Netdev 0x16

IDPF

Infrastructure Datapath Function

Team Intel

intel®

What :

- An Open Industry-standard developed by Intel & Google "IDPF" – Infrastructure Data Path Function for PCIe

  - High-Performance Data Path
  - Ethernet & RDMA* capable
  - Broad O/S Ecosystem support
  - NIC, bare metal, VMs & containerized usages
    - Composed in any manner desired, SR-IOV, SIOV or fully/partially emulated devices
    - Presented to the O/S as a physical Function (PF) or a virtual function (VF) PCIe device, as needed
  - Container dedicated N/W Interfaces (Container Dedicated Queues CDQ)
  - Capability-negotiated & extensible
  - Can support non-native IDPF hardware
  - Supports Live Migration

* Feature to be added after Base V1.0 features

Why:

- Need for high performance feature rich Generic Network Interface as current generic Interface (virtio 0.95/1.1) does not scale very well for high performance.
- Have a HW optimized PCIE footprint: small and just for fast path registers.
- RDMA support
- Advanced features for the Cloud

PCIE Fast path virtio/IDPF comparison

	Virtio 0.95/1.0 (split queue , out of order)	Virtio 1.1 (compact queue , out of order)	IDPF (split queue , out of order)
TX PCIE accesses by the device	**Submission** •Read DBL from host memory - once in a batch or few batches •Read avail ring – once in a batch. •Read desc ring – once in packet •Read net_hdr – once in a packet •Read header/payload… Latency - 2 extra reads (avail + net hdr) comparing to IDPF addition extra read once in a while for the DBL from host memory Prepended Net_her is 14B long which creates unaligned access. **Completion (2B)** Write completion ring – once in batch	**Submission** - ~~•Read avail ring – once in a batch.~~ •Read desc ring – once in batch •Read net_hdr – once in a packet •Read header/payload… Latency - 1 extra reads (net hdr) comparing to IDPF Prepended Net_her is 14B long which creates unaligned access. **Completion (16B)** Write completion ring – once in batch	**Submission** - ~~•Read avail ring – once in a batch.~~ •Read desc ring – once in batch ~~•Read net_hdr – once in a packet~~ •Read header/payload… **Completion (4B)** Write completion ring – once in batch
RX PCIE accesses by the device	**Submission** •Read avail ring – once in a batch. •Read desc ring – once in packet Latency - 1 extra reads (avail) comparing to IDPF **Completion** Write net_hdr – once in a packet Write used ring – once in batch Prepended Net_her is 14B long which creates unaligned access.	**Submission** ~~•Read avail ring – once in a batch.~~ •Read desc ring – once in batch Latency - Same as IDPF - **Completion** Write net_hdr – once in a packet Write used ring – once in batch Prepended Net_her is 14B long which creates unaligned access.	**Submission** •Read desc ring – once in batch - - - - **Completion** Write used ring – once in batch

How:

- A proposed TC @ Oasis
  - https://lists.oasis-open.org/archives/oasis-charter-discuss/202210/msg00000.html

- Make IDPF an open Industry-standard (Spec, Driver, SW Backend, LM support)
  - Host Interface,
  - Device Behavior,
  - Setup and configuration flows
  - A single community driver to go along with the Interface and support the devices

Why IDPF : Network Equivalent to NVMe

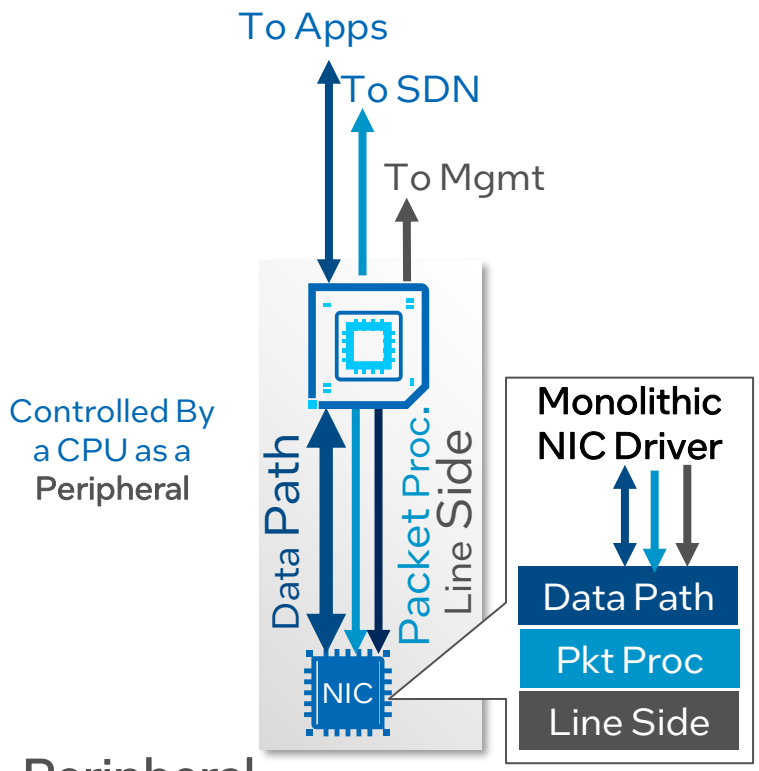
- Standard Host Interface for Networking
  - High Performance, Feature-rich NW device interface
  - Standardized Physical & Virtualized Device Types
  - Supports VMs, bare metal & containers
    - ✓ Container Dedicated NW Interfaces/Queues
- Supports VM Live Migration & Localhost Socket
- Supports PCI Hot Plug
  - Uses semantics from KVM
- Software and hardware backends
  - Can support non-native IDPF hardware w/ acceleration
- Implementation Independent Networking + RDMA*
  - RDMA* Transport Independent (Determined by the implementation)
- Optimized Descriptor Format
  - Capable of Over 200Mpps, 200Gbps using the Linux kernel driver
  - Hardware validated stateless offloads (TSO, CSUM, RSS, RSC, SO_TXTIME)
- Compatibility & Capability Testing as part of ipdk.io

NETWORK	STORAGE
Virtio-net • Backwards Compatible w/ existing VMs	**Virtio-blk, Virtio-scsi** • Backwards Compatible w/ existing VMs
IDPF • Offers higher performance • RDMA*	**NVMe** • Offers higher performance • NVMe controller feature set like Namespaces

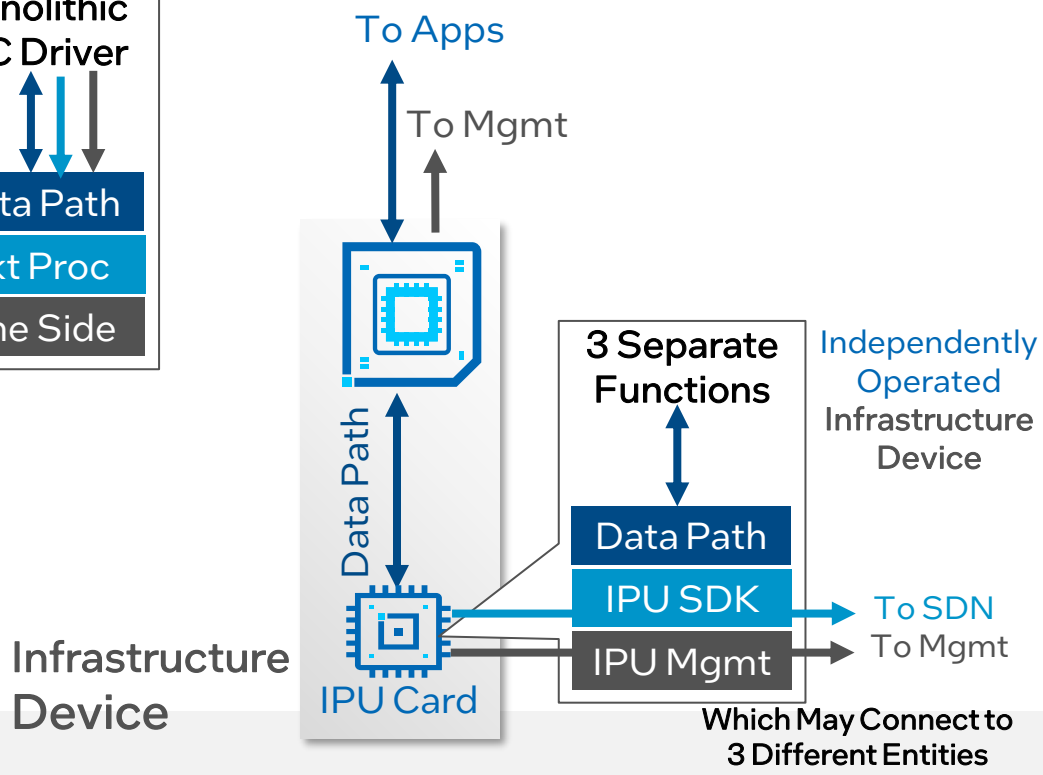
IDPF
Open Standard Spec, Broad OS Driver Support, HW & Software Backend, Bare Metal/Virtualized usage

* Feature to be added after Base V1.0 features

IDPF: Infrastructure Data Path Function

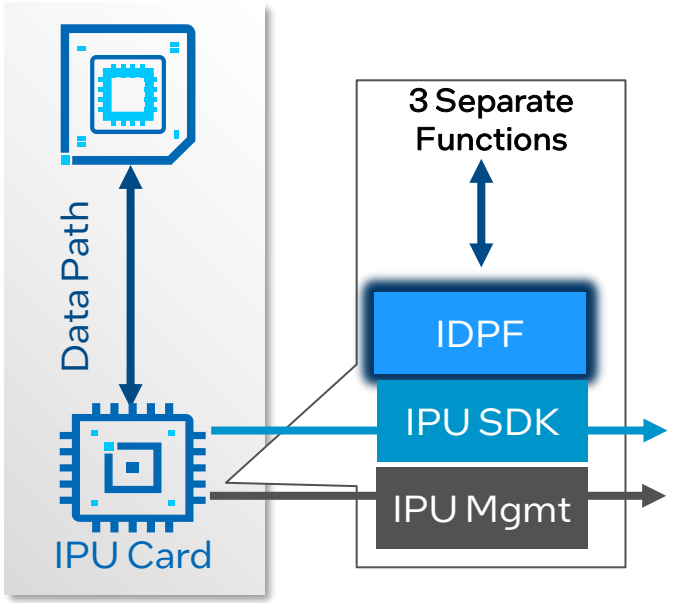


Peripheral Device



Infrastructure Device

Infrastructure Device w/IDPF

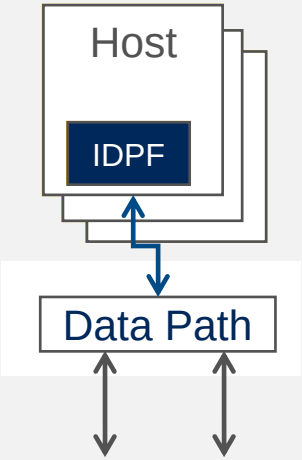


Data Path Function to Hosts & Guests

Retains Feature Set & Validation Across OSes of Foundational NICs

Where Can IDPF Run?

