Shared Memory Pool for Representors

William Tu, Michal Swiatkowski, and Yossi Kuperman
Nvidia and Intel
witu@nvidia.com, michal.swiatkowski@intel.com, yossiku@nvidia.com
NetDev 0x18, CA, USA, July 15th-19th, 2024

Abstract

Representor is a special object that controls the slow path of switchdev virtual port, or vport (the representee). When scaling to thousands of vports, a corresponding thousands of representor netdevs are created and thousands of representor netdevs becomes resource heavy, especially for SmartNICs or DPUs (Data Processing Unit). An Nvidia Bluefield-2 contains 16GB of memory, but creating one thousand representor netdevs can use more than 10GB of RX memory buffer. Existing solutions save memory by sharing the RXQs of a Physical Function among all other reprensetors. However, this approach suffers from unfairness: a single VM can easily monopolize all RXQs, causing no RX buffers available for other VMs. The paper proposes a design with adjustable RXQ depth using a shared page pool. Our result shows that we can save 50% of memory with less than 10% of performance impact.

Keywords

SmartNIC/DPU, Switchdev, Page Pool, Receive Queues

Introduction

Modern network devices support advanced switching and offloading capabilities, called switchdev mode. In switchdev mode, users can create multiple virtual ports, or vports, through sysfs and assign vports to virtual machines or containers. The switchdev mode includes PFs, the Primary Functions of a PCIe device, with full access to the device's capabilities. PFs manages multiple types of vports, including VFs (Virtual Functions) and SFs (Sub/Scalable Functions).

As shown in Figure 1, the switchdev design follows the split model: a fast-path and slow-path. A NIC with advanced offload capabilities and flow table rules is the fast-path, while the corresponding software switch that handles initial flow rule setup, or the miss traffic processing, is considered the slow-path. Different vendors such as Nvidia Connect-X or Intel ICE supports different offload capabilities, but they follow similar software slow-path design. Depending on the hardware, the slow-path can either runs on host CPUs, or SmartNIC ARM cores, with either OVS or Linux bridge handling the traffic.

The slow-path ports that handles miss traffic is called representor netdevs, or repr, registered as a regular Linux Ethernet device and is responsible for administratively configurations of the fast-path vports. In typical use cases, representor netdev and its representee, vport, are 1-to-1 mapping: a VF/SF has a its own netdev for fast-path traffic in host, and another representor netdev for slow-path traffic in DPU. Such a design is reasonable with tens or hundreds of vports, but when scaling to thousands of VFs/SFs, the memory resource becomes a bottleneck. For example, assuming each representor's RXQ has 1024 entries, and each entry pointing to a 4K-sized page. Assume the representor netdev has 4 RXQs, such a netdev will consume 4 * 4K * 1024 = 16MB. With 1k representor netdevs, the memory consumption can grow to 16GB (16MB * 1024), just for RXQs and its buffers. In fact, although a netdev consumes memory other than its RXQs, we found that majority of memory comes from pre-allocation of RXQ buffers [4], which is used to handle the burst of incoming traffic.

The pre-allocation of RXQs and buffers are configured by Linux ethtool -G/-L option, with different vendors setting different default value. Nvidia MLX5, by default, pre-allocates 1024 entries/depth in each of its RXQs, with each entry pointing to a MTU-sized buffer or a page (without striding RQ). For Intel ICE, its default RX queue depth is 2048. We first explore a trivial memory saving solution: Reducing the number of RXQ depth of representor netdevs, from 1024/2048 to 64 (the minimum of NAPI budget). Although this saves memory but a burst of traffic over 64 packets will be dropped, causing longer connection offload setup time and performance degradation.

Instead of having one representor netdev per SF/VF, resulting in huge memory consumption, one solution deployed today is to redirect all the slow-path traffic from all VFs/SFs to a single netdev, usually PF or uplink representor netdev. In this design, the PF's RXQs receive all slow-path traffic from all other representors and all representors share the same set of RXQs memory pool provided by PF. This shared RXQ of PF design saves memory because the non-PF representors no longer allocate their own RXQs. The design is used currently used by Intel ICE, Netronome, Broadcom bnxt, and SFC [9] drivers.

With shared RXQ, we are able to create 1K SFs and 1K representor netdevs within our memory budget. However, it's hard to measure the performance impact and apply QoS for each individual representors since all traffic are mixed together in the same RXQs. For example, to prioritize a critical

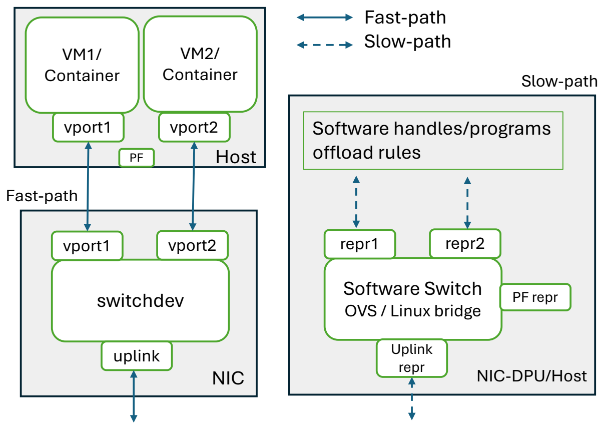

Figure 1: The fast and slow path model. The switchdev in NIC is the fast path handling hardware offloaded traffic among VFs/SFs, and the DPU which contains representor (repr) netdevs and software switch handles the slow path traffic. The slow-path can be either in Host, or be part of the NIC.

