enfabrica

Lightweight Approach to Kickstart Development of NIC Driver

David Ahern

October 2023

Context and Motivation

Great idea for a new ethernet NIC

- Need to turn slides and ideas into something real - fast

Key point - starting from scratch!

- No existing H/W-S/W API to leverage
- No existing S/W to extend or adapt

Parallel H/W - S/W development

Ability to do POCs very quickly

- zctap, Linux devmem, etc.
- Do not want to wait for H/W support, APIs to be defined, etc.

qemu Based Models

Absolutely provides value - **in time**

Requires definitions around H/W - S/W API

- Beginning of time: no BARs, no CSRs, no messaging between host S/W and firmware

PCI interface is not unique to your device

- DMA, MSI-X, BAR management - needed in time; not a priority

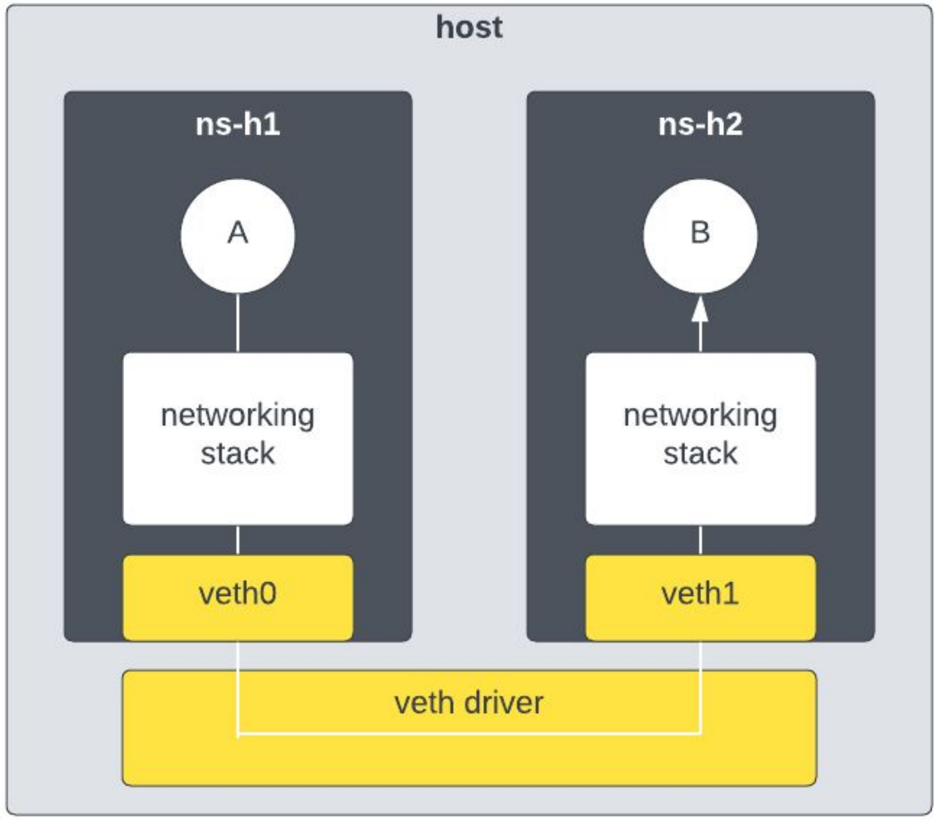
S/W driver development does not need to be serialized behind the H/W - S/W API

- Many, many ethernet and Linux networking APIs can be coded
- Start building unit, integration and end-to-end test cases
- Start working on related S/W (e.g., SDKs, IB module and provider, ...)

