref(skb))
                return;

        /* if reaching here SKB is ready to free */
        trace_consume_skb(skb, __builtin_return_address(0));

        /* if SKB is a clone, don't handle this case */
        if (skb->fclone != SKB_FCLONE_UNAVAILABLE) {
                __kfree_skb(skb);
                return;
        }

        skb_release_all(skb, SKB_CONSUMED);
        napi_skb_cache_put(skb);
}
```

```
static void kfree_skb_napi_cache(struct sk_buff *skb)
{
        /* if SKB is a clone, don't handle this case */
        if (skb->fclone != SKB_FCLONE_UNAVAILABLE) {
                __kfree_skb(skb);
                return;
        }

        local_bh_disable();
          napi_kfree_skb(skb, SKB_CONSUMED);
        local_bh_enable();
}
```

3. Performance Optimization

1. net: xsk: introduce XDP_MAX_TX_SKB_BUDGET setsockopt
2. xsk: use a smaller new lock for shared pool case
3. xsk: add indirect call for xsk_destruct_skb
4. xsk: move cq_cached_prod_lock to avoid touching a cacheline in sending path
5. net: advance skb_defer_disable_key check in napi_consume_skb
6. settings:
    a. net.core.skb_defer_max = 0
    b. setsockopt(...SO_SNDBUF…)
    c. grubby --update-kernel /boot/vmlinuz-xxx --args="mitigations=off"
    d. ethtool -G eth1 tx [value]
    e. taskset to set cpus (to use the same numa to send data)

4. Batch Xmit

1. batch idea
   a. xsk: batch xmit in copy mode
   b. It's mostly inspired from GSO/GRO

```
# Experiments
Tested on ixgbe at 10Gb/sec with the following settings:
1. mitigations off
2. ethtool -G enp2s0f1 tx 512
3. sysctl -w net.core.skb_defer_max=0
4. sysctl -w net.core.wmem_max=21299200 and sndbuf is the same value
5. XDP_MAX_TX_SKB_BUDGET 512

taskset -c 1 ./xdpsock -i enp2s0f1 -t  -S -s 64

copy mode(before):          1,801,007 pps (baseline)
AF_PACKET:                  1,375,808 pps (-23.6%)
zc mode:                   13,333,593 pps (+640.3%)
batch mode(batch 1):        1,976,821 pps (+9.8%)
batch mode(batch 64):       3,389,704 pps (+88.2%)
batch mode(batch 256):      3,387,563 pps (+88.0%)
```

4. Batch Xmit

1. AF_PACKET(1)
   a. AF_XDP is derived from tpacket v3 which at that point was called tpacket v4.
   b. packet_snd is more complicated than xsk_generic_xmit() which can be proved in the benchmark data.
   c. References:
      i. rfc Introducing AF_XDP support
      ii. v2 Introducing AF_XDP support
      iii. v3 Introducing AF_XDP support
      iv. Documentation/networking/packet_mmap.txt

```
static int packet_snd(struct socket *sock, struct msghdr *msg, size_t len)
{
        struct sock *sk = sock->sk;
        DECLARE_SOCKADDR(struct sockaddr_ll *, saddr, msg->msg_name);
        struct sk_buff *skb;
        struct net_device *dev;
        __be16 proto;
        unsigned char *addr = NULL;
        int err, reserve = 0;
        struct sockcm_cookie sockc;
        struct virtio_net_hdr vnet_hdr = { 0 };
        int offset = 0;
        struct packet_sock *po = pkt_sk(sk);
        int vnet_hdr_sz = READ_ONCE(po->vnet_hdr_sz);
        int hlen, tlen, linear;
        int extra_len = 0;