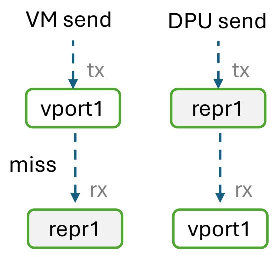

Figure 2: The vport (representee) and representor: packets transmitted by the representor netdev are delivered to its vport (right); packets transmitted by the vport are received by the representor netdev, if it fails to match any offload rule (left).

VM's traffic which comes into the shared RXQ becomes impossible. Moreover, we found that, a VM using a VF/SF can easily monopolize the entire shared RXQs by sending high volume traffic, making all other VMs with VF/SF getting zero bandwidth. We called this a fairness problem caused by shared RXQs.

We propose a solution called adjustable RXQ with shared page pool. We first save memory by dynamically reduce the RXQ depth of the device drivers, and then manage all the RXQ buffers in a shared page pool. To provide fairness, we borrow the idea from buffer management problem in shared-memory packet switches, where the hardware switch has a shared memory buffer pool for all its output ports and it guarantees both the performance and fairness [6, 2], using a mechanism called dynamic threshold.

We describe background, discuss different designs, solutions to the fairness problem, and finally some performance number. The paper has the following contributions:

- Describe the existing solutions from different driver vendors, with or without using shared RXQ deisgn.
- Propose adjustable RXQ design to save memory with minimal performance impact.
- Propose a software-based shared page pool for all netdevs, with inherent fairness mechanism.

The paper is based on recent Linux kernel, 6.9.0, which might evolve in the future. We welcome anyone to review and correct any content in any format (PR, or email to us)[1].

Background

Eswitch and Switchdev Mode

An eSwitch, or embedded switch, is a hardware component within modern network controllers, also known as a Virtual Ethernet Bridge (VEB). The eSwitch is managed by the Physical Function (PF) driver of the Ethernet or in the case of SmartNIC, managed by the ECPF (Embedded CPU's PF) that runs in the smartNIC's core. Eswitch can operate in two modes: Legacy mode and Switchdev mode.

Legacy mode operates based on traditional MAC/VLAN steering rules. Switching decisions are made based on MAC addresses, VLANs, etc. There is limited ability to offload switching rules to hardware. On the other hand, switchdev mode allows for more advanced offloading capabilities of the E-Switch to hardware. In switchdev mode, more switching rules and logic can be offloaded to the hardware switch ASIC, for example push and pop tunneling protocol, connection tracking, etc. The switchdev mode requires a slow path, software switch such as OVS or Linux bridge, and representor netdevs that handle miss traffic. The slow-path handles flows that can not be offloaded or miss the hardware flow table lookup from the VFs/SFs.

Representors

A vport can be PFs (used by administrator), or VFs or SFs (assigned to VMs or containers). When the hardware offload table is empty or disabled, all packets are processed by the slow path. Representor netdev and vport work as a pair and are like a pipe, as shown in Figure 2. Packets transmitted by the representor netdev from slow-path DPU are delivered to its vport in fast-path; packets transmitted by the vport from VM are received by the representor netdev, if it fails to match any offload rule.

For example in Figure 1, VM1 would like to setup a new TCP connection to VM2. The first TCP SYN packet sent from VM1 misses the switchdev's flow table, due to no match, and arrives at the receive queue of vport1's corresponding representor netdev, repr1. OVS, in this case, with hardware-offload enabled, will parse the protocol and insert an offload rule, using Linux tc-flower or DPDK rte_flow interface. In addition, OVS also forwards/sends the packet to the repr2 port, which will deliver the packet to vport2's receive queue at VM2. While VM2 receives the TCP SYN from VM1, VM2 replies with SYN+ACK, and again goes through the slow path. Once the connection is established and flow rules are in NIC's flow table, the TCP data stream will be going through the fast-path in NIC, and OVS in DPU no longer

[1]source: https://github.com/williamtu/netdevconf18-sharedmem

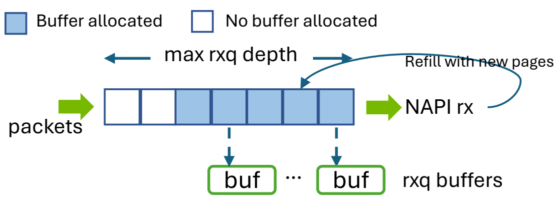

Figure 3: Network driver allocates buffers to fill its RX queue. NIC DMA packet payload into the buffers and NAPI rx function process the packet, refill the queue by re-allocating new buffers.

involves in subsequent packet processing.