Focus on what is important now

- Design of queue descriptors and requests from H/W
- S/W arch and development, test environment and confidence in the driver and related code - all on a simplified device model

Prior Art: veth devices



veth is an example of **intent**

- not tied to a PCI device

Very simple device model

- Tx from one veth is Rx into its peer
  - skb is forwarded
- No packet processing done by the model
  - just a passthrough

Allows complex S/W testing within a single host

- Namespaces represent endpoints

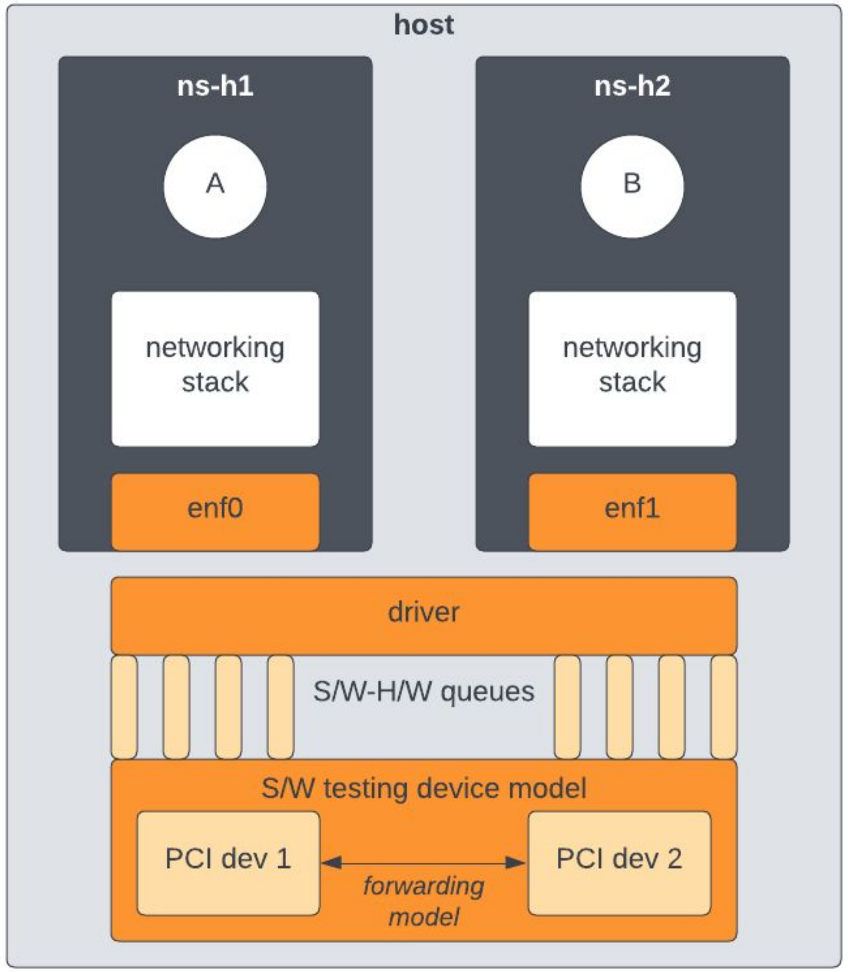
Extending veth Intent

Add a device model

- use H/W queue descriptors
- Tx: skb is posted to S/W model via descriptor
  - Tx completions when entry processed by model
- Rx: S/W model accepts packet from peer device and fills buffer and descriptor
  - Rx completions generated for packet

Forwarding model between devices

- depends on what the device is doing or mimicking or what is relevant to the testing
- “simple” - peer-to-peer - Tx from one device is Rx into peer
- “switch model” - lookup dmac
- “routing model”