        /*
         *      Get and verify the address.
         */

        if (likely(saddr == NULL)) {
                dev     = packet_cached_dev_get(po);
                proto   = READ_ONCE(po->num);
        } else {
                err = -EINVAL;
                if (msg->msg_namelen < sizeof(struct sockaddr_ll))
                        goto out;
                if (msg->msg_namelen < (saddr->sll_halen + offsetof(struct
                        goto out;
                proto   = saddr->sll_protocol;
                dev = dev_get_by_index(sock_net(sk), saddr->sll_ifindex);
                if (sock->type == SOCK_DGRAM) {
                        if (dev && msg->msg_namelen < dev->addr_len +
                                   offsetof(struct sockaddr_ll, sll_addr))
                                goto out_unlock;
                        addr = saddr->sll_addr;
```

Figure 1: tpacket send path

4. Batch Xmit

1. AF_PACKET(2)
    a. tpacket v3 test
        i. send 1 packet at one time: **74w** pps
        ii. send 1 packet at one time with qdisc bypass: **110w** pps
        iii. send 64 packets in batch with qdisc bypass: **155w** pps
    b. af_xdp test
        i. send 64 packets in batch (qdisc bypass): **180w** pps

```c
if (batch_counter >= BATCH_SIZE) {
    if (sendto(sock_fd, NULL, 0, MSG_DONTWAIT, NULL, 0) < 0) {
        if (errno != ENOBUFS && errno != EAGAIN) {
            perror("sendto() 失败");
            break;
        }
    }
    batch_counter = 0;
}
```

Figure 1: how batch xmit works in benchmark

```
[root@TENCENT64 AF_XDP-example]# timeout 4 ./benchmark_tpacketv3 eth1
[15:25:29] PPS: 735800
[15:25:30] PPS: 748619
[15:25:31] PPS: 749800
```

```
[root@TENCENT64 AF_XDP-example]# timeout 5 ./benchmark_batch eth1
[11:09:02] PPS: 1083156
[11:09:03] PPS: 1113518
[11:09:04] PPS: 1115970
[11:09:05] PPS: 1104210
```

```
[root@TENCENT64 AF_XDP-example]# timeout 5 ./benchmark_batch eth1
[14:37:32] PPS: 1537664
[14:37:33] PPS: 1568576
[14:37:34] PPS: 1562688
[14:37:35] PPS: 1545968
```

Figure 2: test results

4. Batch Xmit

1. allocating skbs in batch
   a. [xsk: add xsk_alloc_batch_skb() to build skbs in batch]
   b. Four steps
      i. bulk allocation of skbs
      ii. for loop to init skbs
      iii. for loop to use the old build interface to complete the xsk style initialization
      iv. for loop to clean up skbs if needed

```
-   97.39%     0.02%  xdpsock  [kernel.kallsyms]
   - 97.37% xsk_sendmsg
      - 97.32% __xsk_sendmsg.constprop.0.isra.0
         - 96.17% __xsk_generic_xmit
            - 50.00% xsk_build_skb
               - 40.12% sock_alloc_send_pskb
                  + 21.45% alloc_skb_with_frags
                    12.62% skb_set_owner_w
                 2.61% skb_store_bits
                 1.96% memcpy_orig
            - 25.70% __dev_direct_xmit
```

Figure 5: perf output

```
skb_count += kmem_cache_alloc_bulk(net_hotdata.skbuff_cache,
                                   gfp_mask,
                                   nb_pkts - skb_count,
                                   (void **)&skbs[skb_count]);
```

Figure 1: allocate

```
skbuff_clear(skb);
data = kmalloc_reserve(&size, gfp_mask, node, skb);
if (unlikely(!data)) {
        *err = -ENOBUFS;
        break;
}
__finalize_skb_around(skb, data, size);
/* Replace skb_set_owner_w() with the following */
skb->sk = &xs->sk;
skb->destructor = sock_wfree;
```

Figure 2: init