Note that the slow-path is a software fall-back for the fast-path. The behaviors applied on packets of the slow-path (representor ports and software switch) should be the same as fast-path (vports and switchdev). And because representor and vport are acting like pipe, a network policy rule such as OpenFlow on a representor netdev applies to a packet on its receive path is the same as it applies on vport's transmit path.

Buffer Pre-allocation for RX Queue

Network device drivers pre-allocate a circular buffers for absorbing the burstness of network traffic. Once packets arrive, NIC DMAs a batch of packets into these pre-allocated buffers, potentially hundreds of them, before the host CPU is notified/kicked in by interrupts to process these packets. For performance reason, NIC contains multiple RX queues with each queue represents an NAPI context and an IRQ to kick in to process rx packet. Usually the number of RX queues is the same as number of CPU that involves in packet processing. As shown in Figure 3, each RXQ is a circular buffers contains several entries, with each entries point to a packet buffer. So the amount of pre-allocated memory for RXQ is a product of buffer size * number of queues * queue depth. A typical example of a data center server with 64 cores would consume memory of: 4k * 64 queues * 1k entries = 256MB, for a single netdev.

Larger RXQ depth, or more entries in RXQ, has many benefits. It helps not only for tolerating the burstness of traffic, but also the CPU processing time/jitter of a packet. When CPU is under heavy loading, a packet arriving RX interrupt might not be able to immediately kick in the CPU to process packets. In addition, each packet usually has different processing time. The time for CPU to process a TLS packet is definitely longer than an ARP packet. Thus, a deeper RXQ also helps avoiding packets dropped due to intermittently longer packet processing time.

However, memory is not cheap, especially when scaling to 1K devices and with 1k representor netdevs in DPU. And reserving several GBs of pre-allocated memory idle just waiting for a burst of traffic or even unused when the low traffic volume is a huge resource waste. We discuss different solutions in later sections.

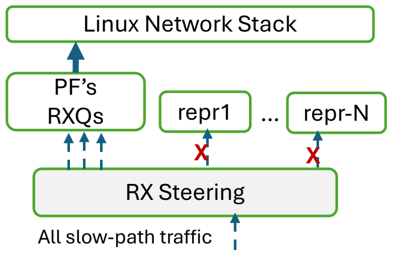

Figure 4: Shared RXQ Design: A PF or uplink-representor netdev receives all slow-path traffic for other representors, and reconstructs their skb and passes to upper network stacks.

Design

With the goal of supporting 1K of SFs/VFs with representor netdevs on DPU, we set the following requirements: 1) 1k netdevs creation time around 10 minutes, including configurations and bringing the device up, 2) memory consumption of DPU or host system is within system limit, and 3) all the VFs/SFs representor get fair share of receive memory buffer, which leads to fair share of bandwidth. The following subsections presents four designs: A. Dedicated RXQ, B. Shared RXQ of PF, C. Adjustable RXQ with shared memory pool, and D. Shared RXQ with hardware meters.

A. Dedicated RXQ

This design is the default design with minimal or no change. Dedicated RXQ is used in previous version of Intel ICE before patch [10] and also the current version of Marvell Octeontx2 [8]. In this design, each representor has its own RXQ, and do not share its RXQ with any other representors. The design guarantees performance isolation between representors, but the downsides are that it 1) consumes lots of RXQ memory in the system, and 2) some NIC hardware has limitation of how many hardware RX queues it can create. For example, Intel ICE can create up to 1k queues. If users want to use more than 1k representors, there is not enough queues to assign to each representor netdev.

B. Shared RXQs

Since the representor netdev only handles the miss traffic or first few packets for connection setup, one solution is to aggregate all slow-path traffic from all representor netdevs into a single one, usually the PF representor or the uplink representor netdev. The PF representor acts like an intermediate multiplex layer, and it sees all slow path traffic from all other reprensentors. When packets arrive, it looks up an internal data structure to identify the original vport id of VF/SF, reconstructs the skb-¿dev to the correct orignal netdev, and passes up kernel stack. As a result, the non-PF or non-uplink representor netdev no longer sees packets and no need to allocate RX queue, as shown in Figure 4.

This design saves huge amount of memory because most of the representors' RX functions are avoided, with all representors' traffic using PF representor netdev's RXQs. In the case of ICE driver, even the TX function of the representors are handled by PF representor, making other representors purely a control/management interfaces. Different vendors have different implementation, but in general, the device driver needs to do the following:

- RX Hardware Steering: instead of sending slow-path traffic to each individual representors, the NIC needs to steers/redirects all slow-path traffic to the PF representor netdev, and mark the packet with its original vport information, e.g., vport_id, in metadata or packet buffer.
- RX in Driver: the PF representor netdev sees all traffic including others arriving at its RX queues. It needs to keeps a map of vport_id to the packet's original netdev struct. By extracting the vport_id from metadata, the original netdev is restored to sk_buff and continue kernel network stack.
- TX in Driver: because the vport and representor act like a pipe, the PF representor netdev need to know which vport's RX queue the packet should be *send* to. Drivers can use the dst_entry of skb, or other per-packet metadata field, and rely on internal switch to deliver the packet to the vport.