Physical NIC

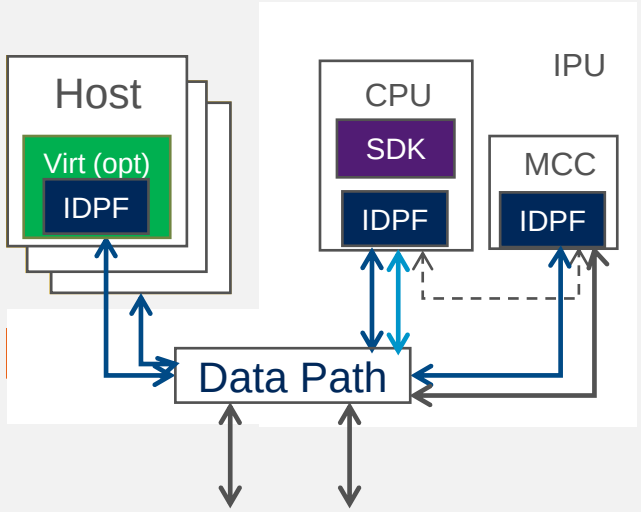


Data Path: Supportable from all hosts, guests
Main & IPU CPUs using IDPF

Line Side: Physical Ethernet Links

Host: CDQ, SRIOV, Guests can have CDQs

Bare Metal Hosting



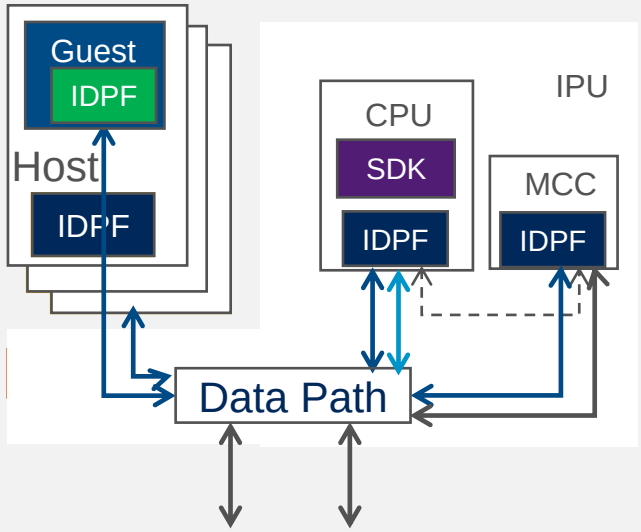
Data Path: Supportable from all hosts, guests
Main & IPU CPUs using IDPF
Host can be physical IDPF or
composed virtual device (eg virtio)

Line Side: Physical Ethernet Links

Line Side Proxy (Dotted): Used in IPU

Host: CDQ, CMS Hotplug

VM Hosting



Data Path: Supportable from all hosts, guests
Main & IPU CPUs using IDPF
Guest uses an emulated IDPF
accelerated with Host IDPF devices

Line Side: Physical Ethernet Links

Line Side Proxy (Dotted): Used in IPU

Host: Guests can have CDQs

* "MCC" – Management Control Complex – System Management, Monitoring, Manageability, etc.

IDPF Feature Set

Broad OS Support

- DPDK 0x1452 (PF), 0x145c (VF)
  - ✓ 0X1453 Runs same PMD as IDPF, plus MMIO for IPU SDK
- Linux: 0x1452, 0x145c
- ESX : 0x1452

Planned

- ESXio
- Windows
- CPFL

Core Features (All OSes)

- TX/RX on Multiple Queues
- Stateless offloads:
  - CSUM, TSO, RSS
- Jumbo frames
- Locally Admin. MAC Address
- PXE Boot
- Capability Negotiation

Linux Specific Features

- Line Side Proxy
- AF_XDP
- RSC
- VLAN Add/Strip
- RDMA*
- Header Split
- Earliest Departure Time (EDT)
- Inline Ipsec
- PTP

* Feature to be added after Base V1.0 features

Infrastructure Datapath Function Driver

IDPF : How is it different from iavf

- iavf
  - Intel's Adaptive Virtual Function driver for Foundational NICs
  - VF resources are managed by the PF driver
  - Uses Virtchnl 1.x API over a control channel between VF and PF for resource configuration
- idpf
  - Vendor Neutral Infrastrastructure Datapath Function driver for IPUs/DPUs/FNICs
  - Acts as a driver for PF/VF instances exposed to the host
  - Host PF/VF driver resources are managed by the Control Plane running on the device
  - New Control channel between the driver and the device using Virtchnl 2.0 API for capability learning, negotiation and resource configuration

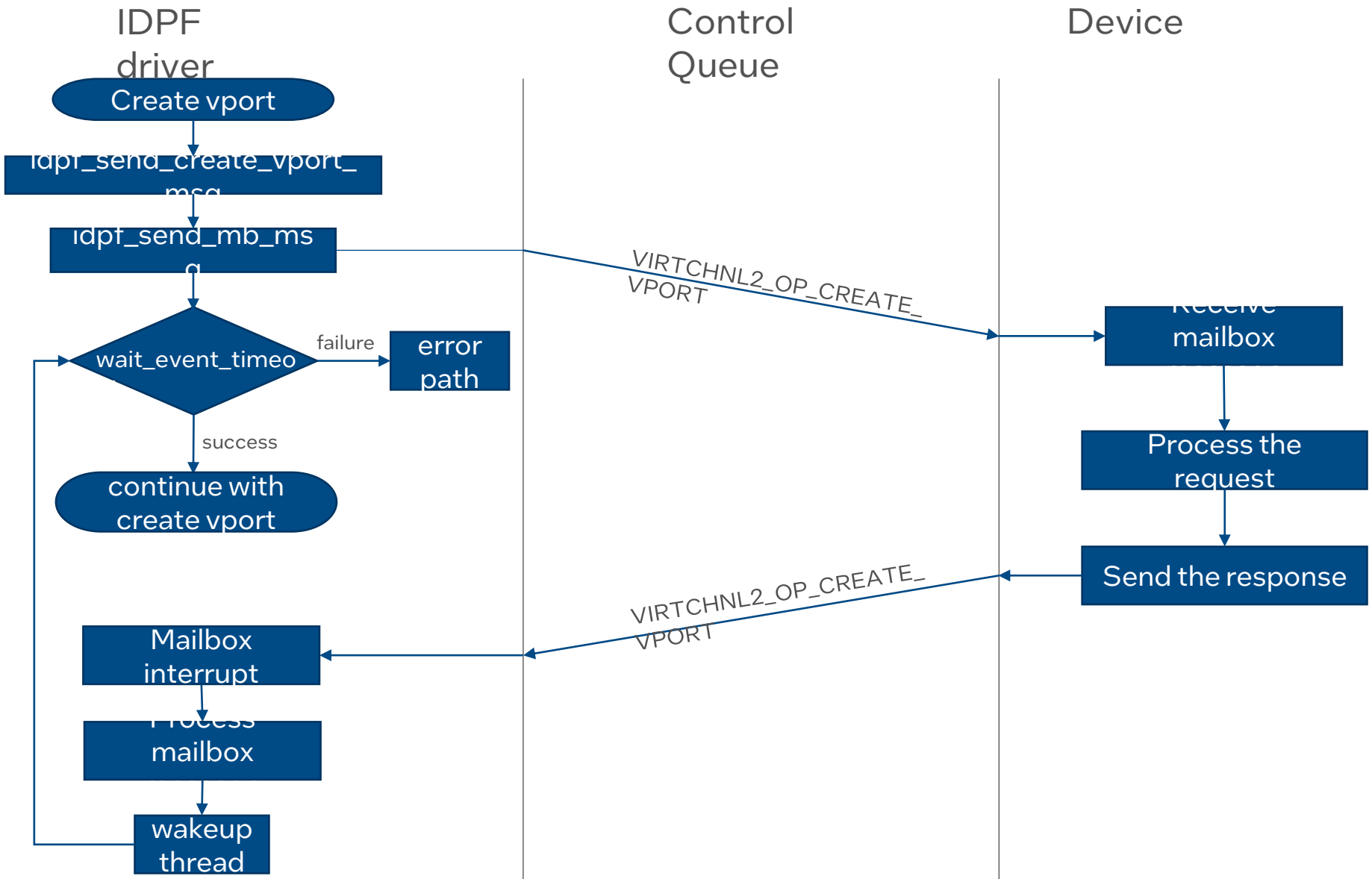
IDPF – What is New

- Granular and Negotiated Capabilities
  - Allows the Control Plane to expose the capabilities based on device features and configured policy for the instance.
  - Checksum, Segmentation, RSS, HW GRO, Header Split etc
  - Number of Vports, Queues, Interrupt Vectors
- Learn device register offsets, descriptor formats
- New TX/RX Flows to support
  - Split Queue Model (RX, RX-Buffer, TX, TX-Completion)
    - Large number of RX queues with reduced host memory footprint
    - Single writer for each queue (SW or HW)
    - Buffer Queue Groups (large and small buffer queues)
  - Per Flow QoS
    - Out of Order TX Completions
    - Early Departure Time Support
  - Receive Segment Coalescing (HW GRO)

Driver Initialization Sequence

- Get Version
- Get Capabilities
- Allocate Vectors
- Create Vport(s)
- Configure TX Queues
- Configure RX Queues
- Map Queues to Vectors
- Configure RSS
- Enable Queues
- Enable Vport