```
skb = xsk_build_skb(xs, skb, &descs[j]);
```

Figure 3: build

```
while (k < i)
        kfree_skb(skbs[skb_count - 1 - k++]);
```

Figure 4: reclaim

4. Batch Xmit

1. small data transmission(1)
   a. xsk: cache data buffers to avoid frequently calling kmalloc_reserve
   b. kmalloc_reserve() consuming 40% cycles is a bit heavy, which was observed.
   c. The goal is take the fast path in kmalloc_reserve() out and allocate ‘data’ in batch.

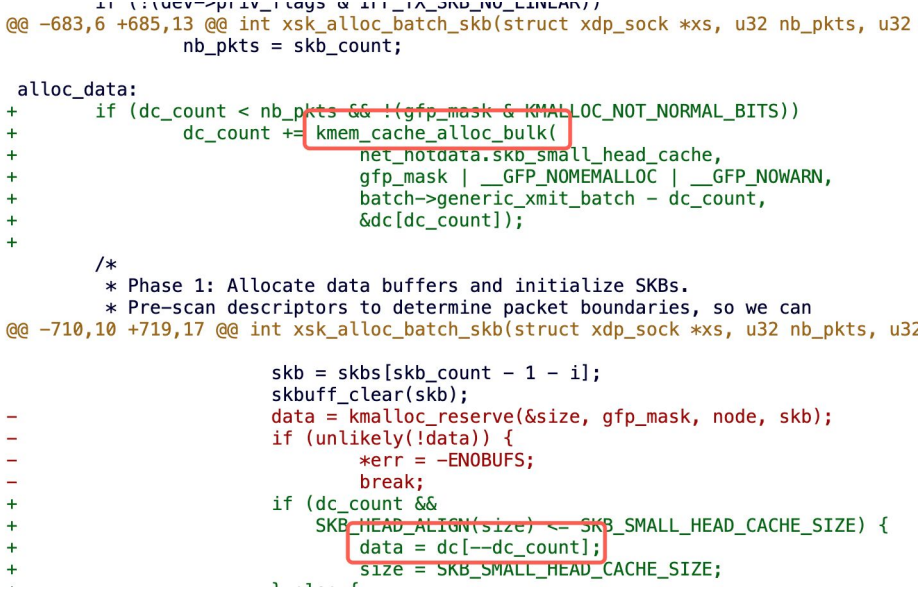
```
static void *kmalloc_reserve(unsigned int *size, gfp_t flags, int node,
                             struct sk_buff *skb)
{
        size_t obj_size;
        void *obj;

        obj_size = SKB_HEAD_ALIGN(*size);
        if (obj_size <= SKB_SMALL_HEAD_CACHE_SIZE &&
            !(flags & KMALLOC_NOT_NORMAL_BITS)) {
                obj = kmem_cache_alloc_node(net_hotdata.skb_small_head_cache,
                                            flags | __GFP_NOMEMALLOC | __GFP_NOWARN,
                                            node);
                *size = SKB_SMALL_HEAD_CACHE_SIZE;
                if (likely(obj))
                        goto out;
                /* Try again but now we are using pfmemalloc reserves */
                if (skb)
                        skb->pfmemalloc = true;
                return kmalloc_pfmemalloc(0, flags, node);
        }
```

```
 alloc_data:
+       if (dc_count < nb_pkts && !(gfp_mask & KMALLOC_NOT_NORMAL_BITS))
+               dc_count += kmem_cache_alloc_bulk(
+                               net_hotdata.skb_small_head_cache,
+                               gfp_mask | __GFP_NOMEMALLOC | __GFP_NOWARN,
+                               batch->generic_xmit_batch - dc_count,
+                               &dc[dc_count]);
+
```

Figure 1: small memory allocation in kmalloc_reserve

4. Batch Xmit

1. small data transmission(2)
   a. xsk: cache data buffers to avoid frequently calling kmalloc_reserve
   b. kmalloc_reserve() consuming 40% cycles is a bit heavy, which was observed.
   c. The goal is take the fast path in kmalloc_reserve() out and allocate ‘data’ in batch.
   d. Three steps
      i. dc_count = batch value
      ii. allocate/cache dc_count skbs
      iii. use the cached skb
   e. It improves ~10% overall performance in small packet transmission