Pros and Cons: Although this design saves lots of memory by redirecting all traffic to PF representor netdev, users need to tune the following two parameters: because it consumes only one service netdev's memory. Depending on number of SF/VFs and traffic volume, users need to configure two parameters of service netdev:

- the number of RXQs should increase to decrease the chance of multiple flows from multiple vports rxhash to the same queue, e.g., using ethtool -L combined. Most of the drivers limit its max number of RXQs to the number of CPUs.
- the RX queue depth should increase to absorb more traffic burst from multiple vports, e.g., using ethtool -G rx. However, this might increase the packet processing latency.
- Reduce the NAPI schedule delay, which means NAPI can process packets as soon as possible when packets arrived. This lowers the chance of RXQ utilization grows to fast or overflow. (I don't know any tool to do this).

The design is simple to implement, but we found that it suffers from a *fairness* issue. A single high volume sender, ex: dpdk-pktgen, sending to its vport and with packets arriving at slow path, can easily consume all the buffers in the PF representor netdev's RXQ. And because all other representors share the same PF's RXQ, this makes other representor and its vport traffic being dropped due to no available RX buffer at all. An example setup script to show case the issue using OVS is provided in Listing 4.

C-1. Adjustable RXQ Depth

If the shared RXQ of PF suffers from fairness issue, another solution is to *not* sharing the RXQ, but think about other ways to save memory. For every RXQ in every netdev, existing network drivers implement *static* RXQ allocation. Whenever a driver initializes, it always pre-allocates the number of buffers to the full RXQ depth, and whenever a batch of packets arrived and processed by NAPI poll, device driver again refills to full by re-allocating buffers to the *full* RXQ depth.

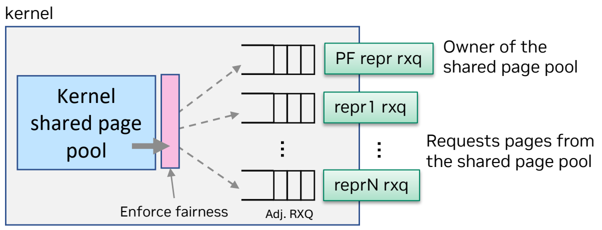

Figure 5: The adjustable RXQ from multiple reprs requests pages from the shared page pool. A fairness layer guarantees each repr gets the proportional shares of the total available memory.

To save memory, instead, the adjustable RXQ depth does not refill to full RXQ depth, but only to a threshold value, called low_watermark. For example, the mlx5 driver, by default, always tries to allocate to full 1024 rxq buffers. With adjustable RXQ, the following logic is applied when a driver is receiving packets in its NAPI context:

- NAPI busy: allocate buffers and refill to full queue depth, high_watermark. The high_watermark equals the max RXQ depth set by ethtool -G rx.
- NAPI interrupt: allocate buffers and refill to low watermark if current RXQ depth is lower than low watermark
- NAPI interrupt: do not allocate and do not refill if current RXQ depth is higher than low_watermark.

Pros and Cons: We set the low_watermark to be 128 (2 * NAPI_BUDGET). The low_watermark means the minimal available RXQ buffers to handle the worst case of traffic burst. At driver initialization time, all queues are allocated only up-to the low_watermark. Once the first burst of traffic arrives, users might see more packets dropped compared to full rxq depth allocation (1024). Fortunately, a NAPI busy state will trigger driver to refill RXQ to its full depth, making the performance impact minimal. In this design, we pay the performance price to save memory. The first burst traffic of a connection definitely see more packets dropped, and once in NAPI-busy, the driver acts the same as static RX queue allocation.

Such a design saves memory at driver initialization, as well as when at the NAPI-interrupt state, device driver will slowly return RXQ buffers back to kernel due to packet arrives but driver does not re-allocate. However, what if an RXQ is in NAPI-interrupt state, but there is no packet arrived to trigger returning buffers to system? Then driver needs to detect and drain the RX queue back to the low_watermark.

We implement this prototype and evaluate its impact in performance evaluation section.

C-2. Adjustable RXQ Depth with Shared Page Pool

We found that the memory savings with the design of adjustable RXQ depth might not be enough. Currently each RXQ creates its own kernel page pool, and in NAPI, refilling the RXQ buffers by allocating a page pool entry from its own per-queue page pool. When scaling to thousands of representor netdevs, there is no guarantee that all the representor netdev gets fair shares of system memory. That is, the later created representor netdevs might see system out-of-memory for initializing its RXQ, even with the Adjustable RXQ Depth design.

We propose shared page pool, a layer of page pool that runs on top of current Linux page pool APIs. The shared page pool supports for a group of netdevs to register as user, request buffers to use in its RXQs, and the shared memory pool fairly allocate memory for each user based on the its current usage and total available memory in the pool.