Leveraging tap Devices

"Peer device" is tap based

Tx **from model**

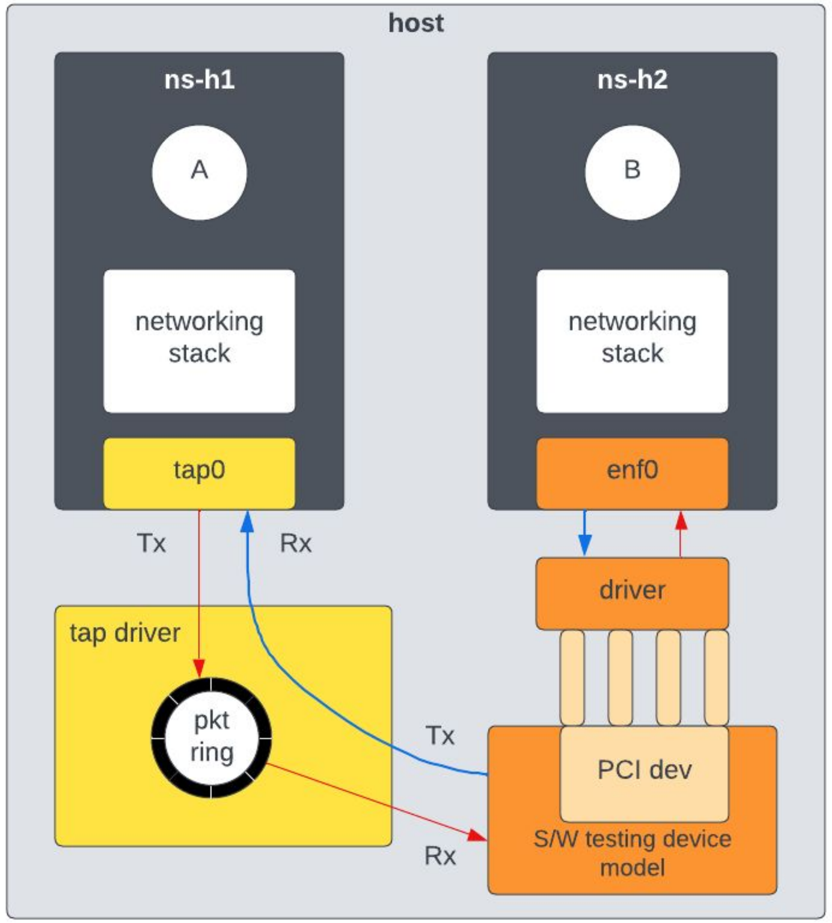
- kernel_sendmsg to handoff packet to tap device
- Rx into tap == Rx into networking stack

Tx **from tap**

- Packets placed into ring and ring owner notified
- Device model pulls a packet from the ring
  - equivalent to receiving from the wire

Allows

- Unidirectional development (start with Tx)
- Packet captures to verify manipulations done by device model (e.g., checksum offload)
- netem on tap devices to introduce random drops in network path