```
@@ -683,6 +685,13 @@ int xsk_alloc_batch_skb(struct xdp_sock *xs, u32 nb_pkts, u32
                nb_pkts = skb_count;

 alloc_data:
+       if (dc_count < nb_pkts && !(gfp_mask & KMALLOC_NOT_NORMAL_BITS))
+               dc_count += kmem_cache_alloc_bulk(
+                               net_hotdata.skb_small_head_cache,
+                               gfp_mask | __GFP_NOMEMALLOC | __GFP_NOWARN,
+                               batch->generic_xmit_batch - dc_count,
+                               &dc[dc_count]);
+
        /*
         * Phase 1: Allocate data buffers and initialize SKBs.
         * Pre-scan descriptors to determine packet boundaries, so we can
@@ -710,10 +719,17 @@ int xsk_alloc_batch_skb(struct xdp_sock *xs, u32 nb_pkts, u32
                        skb = skbs[skb_count - 1 - i];
                        skbuff_clear(skb);
-                       data = kmalloc_reserve(&size, gfp_mask, node, skb);
-                       if (unlikely(!data)) {
-                               *err = -ENOBUFS;
-                               break;
+                       if (dc_count &&
+                           SKB_HEAD_ALIGN(size) <= SKB_SMALL_HEAD_CACHE_SIZE) {
+                               data = dc[--dc_count];
+                               size = SKB_SMALL_HEAD_CACHE_SIZE;
```

4. Batch Xmit

1. netdev_start_xmit in batch
   a. xsk: add direct xmit in batch function
   b. I was inspired by this patch, then I tried to make everything in batch which is now how the whole series looks like.
   c. Previously, when a skb is sent, we need to turn on/turn off the irq and also lock/unlock the TX LOCK, which is unnecessary.

```
+int xsk_direct_xmit_batch(struct xdp_sock *xs, struct net_device *dev)
+{
+       u16 queue_id = xs->queue_id;
+       struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_id);
+       struct sk_buff_head *send_queue = &xs->batch.send_queue;
+       int ret = NETDEV_TX_BUSY;
+       struct sk_buff *skb;
+
+       local_bh_disable();
+       HARD_TX_LOCK(dev, txq, smp_processor_id());
+       while ((skb = __skb_dequeue(send_queue)) != NULL) {
+               struct sk_buff *orig_skb = skb;
+               bool again = false;
+
+               skb = validate_xmit_skb_list(skb, dev, &again);
+               if (skb != orig_skb) {
+                       dev_core_stats_tx_dropped_inc(dev);
+                       kfree_skb_list(skb);
+                       ret = NET_XMIT_DROP;
+                       break;
+               }
+
+               if (netif_xmit_frozen_or_drv_stopped(txq)) {
+                       __skb_queue_head(send_queue, skb);
+                       break;
+               }
+               skb_set_queue_mapping(skb, queue_id);
+               ret = netdev_start_xmit(skb, dev, txq, false);
+               if (ret != NETDEV_TX_OK) {
+                       if (ret == NETDEV_TX_BUSY)
+                               __skb_queue_head(send_queue, skb);
+                       break;
+               }
+       }
+       HARD_TX_UNLOCK(dev, txq);
+       local_bh_enable();
```

4. Batch Xmit

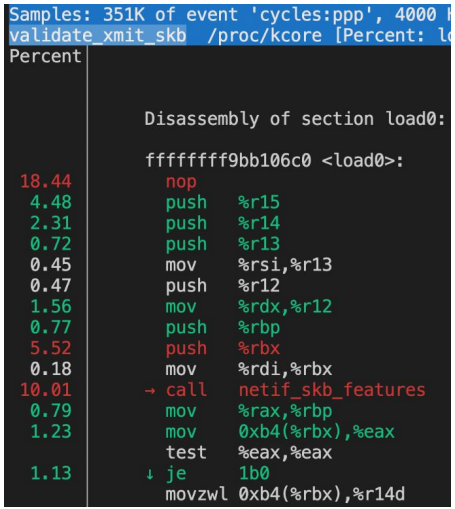
1. reduce hardware doorbells
   a. xsk: support dynamic xmit.more control for batch xmit
   b. This is suggested by Jesper
   c. The core idea is to minimize the times of kicking the hardware doorbell, which minimize the irq counts.
   d. It gives us a 26% performance increase.