Figure 6 shows the idea. Assume N representor netdevs, with each has one RXQ. In the beginning, a special netdev, PF or uplink representor, creates the shared page pool, and joins itself into the pool. The rest of netdevs created later join the shared page pool, by registering itself to the pool, and requests RXQ buffers by calling the alloc and free API to get or put the page into shared pool. We define the following APIs:

```
# Creates page pool, return the handler
spp = shared_pp_create(
    max_num_devs, page_pool_size, ...);

# Return a user id for registerd user
spp_uid1 = shared_pp_join(spp, rep1,
    qid, threshold);
spp_uid2 = shared_pp_join(spp, rep2,
    qid, threshold);

# Representor NAPI processes packets and
# refills, and bounded by max_usage, return
# -ENOMEM if overflow
shared_pp_dev_alloc_page(spp, spp_uid1);
shared_pp_dev_alloc_page(spp, spp_uid2);
shared_pp_put_page(spp, spp_uid1);

#Leave the share page pool
shared_pp_leave(spp, spp_uid1);
```

But how to solve the fairness issue? The problem of multiple RXQs sharing a single page pool is very similar to the shared memory for multiple ports problem in hardware switch [2], configured using devlink-sb [6]. Instead of hardware switch memory, here we have shared page pool memory, and in contract to switch output ports competing for shared buffers, here it's the RXQs from different representor netdevs. Given the similarity, we re-use the dynamic threshold formula to define the max number of RXQ buffers assigned for each RXQ. That is, the max_usage of a particular shared page pool user is defined by a to_alpha value below:

$$\alpha = 2^{(\text{to\_alpha}-10)} \tag{1}$$

and the to_alpha is ranged between 0 to 20, and with α, we can get max_usage below:

$$\text{max\_usage} = \frac{\alpha}{1+\alpha} \times \text{Free\_Buffer} \tag{2}$$

Note that the Free_Buffer means the currently available free pages in the shared page pool. As a result, each user of the pool have a dynamic max_usage based on currently page pool memory utilization.

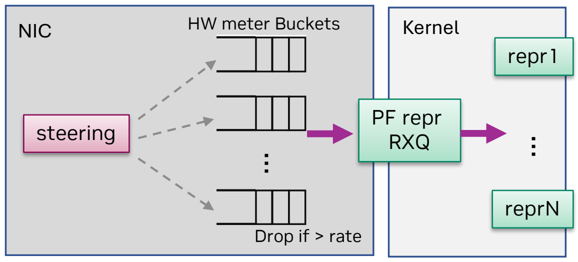

Figure 6: The design uses hardware meter in steering to rate-limit each repr's traffic, so the RXQ buffers won't be monopolized by a particular vport/repr.

For example, assuming to_alpha value is 11, so $\alpha = 2$, max_usage $= 1/2 *$Free_buffer. This mean at any given point of time, a representor netdev can use up to half of the shared memory pool, even if there is no other users. An as more and more users join the shared page pool, the Free_buffer decreases, but with each driver frees up memory after processing packets, the Free_buffer will increase again.

Pros and Cons: The design is simple but leaves the user to configure the value of α. When the traffic distribution to different representors is uneven, a large α will limit the high volume representor to use all the available memory, while a smaller α value with a even traffic distribution might still cause unfairness or some representors get no page. The paper [2] is based on simulation, and we don't have a good tool or benchmark traffic pattern to measure the effectiveness of the design.

Note that existing Linux page pool requires registering a DMA device to do dma map and unmap, and because representors all share the same DMA device, so they can share single page pool. Other use cases are non-physical device such as Linux veth/tun/tap.

D. Shared RXQ with Hardware meters

Another approach to solve the fairness and memory issue is to seek for hardware support. This design is based on A. Shared RXQ of PF, and enforces the fairness *before* packets arriving the RXQ buffer. Most of the NICs supports hardware metering, which allows users to configure it as a rate limiter, either in PPS (packets-per-second) or BPS (bytes-per-seconds). Before traffic arrives and consumes the RXQ's buffer, the meter rate limiter might pass the packet to continue using the shared RXQ of PF, or, the meter rate limiter might drop the packet if a particular flow of a vport is over its rate.

Pros and Cons: We implemented this design and realize a couple issues. First, not all NIC hardware support meters. The implementation is vendor-specific and comes with different limitations. Second, setting the right meter rate is hard. Users usually have no prior knowledge about how fast the DPU/CPU can process the slow path traffic, either in terms of PPS or BPS. Setting tool low under utilizes the RXQs and setting rate too high causes fairness issue. Finally, maintaining thousands of steering rules with meters is not easy, as the steering rules might collide with other rules with different priorities.

Driver	Implementation
Nvidia MLX5	Dedicated RXQ
Intel ICE	Shared RXQs [10]
Intel ICE (before [10])	Dedicated RXQ
Broadcom BNXT	Shared RXQs [9]
Netronome NFP	Shared RXQs [9]
Solarflare SFC	Shared RXQs [9]
Marvell Octeontx2	Dedicated RXQ [8]

Table 1: Summary of existing network drivers and its representor solution.

Implementation

Most of the network device drivers with representor implement the shared RXQ design, which saves memory at the price of possible fairness limitation. Table 1 show a summary different network device driver's implementation choices. For MLX5 driver, we'd like to propose a configuration knobs that turns on or off the Shared RXQ, depending on users' choice.