IDPF and Device Control Plane interaction



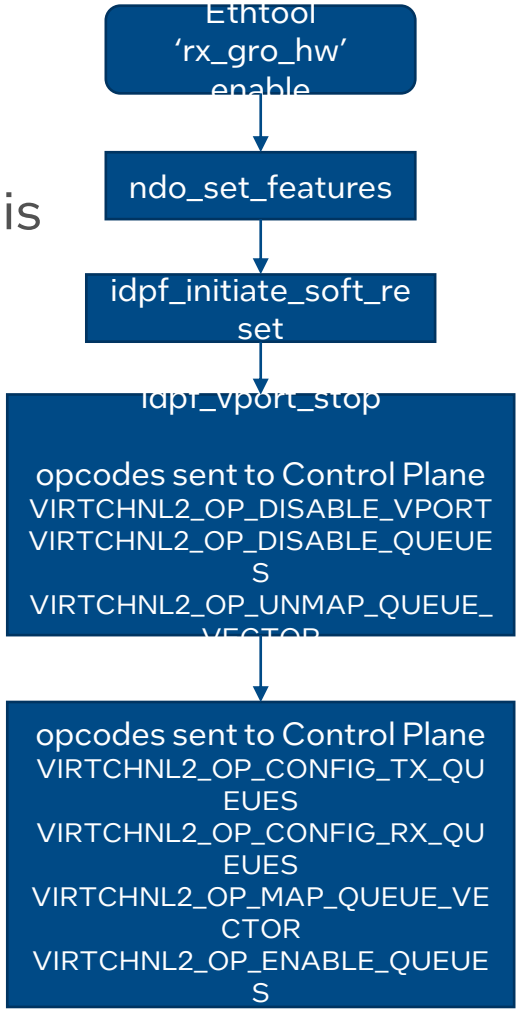
Driver flow to enable RSC (HW GRO) feature

Enable RSC:

1. Driver requests RSC capability to the control plane
2. Set 'NETIF_F_GRO_HW' in netdev features if the capability is enabled

Update queue configuration:

1. User enables 'rx-gro-hw' using ethtool
2. Call back into the driver 'ndo_set_features'
3. Initiate a soft reset
   - Update the RX queue context to enable RSC



Processing the RSC (HW GRO) packet

- Device uses large buffers (4K) to coalesce packets
  - Max coalesced payload size is 64K (16 buffers)
- Reports RSC segment length in the descriptor
- Driver updates the skb
  - NAPI_GRO_CB(skb)->count = rsc_segments;
  - skb_shinfo(skb)->gso_size = rsc_payload_len;
  - skb_shinfo(skb)->gso_type = SKB_GSO_TCPV4/V6;
  - skb_reset_network_header(skb);
  - skb_set_transport_header(skb, sizeof(struct iphdr/ipv6hdr));
  - tcp_v4/v6_check
  - tcp_gro_complete(skb)

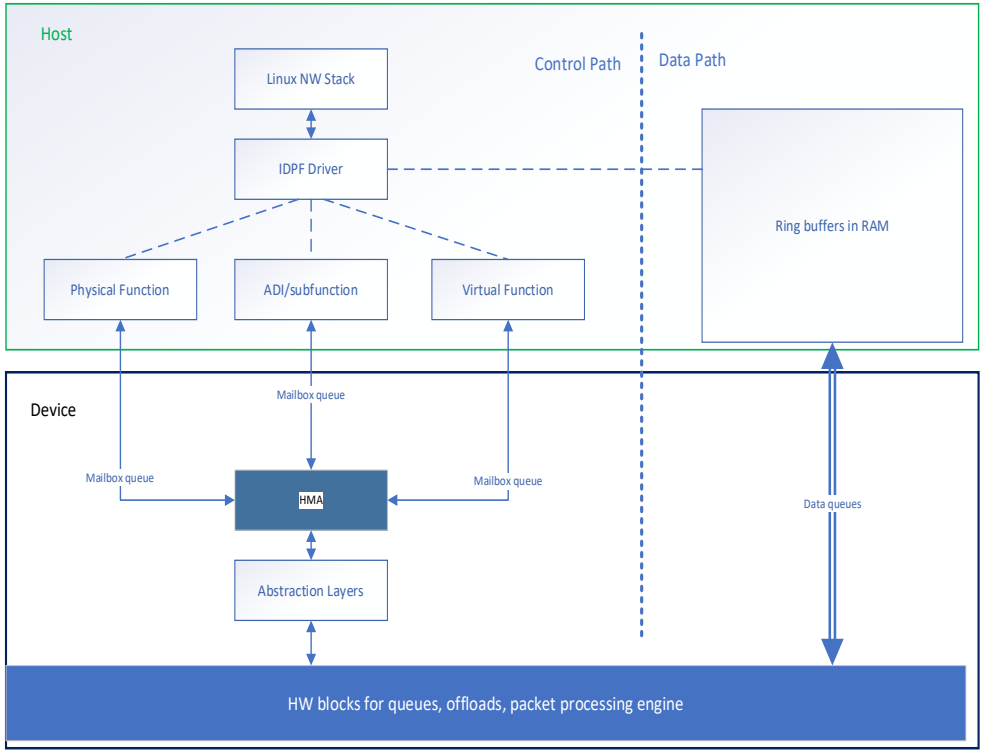
Host Management Agent

Netdev 0x16

Agenda

- What is Host Management Agent?
- Why we need it?
- How does it enable config path?
- Sample flows
  - ✓Basic flow – Create a vport
  - ✓Adv flow - Traffic shaper and EDT

What is HMA

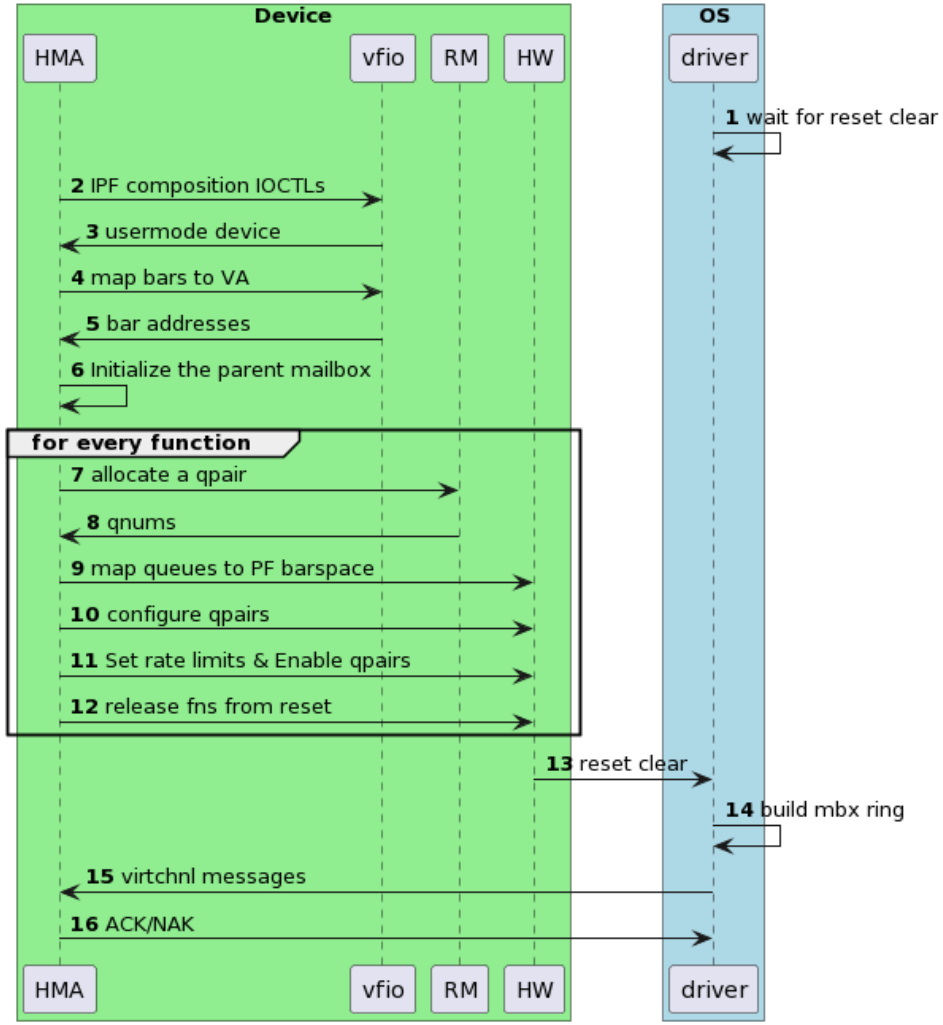


- A component of the device control plane
- Owned by infrastructure provider
- User mode or kernel mode
- Opens up communication channel with host functions
- Implementation of abstract virtchnl APIs
- Resource manager
- Configures scheduler to ensure fairness/qos

Why we need it?

