Adding packet queueing to XDP

Toke Høiland-Jørgensen
Principal Kernel Engineer, Red Hat

Linux Plumbers Conference
September 2022
EXCUSE ME, SIR

DO YOU HAVE A FEW MINUTES TO TALK ABOUT QUEUEING?
Why does XDP need queueing?

100->10 Gbps rate transition, 10ms base RTT
Why does XDP need queueing? (cont)

Other use cases enabled by queueing:

- Different **packet scheduling** policies (FQ, QoS, etc)
- Bandwidth **shaping**
- Network emulation (reordering, delaying packets)
- Custom **buffering schemes** (e.g., hold on to packets while spawning container)
Review: How does XDP_REDIRECT work?

Driver (RX)

Packet flow

- XDP program
- bpf_redirect_map()
- Populate bpf_redirect_info (per-CPU)
- dev_map_enqueue()
- cpu_map_enqueue()
- __xsk_map_redirect()
- XDP_REDIRECT
- xdp_do_redirect()
- xdp_do_flush()
- Bulk queue (16 pkts)

RX softirq (NAPI)

Loop over batch of packets

Bulk queue

Driver (TX)

Packet flow

- ndo_xdp_xmit()
Review: Netstack forwarding flow (simplified)

Driver (RX)

- build_skb()
- netif_receive_skb()
- Protocol handlers
  - Routing

RX softirq (NAPI)

- dev_queue_xmit()
- dev_qdisc_enqueue()
- __netif_schedule()

- raise_softirq(NET_TX_SOFTIRQ)
- qdisc
- qdisc_watchdog()

TX softirq

- net_tx_action()
- qdisc_run()
- dequeue_skb()
- netdev_start_xmit()

Driver (TX)

- ndo_start_xmit()
The ingredients we need

- Somewhere to store packets
- A way to schedule dequeue and transmission
Data structures for packet queues

How many different data structures do these 38 qdiscs use?
Data structures for packet queues

How many different data structures do these 38 qdiscs use?

```c
struct sk_buff {
    union {
        struct {
            struct sk_buff *next;
            struct sk_buff *prev;
        }
        union {
            struct net_device *dev;
            unsigned long dev_scratch;
        }
    };
    struct rb_node rbnode; /* used in netem, ip4 defrag, and tcp stack */
    struct list_head list;
    struct llist_node ll_node;
};
/* [... ] */
```
Proposed BPF API

For BPF the natural interface for storing packets is in a map.

- Implement a **priority queue** map type for packets (can also be used as FIFO)
- Allow XDP programs to queue packets with `bpf_redirect_map(queue, priority)`
- Create a new `bpf_packet_dequeue()` helper to pull packets out
  - Returns `PTR_TO_BTF_ID` of struct `xdp_md` which can be used like the XDP context

See example code on later slide.
Performance overhead of map types

- Baseline redirect: 36 ns
- FIFO: 22 ns
- PIFO: 44 ns
- RB-tree

Adding packet queuing to XDP - Toke Høiland-Jørgensen <toke@redhat.com>
Where is the RB-tree overhead coming from?

1. Identify the overhead points from the given code snippet.

2. Analyze the RB-tree operations and their costs.

3. Compare the costs with the theoretical expectations.

4. Discuss any anomalies or inefficiencies observed.

5. Propose strategies to mitigate the overhead.

---

**Example Excerpt:**

```c
parent = rb_parent(node);
node = parent;
rb_set_black(parent);
RB_RED);
rb_set_parent_color(sibling, parent, RB_RED);
```

**Explanation:**

- **Identifying Overhead Points:** The overhead is primarily due to the `rb_set_black` and `rb_set_parent_color` operations.
- **Analyzing Operations:** Each `rb_set_black` and `rb_set_parent_color` operation has a constant time complexity.
- **Comparing Costs:** The theoretical cost of an RB-tree operation is usually logarithmic in the number of nodes.
- **Anomalies:** The overhead appears to be proportional to the number of operations, which is unexpected for a balanced tree.
- **Mitigation Strategies:** Implementing a more efficient tree structure or optimizing the current structure could reduce the overhead.

---

**Further Reading:**

- Understanding Red-Black Trees
- Performance Optimization Techniques for Tree Structures
- Comparative Analysis of Tree Data Structures

---

**Image Description:**

- Diagrams illustrating RB-tree operations and their overhead.
- Code snippets highlighting critical operations and their costs.