New Devlink Eswitch Attribte: spool-mode

Given the fairness issue, we'd like to propose a new devlink eswitch attribute for users to enable or disable the Shared RXQs. When memory is a precious resource such as SmartNIC or DPU, enabling the Shared RXQs saves the most memory as it only uses the shared RXQs. And the memory consumption does not grow linearly as the number of representors grows. On the other hand, if users worry about potential fairness issue or need performance isolation, then disabling shared RXQ and fallback to the dedicated RXQ can solve the problem. Below is the proposed devlink, spool-mode:

- none: Driver uses dedicated RXQ on its representor netdev.
- basic: Driver uses shared RXQs on its representor netdevs.
- spp (shared page pool): Driver uses shared page pool with adjustable RXQ depth on its representor netdevs.

The attribute supports only in switchdev mode.

```
$ devlink dev eswitch set pci/0000:08:00.0 \
   mode switchdev spool-mode none
$ devlink dev eswitch show pci/0000:08:00.0
   pci/0000:08:00.0: mode legacy \
   inline-mode none encap-mode basic \
   spool-mode none
$ devlink dev eswitch set pci/0000:08:00.0 \
   mode switchdev spool-mode basic
```

With this, existing drivers will need to consider supporting both modes, shared RXQ mode or dedicated mode. (Or return -ENOSUPP).

Performance Evaluation

We conduct our experiment with two servers connected back-to-back using two x86 host, shown in Figure 7. Each host is equipped with one dual port BlueField-2 card. Each x86 host consists of 10 cores and 32GB of RAM. The BlueField-2 is an 8-cores Cortex-A72, with 16GB of RAM. BlueField, by default, in production environment, runs OVS kernel datapath

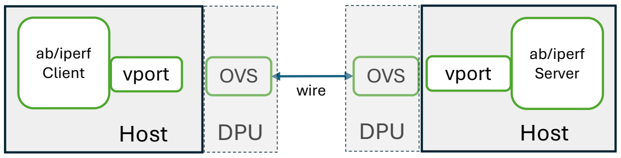

Figure 7: Back-to-back setup with x86 host and ARM DPU for performance evaluation.

with hardware-offload enabled. Unless mentioned, we run all experiements with OVS hardware offload *disabled*. This forces the traffic to go to slow-path representor netdevs so we can easily measure the performance impact.

Memory Consumption

Linux kernel today provides several memory profiling tool, such as /proc/meminfo, /proc/slabinfo. However, there is no tool to measure how much memory is consumed by a particular network device driver, or a netdev. We start by creating 200 SF representors, and measure the difference of the total system free mem, MemFree field in /proc/meminfo. The SF creation and setup is done by the script in Listing 1.

Table 2 shows the memory consumption of each SF-repr and its breakdown. We conclude that, by default RXQ depth of 1024, one SF-repr takes 2.86MB of kernel memory, and among the 2.86MB, 1.83 is consumed by page pool for the RXQ buffers, around 0.11MB is consumed by firmware, and the rest of 0.9MB is for others such as slab, ex: kmalloc, kzalloc, and other kernel netdev structures. When increase from 1 RXQ to 2 RXQs, the memory consumption also increases linearly, as expected.

	1RXQ total	Page Pool	FW	2RXQ
128	1.1	0.2	0.113	2.835
256	1.3	0.4	0.114	2.915
512	1.805	0.78	0.114	2.99
1024	2.86	1.83	0.114	5.025
2048	4.935	3.93	0.115	9.25

Table 2: Memory consumption of a SF-repr and its breakdown of page pool and Firmware usage.

Static Queue Size

Before enabling the dynamic adjustable RXQ size, we first analyze the performance impact of statically configure different RXQ size, from queue depth of 64 to 2048. We ran Apache ab benchmark [**?**], with client on one of the x86 host and server on another x86 host. We disable OVS hardware-offload in DPU so that all traffic can flow through the representor netdev. We than statically adjust the representor netdev's RXQ depth using ethtool -G.

Table 3 shows the result of Apache ab benchmark sending 1 million requests with concurrency level of 100. We report the following metrics:

- Time to complete (sec): total time taken for completing the 1 million requests.

- out of buffers (K): a firmware counter, rx_out_of_buffer, reporting number of packets dropped due to no RXQ buffer available.
- Requests (K) / sec: average HTTP requests per seconds
- Connection Time (ms) and SD: average connection time, including connect, processing, and waiting, of the 1 million connections and their standard deviation (SD).

We found although larger queue depth, 1024 or 2048, consumes more memory, but they performance comparatively well, as they shows shorter time to complete, less packets being dropped, and lower jittering (SD is smaller).

	Time to complete (sec)	**out of buffers (K)**	**Requests / sec (K)**	**Conn Time (ms)**	**Conn Time SD**
64	45.6	104	21.9	5	32
128	29.48	71	33.9	3	20
256	24.55	5.89	40.7	2	4
512	24.06	1.2	41.5	2	2.2
1024	24.09	0	41.5	2	1.9
2048	24.03	0	41.2	2	1

Table 3: Results of Apache ab benchmark with 1 million requests and 100 concurrent connections, with OVS hardware offload disabled in DPU.