Essential Elements from a Driver Perspective

PCI device

- Host stack sees a PCI device
- pci_dev operations

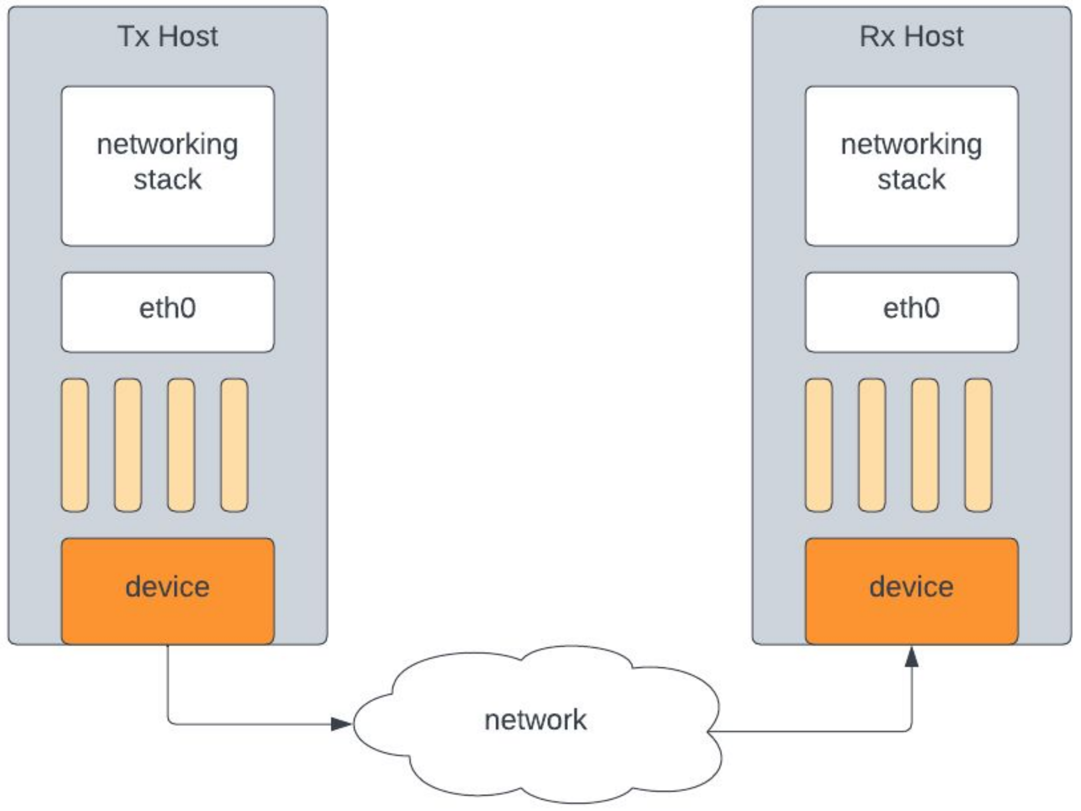
netdev representor

- Allows device to work with standard Linux networking stack APIs

Queues

- host memory
- submitting packets to H/W
- receiving packets from H/W

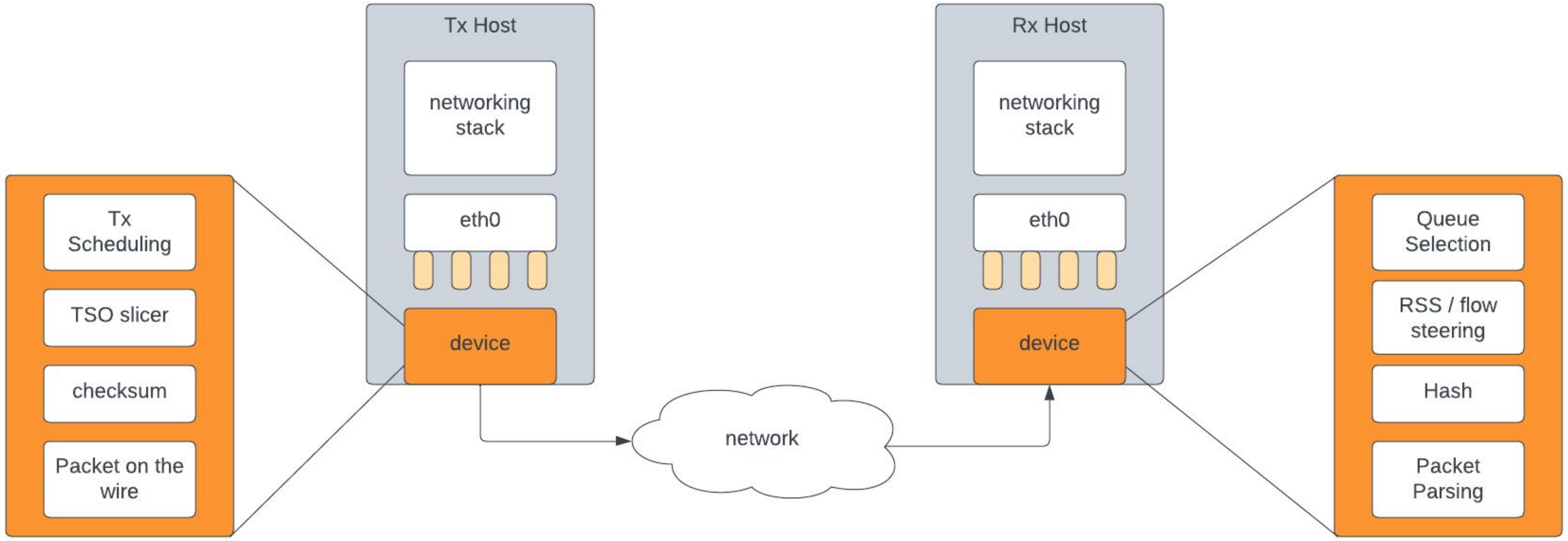
Flow steering and RSS



Hardware Modeling

Does not have to be exact - it's a model

Only worry about essential elements from S/W driver perspective



Driver Code Paths that Matter Now

PCI device operations

- stubs for S/W model, but ready for H/W

devlink operations

- Prepare code to handle devlink ops

Netdev representor

- netdev operations for managing device interface to Linux networking stack

Control path

- Managing queues and their lifecycle
  - host memory operations (e.g., alloc, free), H/W operations (create, enable, disable, delete)
- Flow steering and RSS - ethtool APIs

Code Paths that Matter Now

Datapath

- Queue descriptor formats
- Rx: getting packets from the device
  - Need to supply buffers so memory management and page_pool
  - Completion handling and napi
- Tx: sending packets to your device
  - Completion handling releasing skbs

Closed loop datapath

- Operational sockets (e.g., TCP roundtrip), not just packet injection
- Second level S/W aspects: zerocopy, retransmit, ...