---

**Contact:**

Toke Høiland-Jørgensen <toke@redhat.com>
Sidetrack: PIFO queues

In the literature, the Push-In, First-Out (PIFO) queue appeared in 2016\(^0\).

- It’s a **limited** priority queue (only dequeue at head)
  - Can be implemented in silicon
- We don’t need to limit ourselves to the PIFO, **however**:
  - We can use an **optimised algorithm** for software by Saeed et al\(^1\)

\(^0\) Sivaraman et al, 2016: “Programmable Packet Scheduling at Line Rate”
\(^1\) Saeed et al, 2019: “Eiffel: Efficient and Flexible Packet Scheduling”
The Eiffel PIFO algorithm

Figure 3 from Saeed et al (2019)
The Eiffel PIFO: rotating queues

![Diagram of Eiffel PIFO]

- Adding packet queuing to XDP
- Toke Høiland-Jørgensen <toke@redhat.com>
Performance overhead of map types (again)

- Baseline redirect: 36 ns
- FIFO: 22 ns
- PIFO: 44 ns

Adding packet queueing to XDP

Toke Høiland-Jørgensen <toke@redhat.com>
Data structures: Summary

- We need a data structure (BPF map) to store packets
  - Current qdiscs only use two data structures: FIFO and priority queue
  - A priority queue can be used as a FIFO, so really only one
- API: `bpf_redirect_map()` to enqueue, `add bpf_packet_dequeue()`
- The Eiffel PIFO algorithm performs well
  - Drawback: Priority range is fixed / only growing
  - Is this API limitation acceptable?
Recall: The ingredients we need

- Somewhere to store packets
- A way to schedule dequeue and transmission
TX hook attempt 1: dequeue hook

New **xdp dequeue** program type

- Can be attached to an interface (like XDP program)
- **Returns** a packet to transmit
- Stack calls `ndo_xdp_xmit()` with batch of packets

Submitted as RFC series: [https://lore.kernel.org/r/20220713111430.134810-1-toke@redhat.com](https://lore.kernel.org/r/20220713111430.134810-1-toke@redhat.com)
TX scheduling attempt 1 - replicate netstack

Driver (RX)
- build_skb()
- netif_receive_skb()

Packet flow

RX softirq (NAPI)
- `__netif_schedule()`
- dev_qdisc_enqueue()
- dev_queue_xmit()

Control flow

Protocol handlers
Routing

New helper
New map type
New program hook

TX softirq
- `ndo_start_xmit()`
- `netdev_start_xmit()`
- `net_tx_action()`
- `qdisc_run()`
- dequeue_skb()

Driver (TX)

A d d i n g  p a c k e t  q u e u i n g  t o  X D P

Toke Høiland-Jørgensen <toke@redhat.com>
```c
struct pifo_map {
    __uint (type, BPF_MAP_TYPE_PIFO_XDP);
    __uint (key_size, sizeof(__u32));
    __uint (value_size, sizeof(__u32));
    __uint (max_entries, 10240);
    __uint (map_extra, 8192); /* range */
} pifo SEC(".maps");

SEC("xdp")
int xdp_redirect_map_queue(struct xdp_md *ctx)
{
    int ret;
    ret = bpf_redirect_map(&pifo, 0, 0);

    if (ret == XDP_REDIRECT)
        bpf_schedule_iface_dequeue(ctx, tgt_ifindex, 0);

    return ret;
}

SEC("xdp_dequeue")
void *xdp_redirect_deq_func(struct dequeue_ctx *ctx)
{
    struct xdp_md *pkt;
    __u64 prio = 0;

    pkt = (void *)bpf_packet_dequeue(ctx, &pifo, 0, &prio);

    if (!pkt)
        return NULL;

    return pkt;
}
```
TX hook attempt 1: Problems

Problem: The maintainers didn’t like it

This feature can be done similar to hid-bpf without cast-in-stone uapi and hooks. Such patches would be much easier to land and iterate on top. The amount of bike shedding will be 10 times less. No need for new program type, no new hooks, no new FDs and attach uapi-s.