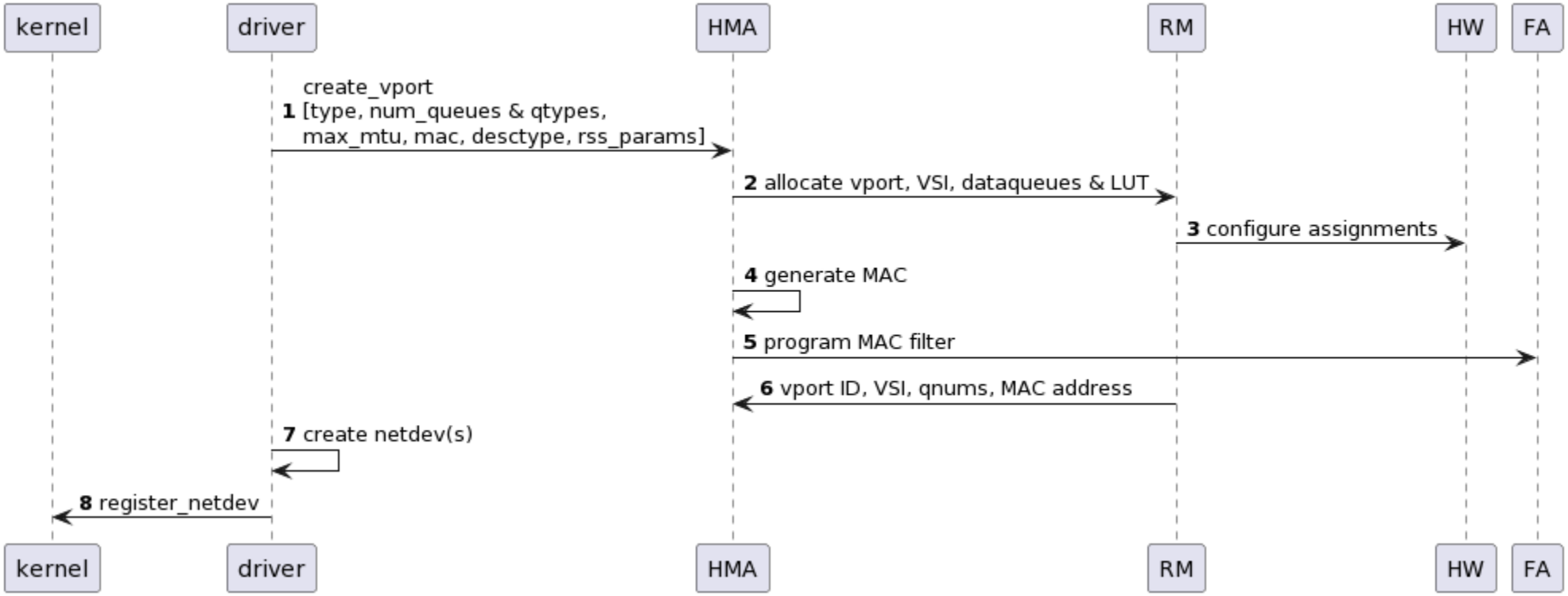
- Enables the split between control path and data path
- Separate channel allows smaller BAR space for host functions
  - Each queue uses the following registers: qlen, head, tail, bah & bal
- Client-Server model enables central management of policies
  - Loaded from filesystem and downloaded from cloud agents during init
  - Allows provisioning VMs differently
  - Fine grain capabilities, negotiable by host drivers
  - Flexible control of resource distribution
- Promotes reuse of host driver for PF/VF/ADI
- Host Driver reuse across multiple silicons/vendors
- ODMs/OEMs can provide deployment specific implementation

HMA initialization



1. Driver waits for the function to become ready
2. VFIO ioctls to retrieve device/region information
3. Store region details
4. Map the bar regions to process address space
5. User virtual address mapped to the bar of underlying device
6. Initialize the parent mbx ring
7. Device specific allocation for queues
8. Allocated queue numbers
9. Map the queue as mailbox in known location
10. Configure the parameters of the queue
11. Release all the functions from reset
12. Request HW to release function
13. Bring the function out of reset
14. Build the default mailbox
15. Exchange of config path messages
16. Processing of requests from driver

Basic flow - Create vport



- Common flow across PF/VF/subdev/SIOV instances
- Small memory footprint to acquire/configure resources

Packet Pacing/ Traffic Shaping

- Reference for EDT and Traffic Shaping:

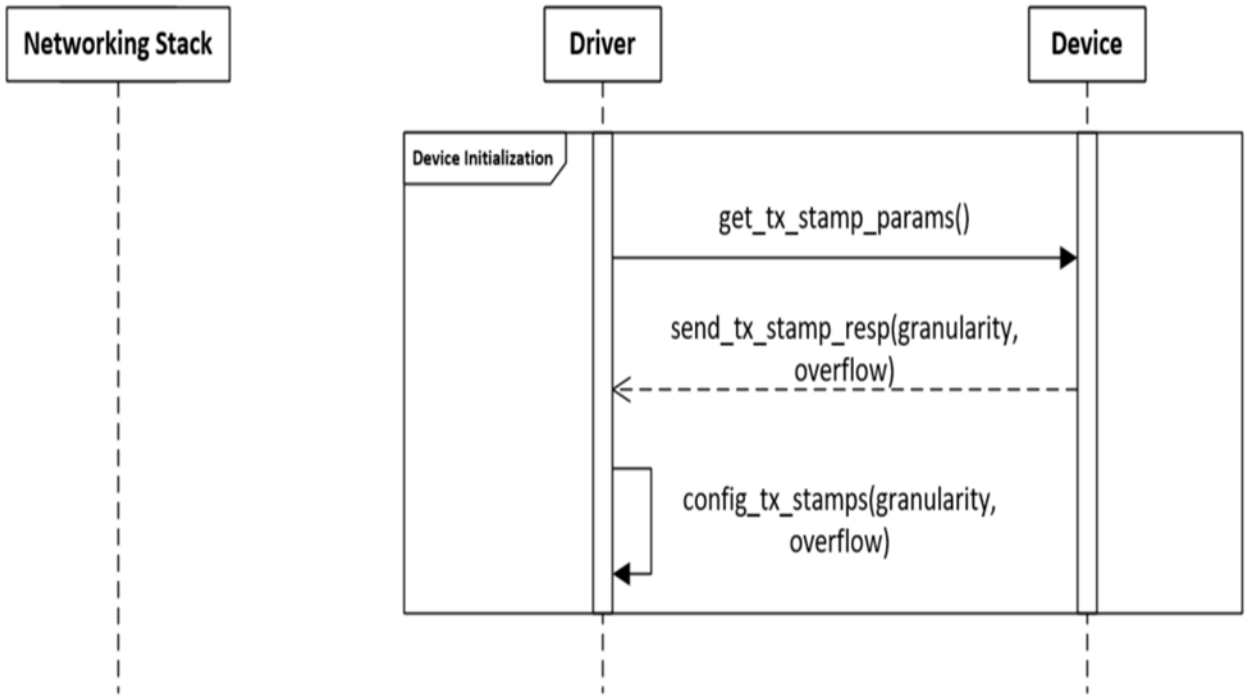
https://legacy.netdevconf.info/0x12/news.html?keynote-van-jacobson-evolving-from-afap-teaching-nics-about-time

Advanced flow: Traffic Shaper and EDT

- Traffic Shaper (TS) implements egress traffic shaping and can delay packet transmission in order to provide shaping.
- TS can buffer packet headers, metadata and associated scatter/gather lists (SGLs) in packet Header Storage implemented in memory.
- EDT support can be HW offloaded as TS can hold data transmission until departure time (Tx timestamp in skb) has reached.
- Since device holds data transmission and transmits when timestamp expire, order of packet descriptor fetch completed may be different than packet data fetch is completed. So, device needs to support out of order completion for packet data.
- To support out-of-order, split queue model is introduced in device, where Tx queues are used to pass buffers from SW to HW, while Tx completion queues are used to pass completion from HW to SW.
- Completion queue gets 2 completion notifications, one for descriptor fetch completion and other one for data fetch completion
- Early descriptor fetch notification enables SW to reuse Tx descriptor queue slot and write a new packet descriptor to Tx queue.
- Timestamp in completion can be used to provide feedback to Networking stack or Shaping SW.
- Shaping SW or congestion control algorithm running on Host OS puts EDT timestamp in packet.

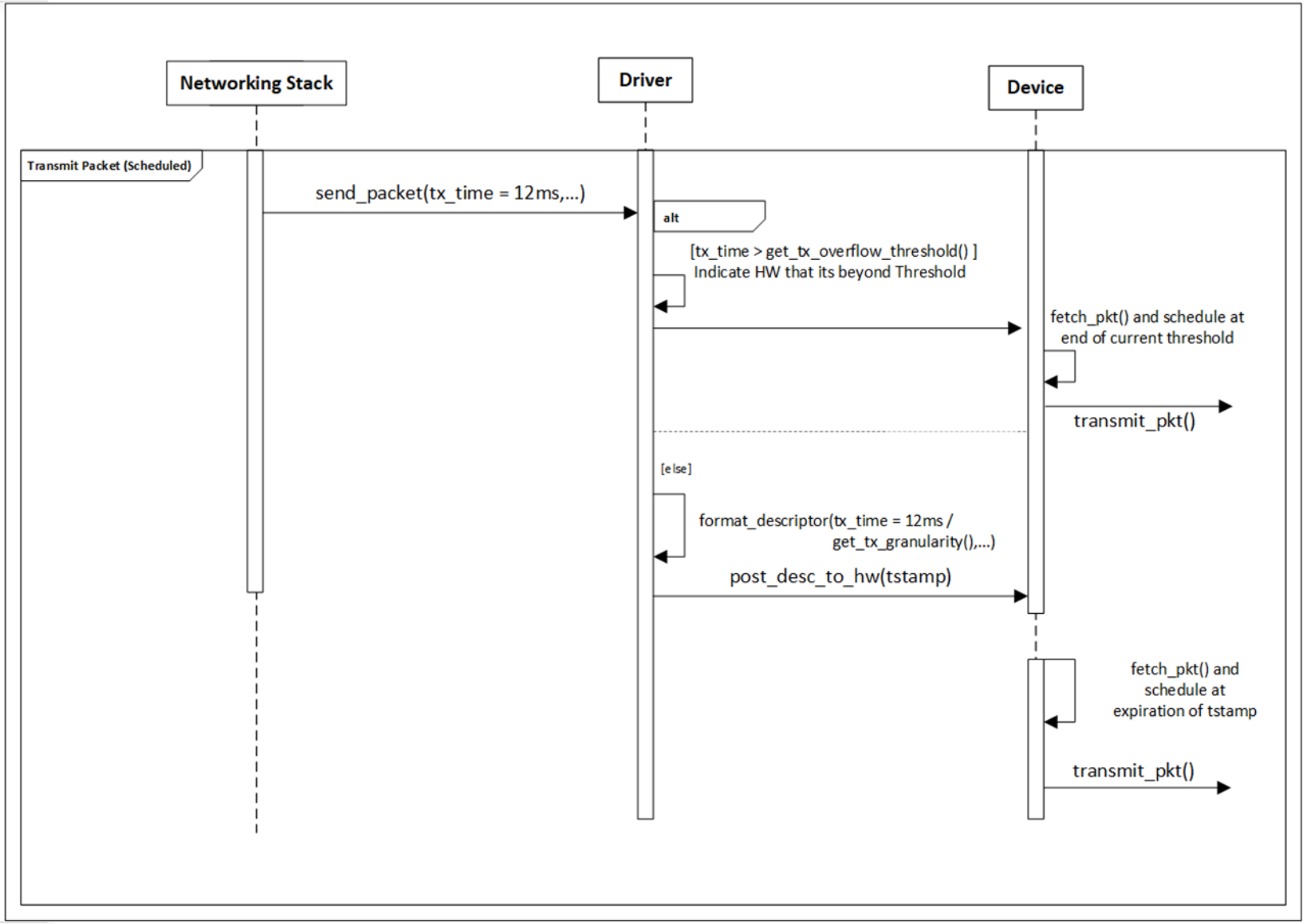
Traffic Shaper and EDT (contd..)