Dynamic Queue Size

We implement the idea of adjustable queue side, by setting the low_watermark to be queue depth of 128, and high water_mark is configured between 256 to 2048, using ethtool -G. The mlx5 driver initializes its RXQ only to the low_watermark. When the mlx5 napi poll function detects in busy state, it refills the RXQ to its high watermark. Otherwise if napi poll is in interrupt state, the driver doesn't refill at all if the current available buffers in the RXQ is higher than the low_watermark. In interrupt mode, we only refill when available buffers in the RXQ is lower than the low_watermark.

Table 4 shows the results.

	Time to complete (sec)	**out of buffers (K)**	**Requests / sec (K)**	**Conn Time (ms)**	**Conn Time SD**
256	24.6	22	40.1	2	9.6
512	24.1	2.1	41.3	2	2.9
1024	24.2	0.95	41.2	2	2
2048	23.9	0.65	41.7	2	1.8

Table 4: Result of Apache ab benchmark with 1 million requests and 100 concurrent connections, using dynamic adjustable RXQ.

Compared Table 4 with Table 3, the dynamic queue size shows noticeable performance difference. We first observe that even with queue depth high_watermark set to 2048, we still see several packets drop. This is due to the driver initializes only at low_watermark, so the first burst of traffic definitely triggers out of buffer drops. In addition, although the total time to complete and the average connection time are almost the same, the jittering (SD) is much higher due to more packets being dropped and retransmission.

Finally, we compare 20 seconds tcp single connection performance using iperf3 in Table 5. The dynamic queue size design performs almost the same, with the worst case of dropping around 8% when queue size is 2048. Note that all the experiments are done by disabling the hardware offload. When enabling hardware offload, most of the traffic are processed in hardware and we're not able to see any performance difference. In conclusion, we feel that the performance impact will be even smaller in production environment when hardware offload is always enabled.

	BW Gbps (Static)	**BW Gbps (Dynamic)**
64	3.78	2.71
128	5.23	5.19
256	6.03	6.15
512	6.25	6.40
1024	6.08	5.79
2048	6.03	5.49

Table 5: TCP bandwidth comparison of static queue size v.s dynamic queue size, with HW offload disabled.

Discussions and Future Work

There are many ideas we are planning to try. Spinning CPU to save RXQ buffers DIM (Dynamic Interrupt Mediation) Shared page pool for virtual devices

Related Work

Setting the right buffer size is important for network performance. The paper [1] provides a guideline for buffer sizing in network routers based on the bandwidth delay product. And the paper [2] proposes buffer management scheme called Dynamic Threshold (DT) for shared-memory packet switches. Although both papers are designed for hardware switches/routers, we found similarities in the case of host networking with switchdev. The buffer sizing, or sizing the RXQ depth, is more complicated in host network as the per-packet processing time varies a lot due to CPU workload, interrupt delivery/moderation, and vendor driver implementation. However, the solution space in host networking is more flexible as ideas can be implemented in software in stead of hardware.

The problem PicNIC [5] tried to solve is similar, however this work focuses on performance isolation breakage caused by the slow-path design. Applying back-pressure or shaping at switchdev vport is difficult as we only want to apply on slow-path traffic, but the shaping rule will be applied on both slow-path and fast-path traffic. The early drops solution is possible using hardware meter, as discussed in the design session.

ShRing [7], unlike this work, shares Rx buffers between cores and not only within cores. Junction [3], similarly to this work, shares Rx buffers to reduce memory use. But, in contrast to this work, Junction uses a single coordinator core to repost Rx buffers.

Conclusion

Scaling to thousands of vports with their representor netdevs is challenging, especially in the SmartNIC/DPU deployment. The paper shares our experience in Nvida BlueField deployment, and examines the existing solutions from various device drivers, using shared RXQ of PF, or dedicated RXQ. We discuss each design's pros and cons, and propose a new design called adjustable RXQ with shared page pool. We conclude that a huge amount of memory can be saved, by reducing default RXQ depth from 1024 to 512/256, and the performance impact remains very little.

Acknowledgments

We'd live to thank many people for their valuable feedback, from internally and the Linux kernel community, including Saeed Mahameed, Jiri Pirko, Parav Pandit, Tariq Toukan, Jakub Kicinski, Bodong Wang, Wei Bai, and Dan Jurgens.

References

[1] Appenzeller, G.; Keslassy, I.; and McKeown, N. 2004. Sizing router buffers. *ACM SIGCOMM Computer Communication Review* 34(4):281–292.

[2] Choudhury, A., and Hahne, E. 1998. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking* 6(2):130–140.

[3] Fried, J.; Chaudhry, G. I.; Saurez, E.; Choukse, E.; Goiri, Í.; Elnikety, S.; Fonseca, R.; and Belay, A. 2024. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 55–73.