Driver and H/W stats

- ethtool API

Key point: Able to get a lot of driver infrastructure in place and ready for extension to H/W details

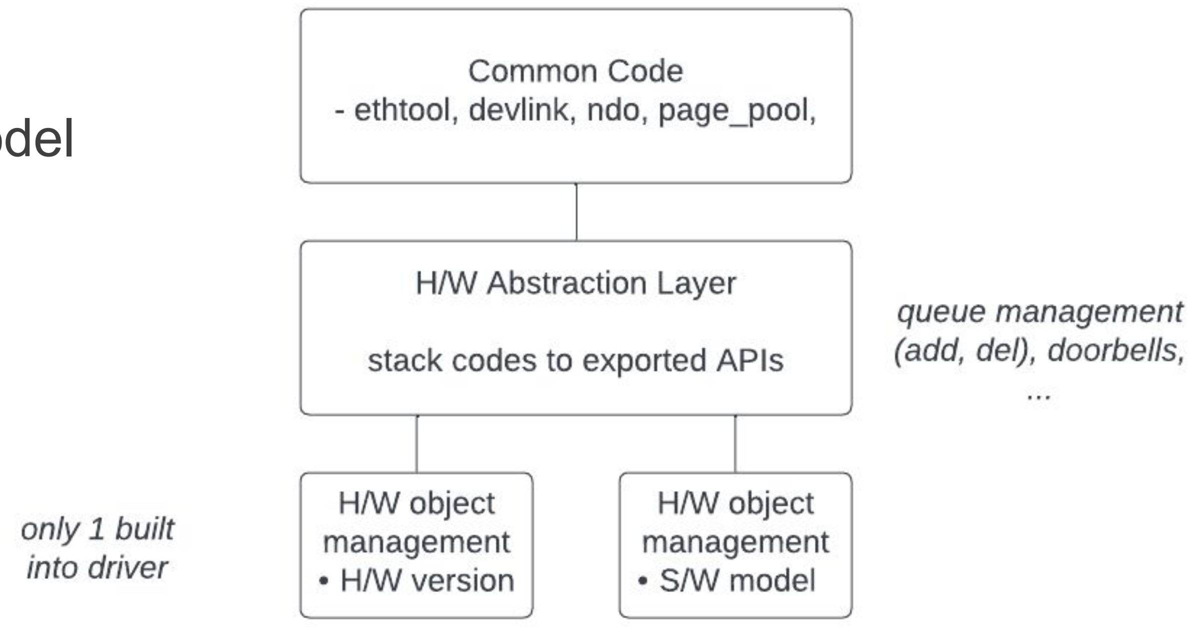
Source Code Organization

Use HAL APIs

- Ensure S/W stack sees consistent API
- Model differences are hidden behind model specific handlers of the HAL
  - e.g., mydev_hw.c vs mydev_swmodel.c

Flags dictate backend to compile in

S/W model operates in virtual addresses; H/W model needs physical address



Module Initialization

Standard starter for a module

- module_init(enf_init_module);
- module_exit(enf_exit_module);

Need device model dependent implementations

- H/W model
  - Call pci_register_driver
- S/W model
  - Create N pci_dev instances (see next slides)
  - Fake PCI probe
  - Spawn kernel thread to monitor Tx queues (and tap packet ring if relevant) and forward packets (more later)
    - pin it to 1 cpu; migrations of kthread cause unnecessary headaches