- TS supports multiple timestamping parameters such as granularity and time range for which packets can be held in memory (overflow threshold).
- Device control plane configures appropriate set of timestamp parameters according to use case.
- Timestamp parameters are communicated to host driver via config channel, so that device control plane and Host driver are synchronized.



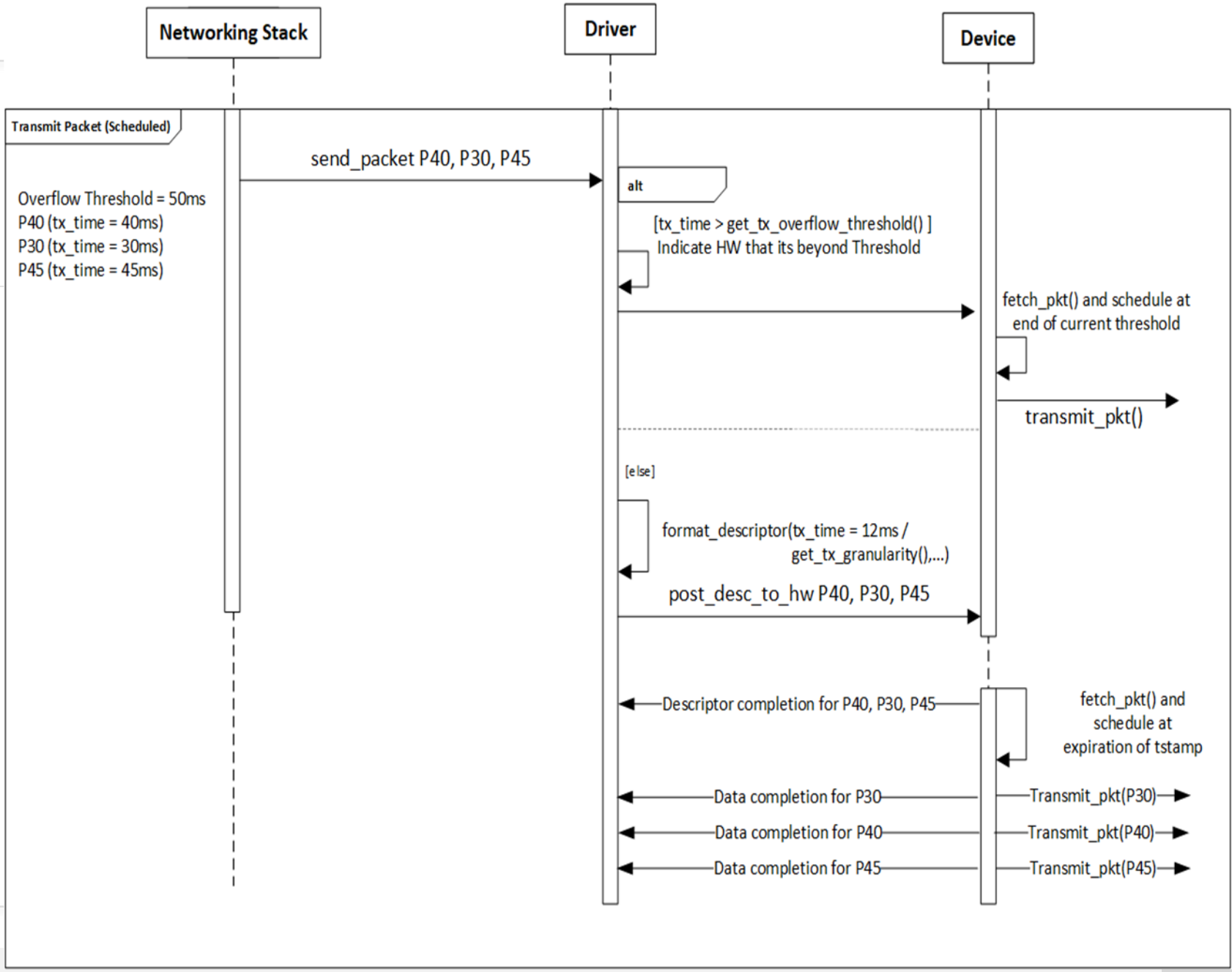
Traffic Shaper and EDT (contd..)

- When the network stack tries to schedule a packet for transmit, timestamp will use the overflow threshold to determine if the packet should be dropped. If it is below the threshold, it will be converted to the appropriate units based on the granularity.



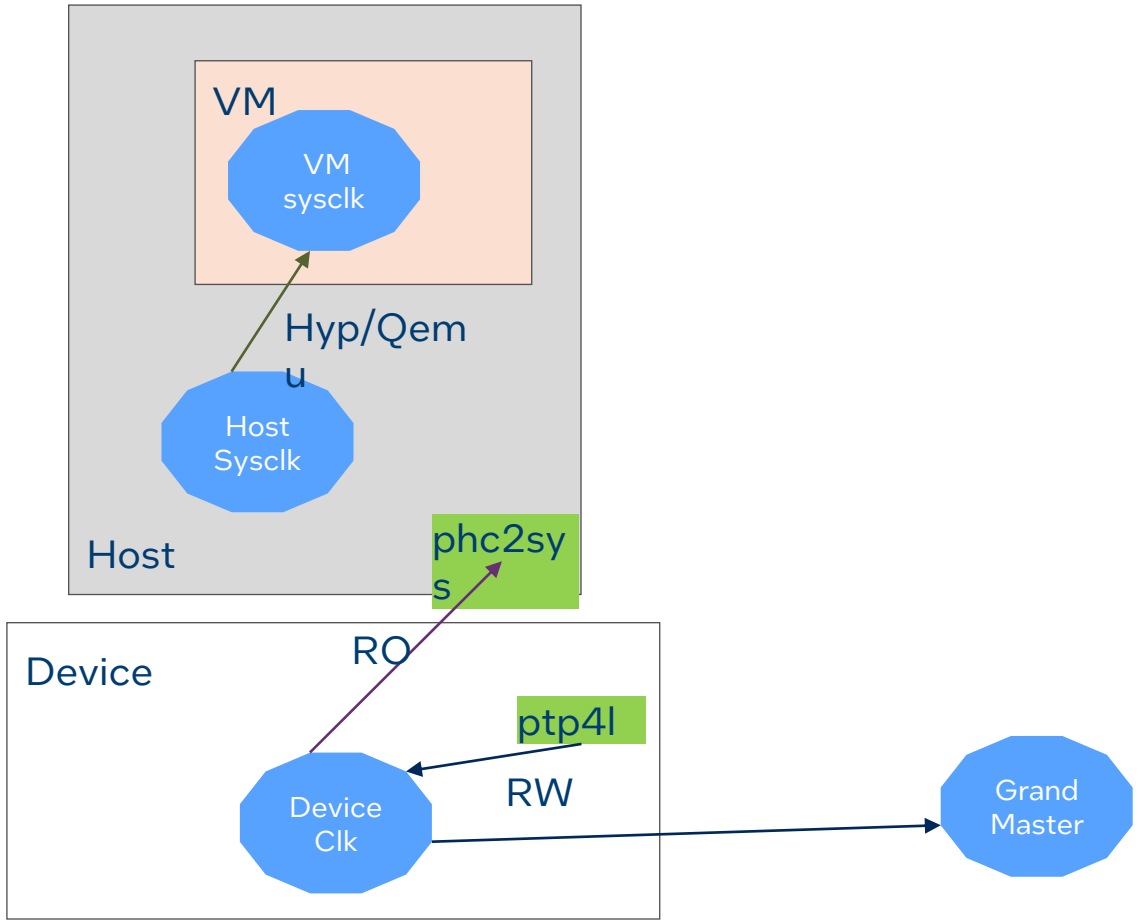
Traffic Shaper and EDT (contd..)