```
@@ -4920,8 +4921,12 @@ int xsk_direct_xmit_batch(struct xdp_sock *xs,
                        __skb_queue_head(send_queue, skb);
                        break;
                }
+
+               if (!skb_peek(send_queue))
+                       more = false;
+
                skb_set_queue_mapping(skb, queue_id);
-               ret = netdev_start_xmit(skb, dev, txq, false);
+               ret = netdev_start_xmit(skb, dev, txq, more);
                if (ret != NETDEV_TX_OK) {
                        if (ret == NETDEV_TX_BUSY)
                                __skb_queue_head(send_queue, skb);
--
```

```
/* notify HW of packet */
if (netif_xmit_stopped(txring_txq(tx_ring)) || !netdev_xmit_more()) {
        writel(i, tx_ring->tail);
}
```

4. Batch Xmit

1. skip unnecessary validation
   a. xsk: try to skip validating skb list in xmit path
   b. I attempted to remove the whole function but I failed, which was pointed out by Paolo
      i. xsk: skip validating skb list in xmit path
      ii. The SG caes is inevitable, so I add the exception in xmit path.
   c. The performance goes up by ~5%.




```
diff --git a/net/core/dev.c b/net/core/dev.c
index a6abd621a7f3..aa38993b9dd4 100644
--- a/net/core/dev.c
+++ b/net/core/dev.c
@@ -4899,6 +4899,7 @@ int xsk_direct_xmit_batch(struct xdp_sock *xs, struct ne
        u16 queue_id = xs->queue_id;
        struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_id);
        struct sk_buff_head *send_queue = &xs->batch.send_queue;
+       bool need_validate = !(dev->features & NETIF_F_SG);
        int ret = NETDEV_TX_BUSY;
        struct sk_buff *skb;
        bool more = true;
@@ -4906,15 +4907,17 @@ int xsk_direct_xmit_batch(struct xdp_sock *xs, struct
        local_bh_disable();
        HARD_TX_LOCK(dev, txq, smp_processor_id());
        while ((skb = __skb_dequeue(send_queue)) != NULL) {
-               struct sk_buff *orig_skb = skb;
-               bool again = false;
-
-               skb = validate_xmit_skb_list(skb, dev, &again);
-               if (skb != orig_skb) {
-                       dev_core_stats_tx_dropped_inc(dev);
-                       kfree_skb_list(skb);
-                       ret = NET_XMIT_DROP;
-                       break;
+               if (unlikely(need_validate)) {
+                       struct sk_buff *orig_skb = skb;
+                       bool again = false;
+
+                       skb = validate_xmit_skb_list(skb, dev, &again);
+                       if (skb != orig_skb) {
+                               dev_core_stats_tx_dropped_inc(dev);
+                               kfree_skb_list(skb);
+                               ret = NET_XMIT_DROP;
+                               break;
+                       }
                }