Device Initialization - PCI interface

Drivers for PCI devices register pci_driver

- probe and remove callbacks

Write callbacks with in model dependent functions

- PCI initialization (resources, iomap, enable, bar, irq, ...) in H/W file
- Stubbed out for S/W model

```
int enf_probe(struct pci_dev *, const struct pci_device_id *)
{
     struct enf_pdev_priv *epp;  // PCI dev private data
     struct devlink *dl;

     // allocate pci_dev private struct using devlink as
     // the container
     dl = devlink_alloc(&dl_ops, sizeof(*epp), &pdev->dev);

     // PCI initialization per-backend
     err = enf_pci_init(epp, pdev, id);

     // register instance with devlink
     err = devlink_register(dl);

     // other devlink calls e.g., devlink_params_register()

     // create netdev representor for PCI device
     // - standard netdev init and register
     // - same regardless of H/W vs S/W model
     netdev = enf_init_netdev(pdev);

     // other relevant device initialization
}
```

Device Initialization - S/W Model

Mimick PCI scan and device probe

- Allocate pci_dev structure
- Use a platform_device to get a `struct device`

Call the properly partitioned PCI probe function

- H/W model can run through the PCI initialization functions
- S/W only model just sets private data for the pci_dev struct

```
struct test_priv {
        struct pci_dev pdev;
        // test data unique to this device instance
        // e.g., queues and metadata (Rx, Tx, completion),
        // RSS and flow steering references, irq references
};

struct enf_test_priv *tpriv;
struct pci_dev *pdev;
char buf[16] = {};

// per-PCI device instance (e.g., veth style requires 2)
tpriv = kzalloc(sizeof(*tpriv), GFP_KERNEL);
pdev = &tpriv->pdev;

snprintf(buf, sizeof(buf) - 1, "testing-dev-%d", dev_id);
platform_dev = platform_device_alloc(buf,
                                     PLATFORM_DEVID_NONE);
err = platform_device_add(platform_dev);

pdev->dev = platform_dev->dev;

/* call the PCI probe function */
err = enf_probe(pdev, NULL);
```

H/W Notifications: irqs

Handling irqs

- S/W model implements irq_request function
- Saves handler, data and cpu affinity
- S/W model kthread ensures irq handler is invoked on right cpu
- irq handler invokes napi_schedule

```
static int enf_test_fake_msix(void *data)
{
     struct enf_irq *irq = data;

     irq->fn(irq->data);
     return 0;
}

static void enf_test_notification(struct enf_irq *irq)
{
     if (irq->cpu == raw_smp_processor_id())
          enf_test_fake_msix(irq);
     else
          smp_call_on_cpu(cpu, enf_test_fake_msix,
                          irq, true);
}


// per-cpu irq handler; schedule Rx softirq
static void enf_channel_irq_handler(void *data)
{
     struct enf_channel *ch = data;

     if (in_irq())
          napi_schedule_irqoff(&ch->napi);
     else
          napi_schedule(&ch->napi);
}
```

H/W Notifications: Doorbells

Doorbells

- HAL wrappers for index moves
  - H/W version writes to CSRs
  - S/W version calls into S/W model to move pidx (e.g., Tx, Rx buffers) or update cidx (e.g., CQ)
- kthread sees pidx move on next scan
  - Do not want to Tx packets inline with index move

Control Path

enfabrica

Queue Management

- netdev initialization needs to create one or more queue sets (e.g., per-cpu)
- enf_rxq_{add,del}(), enf_txq_{add,del}(), enf_cq_{add,del}()
    - alloc / free and start / stop versions

Backend specific handler

- H/W model - creates queues based on H/W - S/W API (e.g., message to firmware)
- S/W model - functions are into model and handled inline