Out-of-order flow



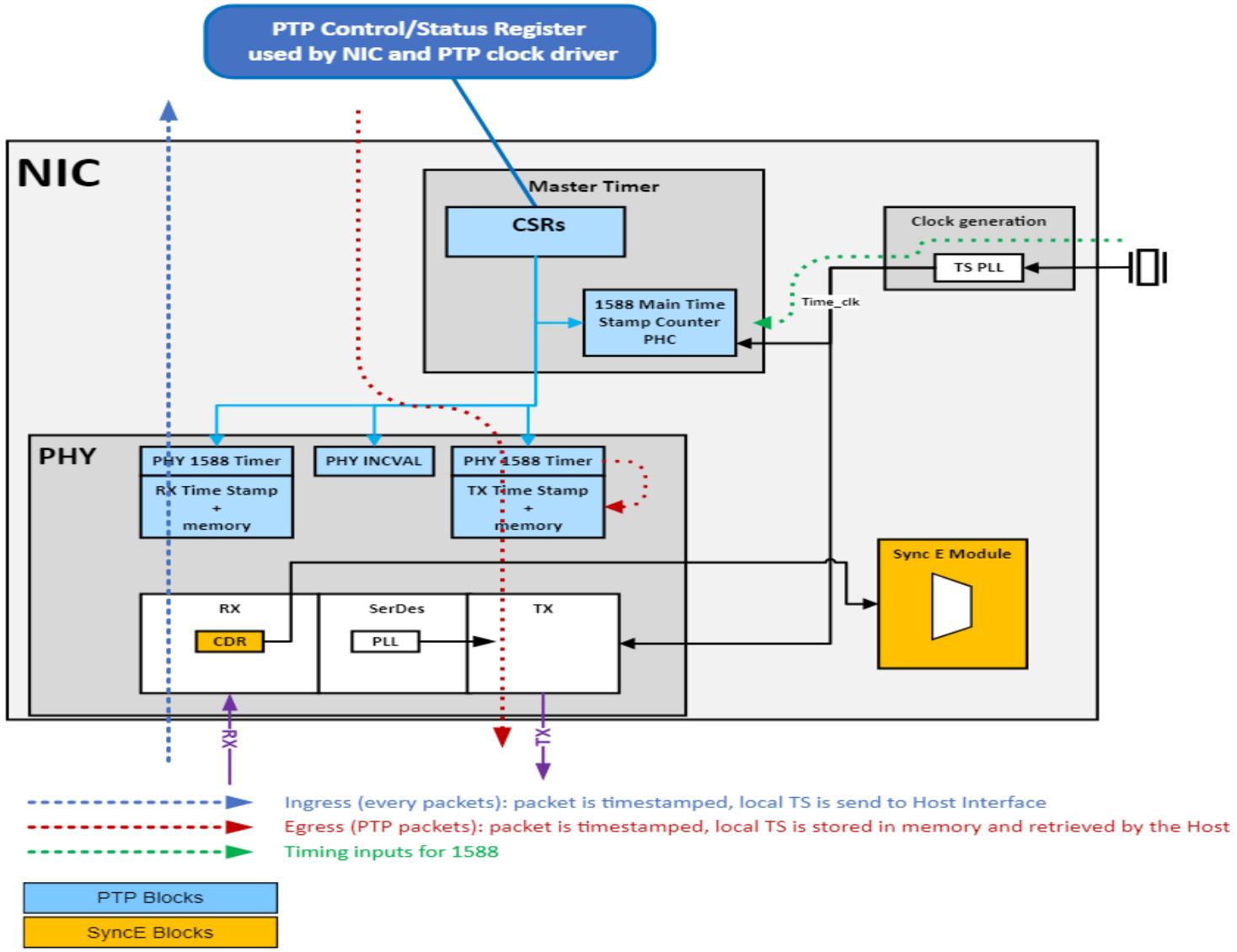
PTP Device Detail

- Device will sync its clock with GrandMaster/External world
- SW running on Device will have write control
- Host will sync system clock with Device clock
- Host will have only RD access to Device clock
- VMs will rely on Hypervisor/Qemu API to sync with Host clock



PTP Device Detail

- Only Status register access to host
- Kernel API:
  - Gettime64
    - SHTIME, SHTIME_L and SHTIME_H
  - Getcrosststamp
    - PTM(Precision Time Measurement) support required
    - ART_L and ART_H
    - SHTIME, SHTIME_L, SHTIME_H



An IDPF User space Driver

- Over VFIO
- DPDK based
  - Polling mode
  - Core affinity
  - Efficient memory management
  - Platform optimization

Device user space access by VFIO

- ioctl fds: container, iommu group, device
- Steps:

```
/* Set IOMMU type according to system*/
container_fd= open("/dev/vfio/vfio", O_RDWR);
ioctl(container_fd, VFIO_SET_IOMMU, type_id);

/* Get iommu group N according PCI device addr and add it to container */
group_fd= open("/dev/vfio/N");
ioctl(group_fd, VFIO_GROUP_SET_CONTAINER, &container_fd);

/* DMA Mapping*/
ioctl(container_fd, VFIO_IOMMU_MAP_DMA, &dma_map);

/* map BAR to user space */
dev_fd = ioctl(group_fd, VFIO_GROUP_GET_DEVICE_FD, dev_addr);
ioctl(dev_fd, VFIO_DEVICE_GET_REGION_INFO, &reg_info);
addr = mmap(NULL, reg_info.size,
          PROT_WRITE | PROT_READ, MAP_SHARED,
          dev_fd, reg_info.offset);
```

IDPF driver

- BAR access: mmapped base + fixed offset
- DMA: DPDK mempool management
- Virtchnl message
  - Mailbox setup and recv/send as DMA ring
  - timer based polling for virtchnl message handling
- Fast Path ring and RX/TX func
  - DMA ring setup on demand
  - TAIL update: mmapped base + offset from virtchnl resp
  - RX/TX in burst in polling mode

Intel® IPU/MEV DPDK Performance Report

Document completed August 17th, 2022

Testing completed Aug 10th, 2022

NEX NSWE NPS PRC DPDK Team

intel®

Notices and Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration details in this report. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

See configuration details in this report.

System under Test (SUT) Configuration

	Configuration
SUT Setup	
Platform	Supermicro X12DAi-N6
# Nodes	1
# Sockets	2
CPU	Intel® Xeon® Platinum 8380 Processor 2.3GHz
Cores/socket, Threads/socket	40/80
Microcode	0xd0002c1
HT	On
Turbo	On
Power management	Disabled
BIOS version	1.1a
System DDR Mem Config: slots / cap / speed	4 slots / 32GB / 3200
Total Memory/Node (DDR, DCPMM)	128 GB
PCH	
OS	Ubuntu 20.04.4 LTS
Version	5.13.0-41-generic

Platform: Supermicro X12DAi-N6
Processor: 2x Intel® Xeon® Platinum 8380 Processor (2.3GHz, 40 cores, 120M LLC Cache)
RAM: 128GB DDR4, 3200 MT/s
BIOS version: 1.1a

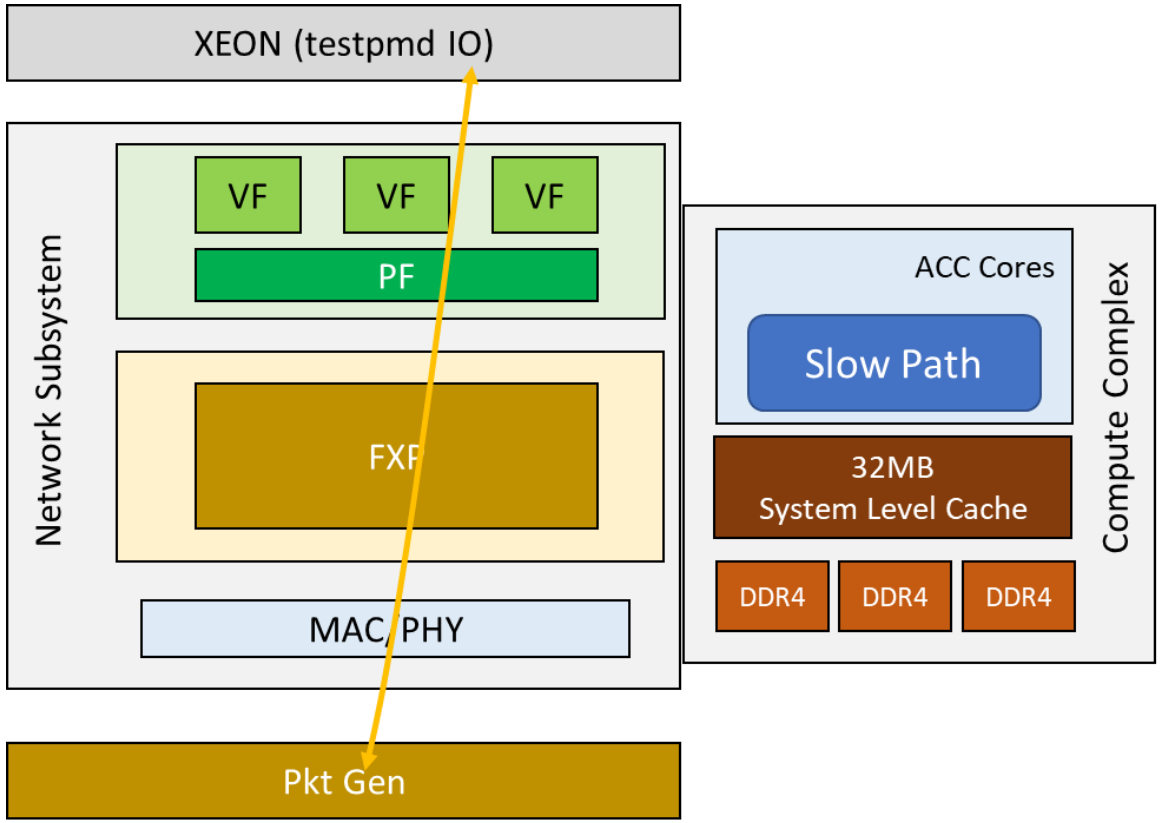
BIOS Settings:
Default except the following:
Processor Configuration -> Hyper-Threading -> Enabled for Throughput/Disabled for Latency
Power and Performance -> CPU Power and Perf Policy -> Performance
Power and Performance -> Workload Configuration -> I/O Sensitive
Advanced > Power & Performance -> CPU C State Control > Package C-State = C0/C1 State
Advanced > Power & Performance -> CPU C State Control > C1E=Disabled

MEV B0 NRB card:
CI: MEV IMC MEV-HW-B0-CI-ts.release.1627

Grub Setting: default_hugepagesz=1G hugepagesz=1G hugepages=24 console=ttyS0,115200n8 ignore_loglevel iommu=pt intel_iommu=on

Date 08/16/2022

DPDK Throughput (Xeon Host) Test Bed



Throughput Test:

Card: MEV B0 NRB card
CI: MEV IMC MEV-HW-B0-CI-ts.release.1627
Packet Generator: IXIA® 200G
DPDK PMD: idpf (internal version)
DPDK app: testpmd, iofwd
DPDK app: run on Xeon Host
DPDK scalar path: normal data path with no vectorization optimization. CPU freq is 3.0GHz
DPDK AVX512 path: optimized data path with AVX512 vectorization. CPU freq is 2.9GHz

Testing Configuration

DPDK Commands on Xeon:

- avx512-single-queue: ./build/app/dpdk-testpmd -l 1,2-2 -n 8 -a 31:00.0,representor=0,rx_single=1,tx_single=1 --force-max-simd-bitwidth=512 -- -i -a --txq=1 --rxq=1 --nb-cores=1
- avx512-split-queue: ./build/app/dpdk-testpmd -l 1,2-2 -n 8 -a 31:00.0,representor=0 --force-max-simd-bitwidth=512 -- -i -a --txq=1 --rxq=1 --nb-cores=1
- scalar-single-queue: ./build/app/dpdk-testpmd -l 1,2-2 -n 8 -a 31:00.0,representor=0,rx_single=1,tx_single=1 --force-max-simd-bitwidth=64 -- -i -a --txq=1 --rxq=1 --nb-cores=1
- scalar-split-queue: ./build/app/dpdk-testpmd -l 1,2-2 -n 8 -a 31:00.0,representor=0 --force-max-simd-bitwidth=64 -- -i -a --txq=1 --rxq=1 --nb-cores=1

Note: above is single core commands, multiple core and multiple queue commands just changed core/queue setting.