[4] Nic memory reserve. https://people.kernel.org/kuba/nic-memory-reserve.

[5] Kumar, P.; Dukkipati, N.; Lewis, N.; Cui, Y.; Wang, Y.; Li, C.; Valancius, V.; Adriaens, J.; Gribble, S.; Foster, N.; et al. 2019. Picnic: predictable virtualized nic. In *Proceedings of the ACM Special Interest Group on Data Communication*. 351–366.

[6] Pirko, J. devlink-sb(8) — linux manual page. https://man7.org/linux/man-pages/man8/devlink-sb.8.html.

[7] Pismenny, B.; Morrison, A.; and Tsafrir, D. 2023. {ShRing}: Networking with shared receive rings. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 949–968.

[8] sowjanya, G. Introduce rvu representors, marvell. https://lore.kernel.org/netdev/20240701201215.5b68e164@kernel.org/T/m93a806be679e647f0dcbfee7edeb2fc53d2cf5cc.

[9] Survey: Documentation: devlink: Add devlink-sd. Linux netdev mailing list. https://lore.kernel.org/netdev/39dbf7f6-76e0-4319-97d8-24b54e788435@nvidia.com/.

[10] Swiatkowski, M. ice: use less resources in switchdev. https://patchwork.kernel.org/project/netdevbpf/cover/20240125125314.1-michal.swiatkowski@linux.intel.com/.

Scripts

```
#!/bin/bash
#ethtool --set-priv-flags p0 rx_striding_rq off
for i in {100..200}; do
    devlink port add pci/0000:08:00.0 flavour pcisf pfnum
     0 sfnum $i
done
for i in {100..200}; do
    ethtool -L $dev combined 1
    ethtool -G $dev rx 1024
    ip link set dev $dev up
done
#optional: create SF
# devlink port function set $dev state active
# devlink dev param set auxiliary/mlx5_core.sf.$i name
     enable_eth value 1 cmode driverinit
# devlink dev reload auxiliary/mlx5_core.sf.$i
```

Listing 1: SF-representor setup on BlueField

```
$ ./tools/net/ynl/cli.py --spec Documentation/netlink/
     specs/netdev.yaml  --dump page-pool-get
[{'id': 673,
  'ifindex': 120,
  'inflight': 512, // 512 pages used for 1024 RXQ depth
  'inflight-mem': 2097152, // about 2MB
  'napi-id': 1179}, // each RXQ has its own NAPI-ID
 {'id': 674,
  'ifindex': 122,
  'inflight': 512,
  'inflight-mem': 2097152,
  'napi-id': 1180},
  ...
```

Listing 2: Collect page_pool usage on BlueField

```
# 1 million requests with 100 concurrency
$ ab -c100 -l -n 1000000 -k -l http://10.1.1.1/
# single TCP connection bandwidth
$ iperf3 -t20 -i2 -c 10.1.1.1
```

Listing 3: Apache ab and iperf3 test

```
#!/bin/bash
# similar setup for Linux bridge, but disable HW offload
     by setting aging=0
PF1=eth2
VFREP1=eth4
VFREP2=eth5
VF1=eth6
VF2=eth7
NS1=ns1
NS2=ns2
setup_dev_ns()
{
        ns=$1
        vfdev=$2
        ip=$3
        ip netns del $ns || true
        ip netns add $ns
        ip link set dev $vfdev netns $ns
        ip netns exec $ns ifconfig $vfdev ${ip}/24 up
}
test_dpdk()
{
    ip netns exec $NS1 bash
    echo 1280 > /sys/devices/system/node/node0/hugepages/
     hugepages-2048kB/nr_hugepages
    dpdk-testpmd -l 0-3 --socket-mem=512  -a 0000:08:00.2
     -- -i --nb-cores=1 --forward-mode=txonly \
--eth-peer=0,b8:3f:d2:ba:65:9e --txpkts=64 --txq=1 --rxq=1
      --stats-period=1 --txonly-multi-flow \
--total-num-mbufs=2048
        exit

}
setup_ovs()
{
        echo 2 > /sys/class/net/$PF1/device/sriov_numvfs
        ovs-vsctl set Open_vSwitch . other_config:hw-
     offload=false
        /usr/share/openvswitch/scripts/ovs-ctl restart
        ovs-vsctl add-br ovsbr0
        ovs-vsctl add-port ovsbr0 $PF1
        ovs-vsctl add-port ovsbr0 $VFREP1
        ovs-vsctl add-port ovsbr0 $VFREP2
        ip link set dev $PF1 up
        ip link set dev $VFREP1 up
        ip link set dev $VFREP2 up
        setup_dev_ns $NS1 $VF1 192.167.111.1
        setup_dev_ns $NS2 $VF2 192.167.111.2
        ip netns exec $NS1 ping -i .05 -c10 192.167.111.2
        ip netns exec $NS2 ping -i .05 -c10 192.167.111.1
}
devlink dev eswitch set pci/0000:08:00.0 mode switchdev
setup_ovs
test_dpdk
```

Listing 4: DPDK-pktgen consumes all RXQ