Data Path - Tx

Code up the entire Tx path - with the exception of DMA mapping

Take a packet from networking stack

- ndo_start_xmit handler

Fill descriptor from skb - data and skb frags

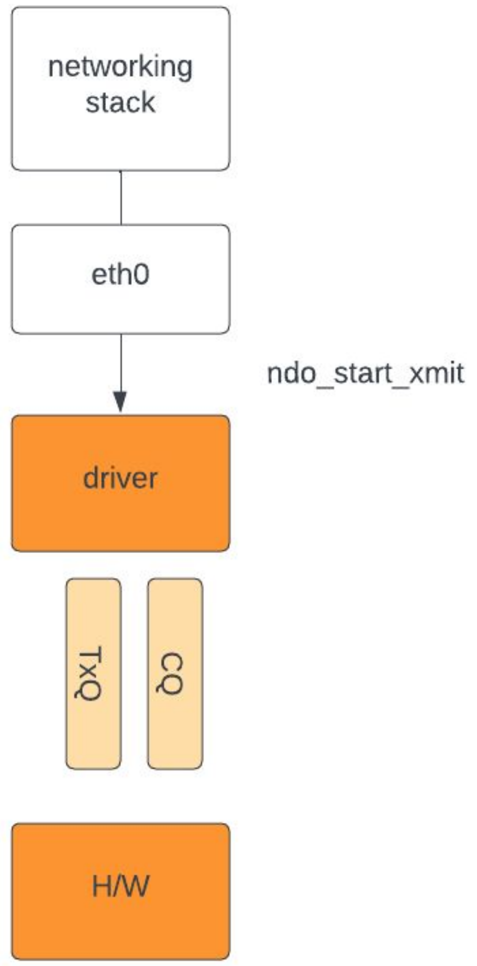
- Address in descriptor needs a wrapper
    - S/W model wants virtual, H/W wants physical or DMA
- Make DMA mapping a helper - empty for S/W model

Save reference to skb waiting for completion

Move pidx based on xmit_more flag

Implement Tx completion handling

- Free skb which for TSQ pushes another one down the stack



S/W Model: Data Path - Tx

kthread for S/W model polls Tx queues for pidx moves

- Model for Tx scheduling decides how many packets to pull from each Tx before moving on to the next

S/W model converts descriptor or packet into iov (leverage kvec and its helpers)

- peer is tap device: call kernel_sendmsg with iov; handoff packet to tap driver
- peer is same type: iov from Tx gets run through H/W model and put into Rx queue and descriptor filled

Sends completion entry

- allows driver to code up completion handling (freeing skb)

Data Path - Rx

enfabrica

Code up the entire Rx path

- napi setup during netdev initialization
- only shenanigans for the S/W model is the irq to napi_schedule handling

napi poller scheduled

- Process Rx ring
- Inject packets into networking stack
- Push new buffers

S/W model

- Pull packet from “wire”
    - for tap: pull skb entry from ring, convert to iov
    - for peer device: convert descriptor entry to iov
    - Run iov through H/W model and land data in buffers and push descriptor entry

Data Path - Rx

H/W will DMA into posted buffers

S/W model will need to memcpy using CPU

- Limits throughput performance of S/W model (~15-30G on a modern DC CPU based on MTU)
- S/W model can cheat for say iperf3 packets - payload is irrelevant so skip the memcpy
  - Allows higher throughput and cycling through rings

Build Your Custom S/W Layers

Tracepoints in the control and data paths

- Starting working on debug infrastructure for your driver

Start working on upper layer S/W

- Working driver - hides most H/W details
- e.g., IB driver, IB provider, *CCL plugins, etc

Advanced Use Cases

enfabrica

netdevice for tap or custom device (e.g., called enf here)

Connect tap or “enf” device to bridge

- L2 connectivity between peer sets within a host
- L2 connectivity between VMs (e.g., running a qemu based device model)

Route between tap or enf devices

- Host namespaces with a router namespace
  - Leverages Linux to do packet forwarding

Similar to what can be done today with veth

- Test configurations based on veth should work with your custom device

Demo

enfabrica

Summary

Driver development and features do not need to be serialized behind formal H/W model

S/W only models allow driver development in a simpler environment

Hide the h/w details behind a HAL

- Wrapper PCI calls, DMA mapping, MSI-X handling
- No-ops for S/W model

Allows easier stress testing of S/W for random “H/W” failures

- Example: what happens to a packet socket app if a completion does not arrive from H/W?
    - hint: either stuck process or cpu lockup

Thank You

Summary

S/W only models allow driver development in a simpler environment

- Driver development and features do not need to be serialized behind formal H/W model

Allows easier stress testing of S/W

- e.g., what happens to a packet socket app if a completion does not arrive from H/W

Abstract the PCI API calls

Best to tackle one direction at a time (Tx is easiest to start, then Rx)

- TSO slicer and checksum offload an excellent example
- tap device helps with that
- Includes when integrating a more precise H/W model based on qemu

Access to packets at strategic points helps

- ingress to device model (packet taken from wire)
- egress from device model (packet put on wire)