                if (netif_xmit_frozen_or_drv_stopped(txq)) {
--
```

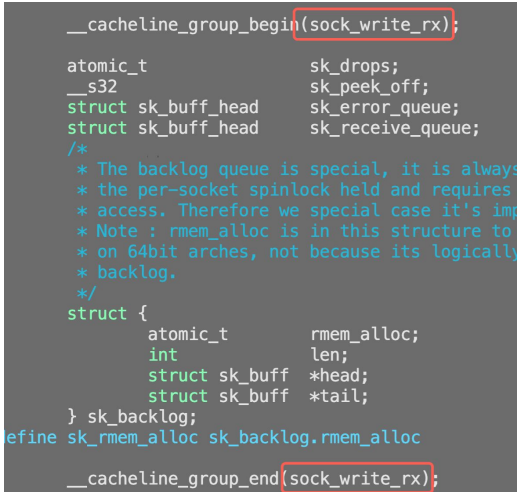
4. Batch Xmit

1. generic main batch logic
   a. xsk: support batch xmit main logic
   b. All in batch
      i. **peek** nb_descs descriptors in tx queue
      ii. **reserve** nb_descs slots in completion queue
      iii. **read** nb_descs descriptors in tx queue
      iv. **allocate** nb_pkts packets
         1. nb_pkts <= nb_descs
         2. skbs is put in the send_queue
      v. **send** packets to the driver

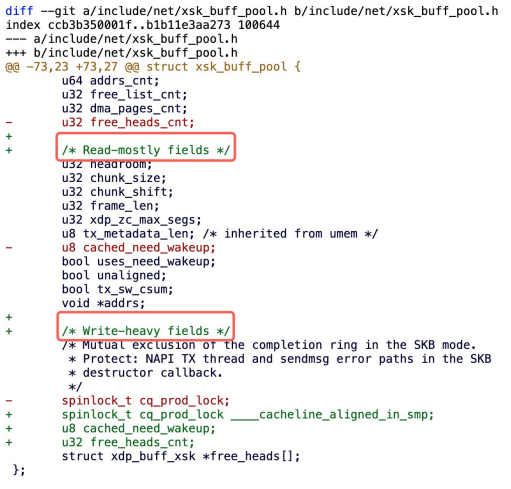
```
+
+       for (i = 0; i < max_budget; i += cons_descs) {
+               u32 nb_pkts = 0;
+               u32 nb_descs;
+
+               nb_descs = min(max_batch, max_budget - i);
+               nb_descs = xskq_cons_nb_entries(xs->tx, nb_descs);
+               if (!nb_descs)
+                       goto out;
+
+               /* This is the backpressure mechanism for the Tx path. Try to
+                * reserve space in the completion queue for all packets, but
+                * if there are fewer slots available, just process that many
+                * packets. This avoids having to implement any buffering in
+                * the Tx path.
+                */
+               nb_descs = xsk_cq_reserve_locked(pool, nb_descs);
+               if (!nb_descs) {
+                       err = -EAGAIN;
+                       goto out;
+               }
+
+               cons_descs = xskq_cons_read_desc_batch_copy(xs->tx, pool, descs,
+                                                           nb_descs, &nb_pkts);
+               if (cons_descs < nb_descs) {
+                       u32 delta = nb_descs - cons_descs;
+
+                       xsk_cq_cancel_locked(pool, delta);
+                       xs->tx->queue_empty_descs += delta;
+                       if (!cons_descs) {
+                               err = -EAGAIN;
+                               goto out;
+                       }
+                       nb_descs = cons_descs;
+               }
+
```

4. Batch Xmit

1. reorganize the hot structure
   a. xsk: separate read-mostly and write-heavy fields in xsk_buff_pool
   b. In LPC 2025 (Japan), Paolo suggested me to use perf c2c to find the hot spot of the structures in xsk. It turned out there is one outstanding spot!
   c. Reorganize them into two types: tx and rx. Now the only one hot spot is gone.
   d. This reorganization improves overall performance by 5-6%.



```
__cacheline_group_begin(sock_write_rx);

atomic_t                sk_drops;
__s32                   sk_peek_off;
struct sk_buff_head     sk_error_queue;
struct sk_buff_head     sk_receive_queue;
/*
 * The backlog queue is special, it is always
 * the per-socket spinlock held and requires
 * access. Therefore we special case it's imp
 * Note : rmem_alloc is in this structure to
 * on 64bit arches, not because its logically
 * backlog.
 */
struct {
        atomic_t        rmem_alloc;
        int             len;
        struct sk_buff  *head;
        struct sk_buff  *tail;
} sk_backlog;
efine sk_rmem_alloc sk_backlog.rmem_alloc