Traffic Generator Configuration:

- Transmit rate: 100% of line rate
- Packet: any mac/ fixed dst ip, random src ip/udp

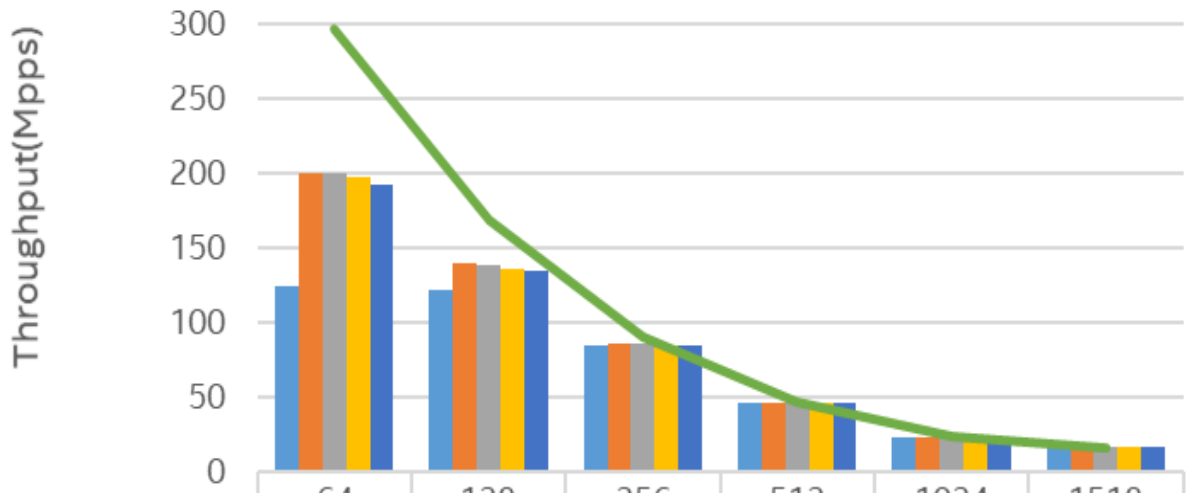
Test Case Naming:

- avx512-single-queue: use AVX512 data path with legacy queue to benchmark throughput on cores.
- avx512-split-queue: use AVX512 data path with split queue to benchmark throughput on cores.
- scalar-single-queue: use scalar data path with legacy queue to benchmark throughput on cores.
- scalar-split-queue: use scalar data path with split queue to benchmark throughput on cores.
- 200Gb line rate: 200GbE line rate in theory.

Multiple core Performance

avx512-single-queue

Higher is better



	64	128	256	512	1024	1518
1c1t-1q	125.15	122.50	84.90	46.64	23.85	16.19
2c2t-2q	200.65	140.28	86.66	46.64	23.85	16.19
4c4t-4q	200.46	138.78	85.65	46.64	23.85	16.19
8c8t-8q	197.28	136.01	84.84	46.64	23.85	16.19
16c16t-16q	192.03	134.85	84.72	46.64	23.85	16.19
200G line rate	297.62	168.92	90.58	46.99	23.95	16.26

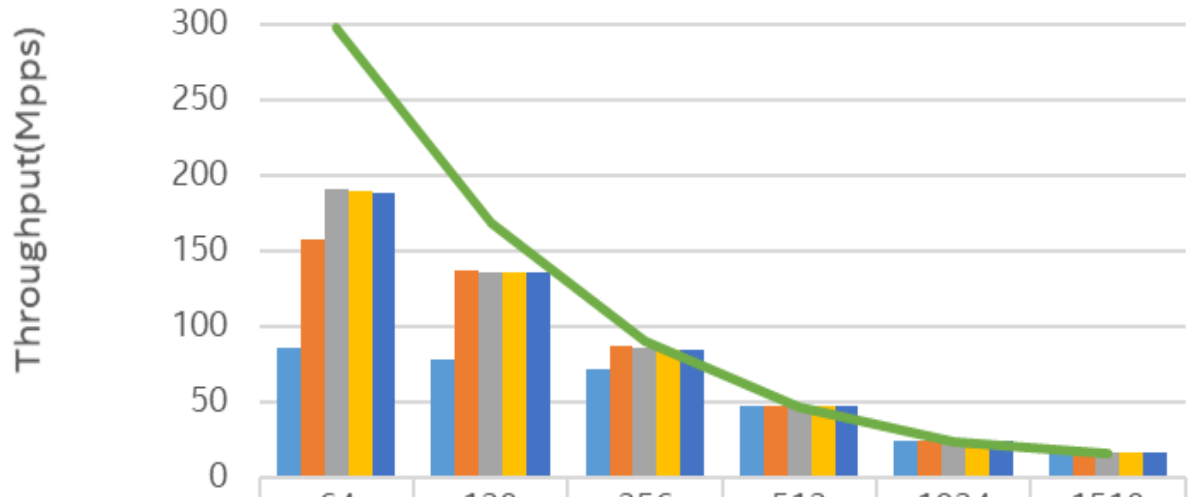
- Up to 200Mpps.

See configuration details in this report. Results may vary.”

Multiple core Performance

avx512-split-queue

Higher is better



	64	128	256	512	1024	1518
1c1t-1q	85.58	77.99	71.12	46.64	23.85	16.19
2c2t-2q	156.98	136.53	86.95	46.64	23.85	16.19
4c4t-4q	190.37	136.28	85.96	46.64	23.85	16.19
8c8t-8q	189.46	135.82	85.08	46.64	23.85	16.19
16c16t-16q	188.68	135.39	84.94	46.64	23.85	16.19
200G line rate	297.62	168.92	90.58	46.99	23.95	16.26

- Up to 190Mpps.

See configuration details in this report. Results may vary.”

Netdev 0x16

IDPF backend

Miao Li, Chenbo Xia

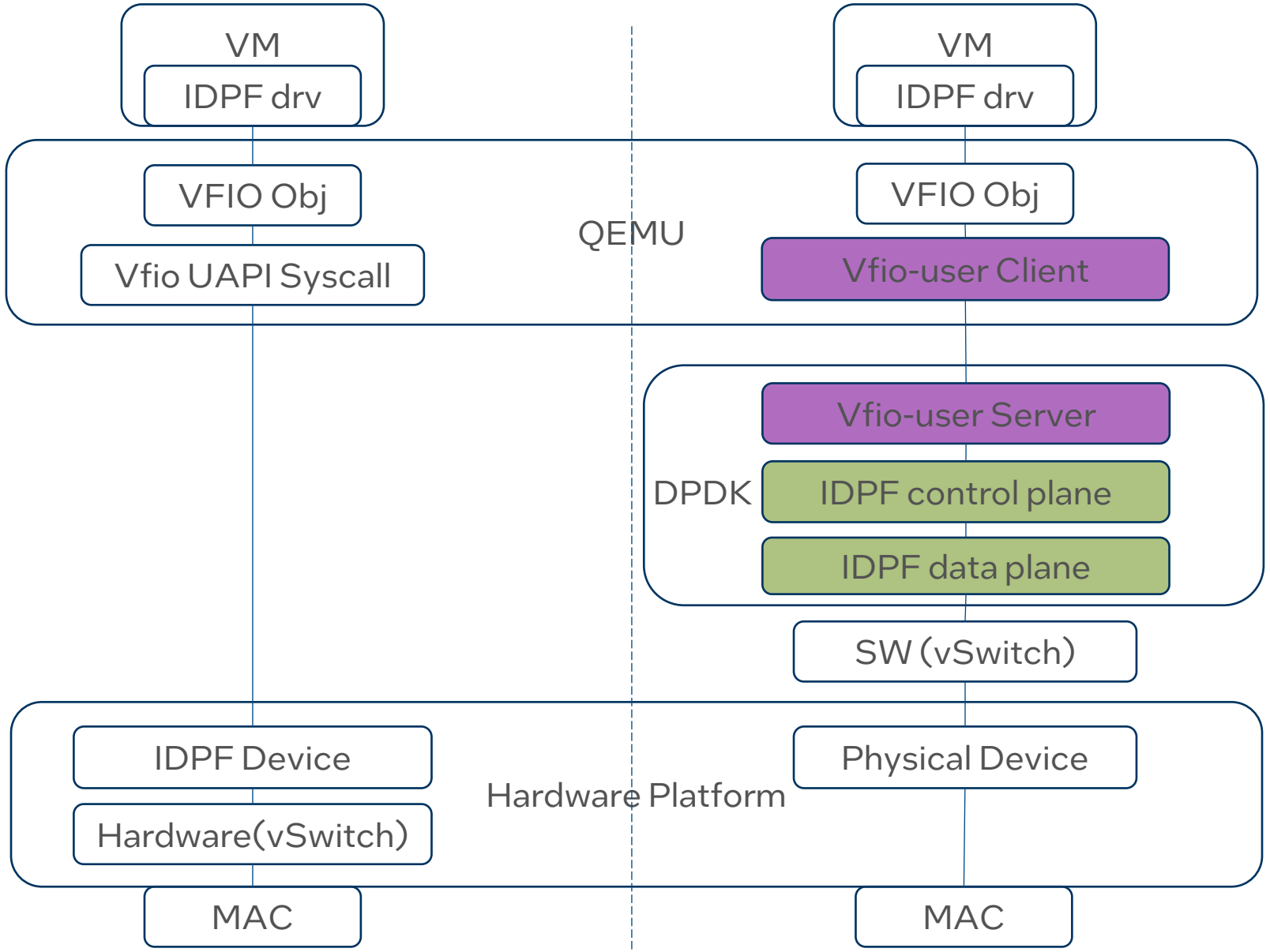
intel®

Device Emulation of IDPF

- IDPF: a vendor neutral device interface that provides a single network interface for hosts, containers and guests

- Emulated device of IDPF (IDPF backend) : a vendor-agnostic software emulated device to provide IDPF compatible layout
  - Simulation of new hardware features
  - Deployment on multi-vendor environment