Alexei in
https://lore.kernel.org/r/20220715011228.tujkugafv6eixbyz@MacBook-Pro-3.local

Turns out he was (almost) right! As seen by attempt 2...
TX hook attempt 2: Use bpf_timers

Driver (RX)

- build_skb()
- netif_receive_skb()
- Protocol handlers
- Routing

RX softirq (NAPI)

- dev_queue_xmit()
- dev_qdisc_enqueue()
- __netif_schedule()
- raise_softirq(NET_TX_SOFTIRQ)

bpf_timer callback

- qdisc_watchdog()

TX softirq

- qdisc_run()
- dequeue_skb()

- netdev_start_xmit()
- net_tx_action()

Driver (TX)

- ndo_start_xmit()
```c
__u64 num_queued = 0;
SEC("xdp")
int xdp_redirect_map_timer(struct xdp_md *ctx)
{
    struct bpf_timer *timer;
    int ret, array_key = 0;
    timer = bpf_map_lookup_elem(&timermap, &array_key);
    if (!timer)
        return XDP_ABORTED;
    if (!timer_init) {
        bpf_timer_init(timer, &timermap,
                       CLOCK_MONOTONIC);
        bpf_timer_set_callback(timer,
                                xdp_timer_cb);
        timer_init = 1;
    }
    ret = bpf_redirect_map(&pifo, 0, 0);
    if (ret == XDP_REDIRECT) {
        num_queued++;
        bpf_timer_start(timer,
                        0 /* call asap */, 0);
    }
    return ret;
}
#define BATCH_SIZE 128
static int xdp_timer_cb(void *map, int *key,
                        struct bpf_timer *timer)
{
    struct xdp_md *pkt;
    __u64 prio = 0;
    int i;
    for (i = 0; i < BATCH_SIZE; i++) {
        pkt = (void *)bpf_packet_dequeue_xdp(&pifo, 0,
                                              &prio);
        if (!pkt)
            break;
        num_queued--;
        bpf_packet_send(pkt, tgt_ifindex, 0);
    }
    bpf_packet_flush();
    if (num_queued)
        bpf_timer_start(timer,
                        0 /* call asap */, 0);
    return 0;
}
```

Adding packet queueing to XDP - Toke Høiland-Jørgensen <toke@redhat.com>
Problem: Overhead of bpf_timer

- Adding packet queueing to XDP
  - Toke Høland-Jørgensen <toke@redhat.com>
TX hook attempt 2: Problems

The callback approach seems promising, but has a few problems:

- **Performance of `bpf_timer`**
  - Overhead (previous slide)
  - Risk of blocking other timer work (?)
  - Replace by generic callback feature as discussed in https://lore.kernel.org/r/cover.1657576063.git.delyank@fb.com?

- **No pushback from driver**
  - How does the BPF program know that the interface is busy?
  - With TX hook stack can keep packets around, what does BPF do?
### Summary: XDP queueing design

Adding packet queueing to XDP - Toke Høiland-Jørgensen <toke@redhat.com>

**Driver (RX)**
- XDP program
  - XDP_REDIRECT
  - xdp_do_redirect()
  - xdp_do_flush()

**RX softirq (NAPI)**
- bpf_redirect_map()
- bpf_timer_start()

**Loop over batch of packets**

**Control flow**

**Packet flow**

**PIFO map**
- pifo_map_enqueue()
- dev_map_enqueue()
- cpu_map_enqueue()
- __xsk_map_redirect()

**Timer softirq**
- bpf_timer_start()
- htimer

**Driver (TX)**
- bpf_timer_callback
- bpf_packet_dequeue()
- bpf_packet_send()
- ndo_xdp_xmit()
End: Questions?

WiP code:

- Kernel patches (implementing both TX hook approaches): https://git.kernel.org/toke/l/xdp-queueing-07
- Test framework for queueing algorithms: https://github.com/xdp-project/bpf-examples/pull/40 (by my PhD student Freysteinn Alfredsson)

Many thanks to Kumar Kartikeya Dwivedi, Jesper Brouer, Anna Brunstrom and Per Hurtig, as well as everyone who reviewed the RFC patchset.
Bonus slide: BPF qdisc

There’s a separate BPF qdisc proposal being worked on by Cong Wang.
Latest RFC: https://lore.kernel.org/r/20220602041028.95124-1-xiyou.wangcong@gmail.com

This is complementary to queueing in XDP - not in competition.

- BPF qdisc for packets going through the stack, XDP queueing is for bypassing the stack when forwarding
- Can hopefully share BPF map type and helpers
- BPF code reuse will likely be similar to TC-BPF/XDP (i.e., some effort required)
For good forwarding performance, splitting work among CPUs is essential.

- For XDP, this is up to the BPF program.
  - All callbacks will be on the same CPU
  - Steering can be done today using cpumap, see: https://github.com/xdp-project/xdp-cpumap-tc
- Possible optimisation: Bind map to particular CPU to elide locking