__cacheline_group_end(sock_write_rx);
```



```
diff --git a/include/net/xsk_buff_pool.h b/include/net/xsk_buff_pool.h
index ccb3b350001f..b1b11e3aa273 100644
--- a/include/net/xsk_buff_pool.h
+++ b/include/net/xsk_buff_pool.h
@@ -73,23 +73,27 @@ struct xsk_buff_pool {
        u64 addrs_cnt;
        u32 free_list_cnt;
        u32 dma_pages_cnt;
-       u32 free_heads_cnt;
+
+       /* Read-mostly fields */
        u32 headroom;
        u32 chunk_size;
        u32 chunk_shift;
        u32 frame_len;
        u32 xdp_zc_max_segs;
        u8 tx_metadata_len; /* inherited from umem */
-       u8 cached_need_wakeup;
        bool uses_need_wakeup;
        bool unaligned;
        bool tx_sw_csum;
        void *addrs;
+
+       /* Write-heavy fields */
        /* Mutual exclusion of the completion ring in the SKB mode.
         * Protect: NAPI TX thread and sendmsg error paths in the SKB
         * destructor callback.
         */
-       spinlock_t cq_prod_lock;
+       spinlock_t cq_prod_lock ____cacheline_aligned_in_smp;
+       u8 cached_need_wakeup;
+       u32 free_heads_cnt;
        struct xdp_buff_xsk *free_heads[];
 };
```

4. Batch Xmit

1. misc minor changes
   a. xsk: optimize xsk_build_skb for batch copy-mode fast path
   b. changes
      i. Replace skb_store_bits with memcpy
      ii. prefetch

```
diff --git a/net/core/skbuff.c b/net/core/skbuff.c
index 5726b1566b2b..bef5270e6332 100644
--- a/net/core/skbuff.c
+++ b/net/core/skbuff.c
@@ -752,14 +752,28 @@ int xsk_alloc_batch_skb(struct xdp_sock *xs, u32
        if (total_truesize)
                refcount_add(total_truesize, &xs->sk.sk_wmem_alloc);

-       /* Phase 3: Build SKBs with packet data */
+       /* Phase 3: Build SKBs with packet data. */
+       struct xsk_buff_pool *pool = xs->pool;
+       void *pool_addrs = pool->addrs;
+       bool unaligned = pool->unaligned;
+
        for (j = 0; j < alloc_descs; j++) {
+               u64 addr = descs[j].addr;
+               void *buffer;
+
+               if (unaligned)
+                       addr = xp_unaligned_add_offset_to_addr(addr);
+               buffer = pool_addrs + addr;
+
+               if (j + 1 < alloc_descs)
+                       prefetch(pool_addrs + descs[j + 1].addr);
+
                if (!xs->skb) {
                        skb = skbs[skb_count - 1 - k];
                        k++;
                }

-               skb = xsk_build_skb(xs, skb, &descs[j]);
+               skb = xsk_build_skb(xs, skb, &descs[j], buffer);
                if (IS_ERR(skb)) {
                        *err = PTR_ERR(skb);
                        break;
```

```
        buffer = xsk_buff_raw_get_data(xs->pool, d

(!skb) {
        hr = max(NET_SKB_PAD, L1_CACHE_ALIGN(dev->
@ struct sk_buff *xsk_build_skb(struct xdp_sock

        skb_reserve(skb, hr);
        skb_put(skb, len);

        err = skb_store_bits(skb, 0, buffer, len);
        if (unlikely(err))
                goto free_err;
        memcpy(skb->data, buffer, len);

        xsk_skb_init_misc(skb, xs, desc->addr);
        if (desc->options & XDP_TX_METADATA) {
```

5. “Issues”?

1. Performance wise
   a. unavoidable xsk_generic_xmit_cache() which is designed to solve multi-buffer issue
2. Maintenance wise
   a. multi-buffer has already messed up the core sending path:
      i. [PATCH net 0/7] xsk: fix AF_XDP multi-buffer Tx descriptor reclaim
      ii. xsk: cache csum_start/csum_offset to fix TOCTOU in xsk_skb_metadata()
      iii. xsk: fix u64 descriptor address truncation on 32-bit architectures
      iv. xsk: fix xsk_addrs slab leak on multi-buffer error path
      v. xsk: avoid skb leak in XDP_TX_METADATA case
      vi. xsk: prevent CQ desync when freeing half-built skbs in xsk_build_skb()
      vii. xsk: fix use-after-free of xs->skb in xsk_build_skb() free_err path
      viii. xsk: handle NULL dereference of the skb without frags issue
      ix. xsk: free the skb when hitting the upper bound MAX_SKB_FRAGS
      x. xsk: reject sw-csum UMEM binding to IFF_TX_SKB_NO_LINEAR devices
      xi. net: advance skb_defer_disable_key check in napi_consume_skb
      xii. …
3. Leave all the discussion in XDP workshop?

Thank you!