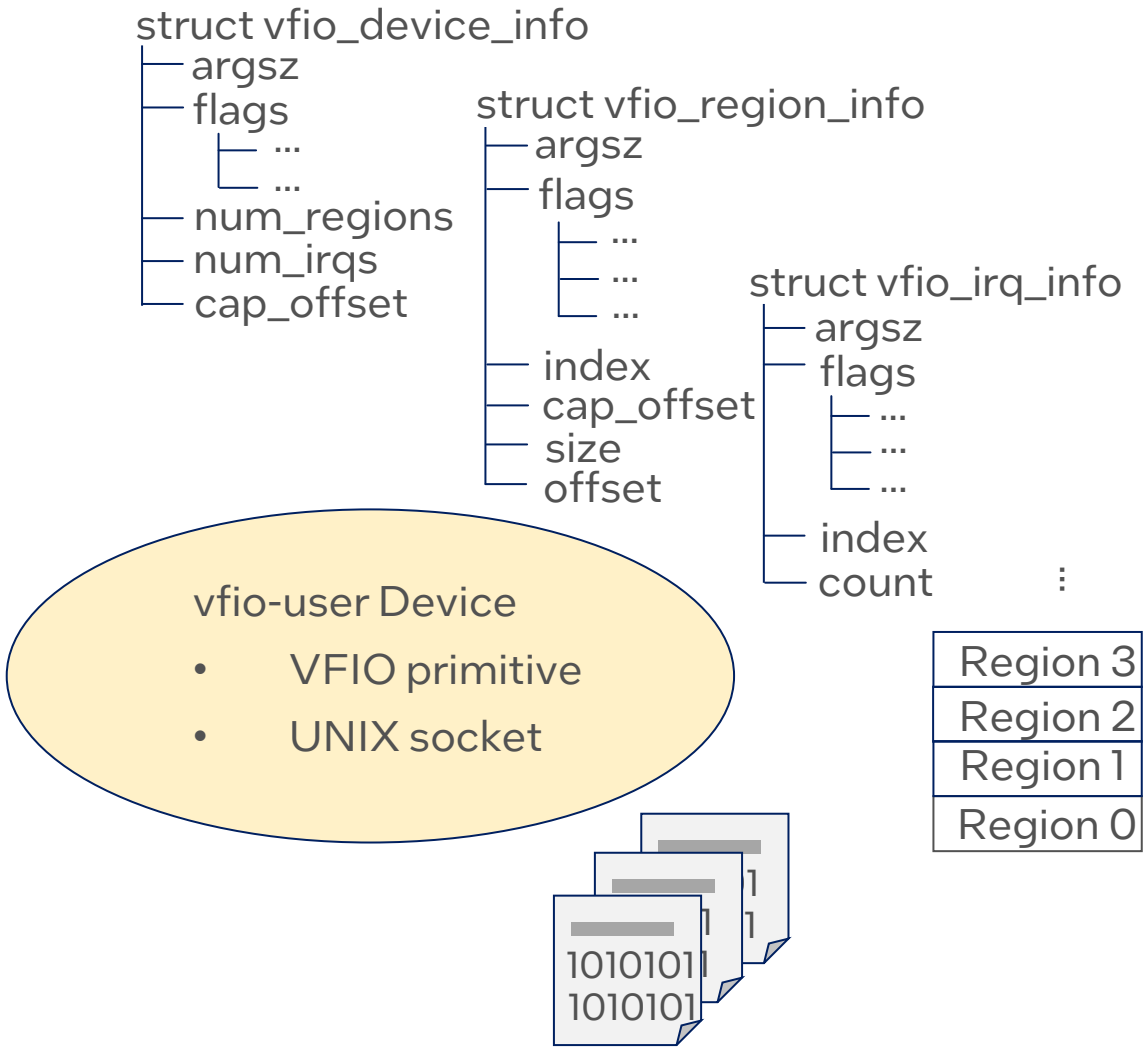
Device Emulation of IDPF

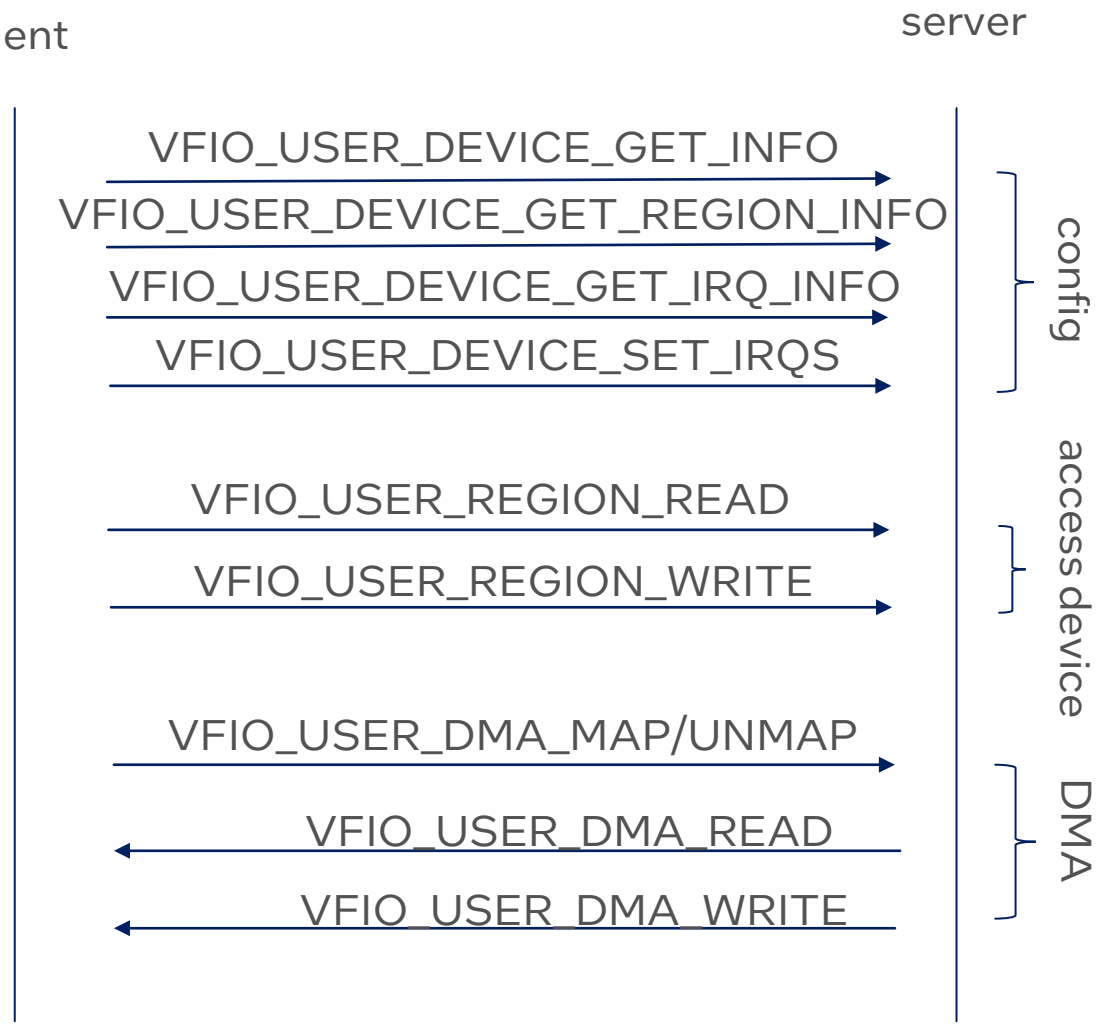


vfio-user protocol: allow a device emulation in a separate process out of QEMU

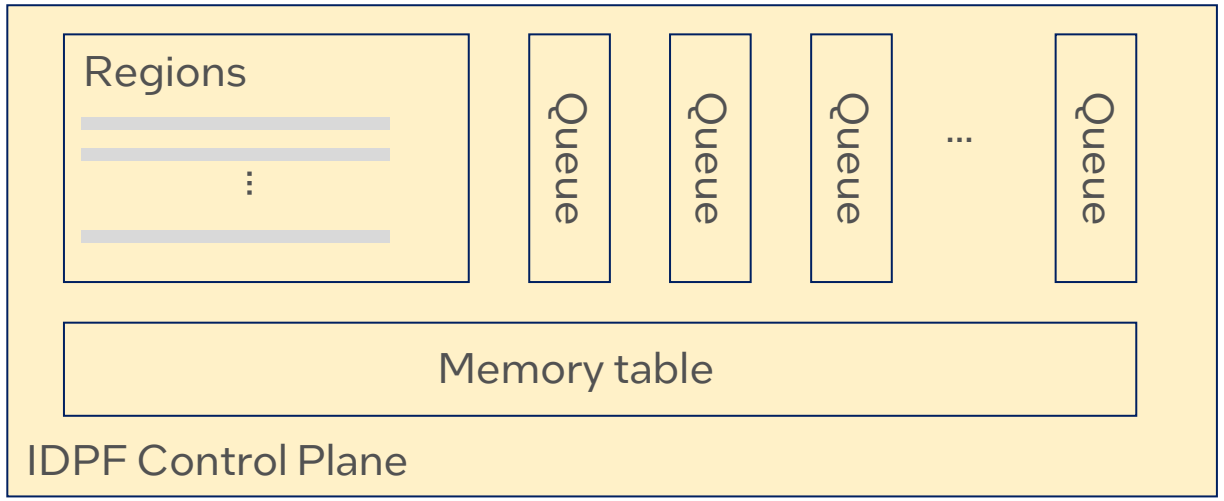
IDPF backend

Vfio-user Protocol





IDPF control plane: Objects




```
struct idpf_emudev {
  struct rte_emudev *edev;
  struct idpf_emu_vfio_user *vfio;
  struct rte_idpf_emu_notify_ops *ops;
  struct rte_idpf_emu_mem *mem;
  struct idpf_emu_intr *intr;
  struct idpf_emu_adminQ adq[RTE_IDPF_EMU_ADMINQ_NUM];
  struct idpf_emu_lanQ *lanq;
  ...
}
struct rte_emudev {
  ...
  uint16_t dev_id;
  struct emu_dev_info dev_info;
  const struct emu_dev_ops *dev_ops;
  void *priv_data;
  ...
} __rte_cache_aligned;
```

IDPF control plane objects

- Regions: device layout
- Queues: address/size + doorbell + interrupt
- Memory table: DMA mapping table

IDPF control plane: Ops

```
struct rte_emudev_ops emu_idpf_ops{
int idpf_emu_dev_start(...);
void idpf_emu_dev_stop(...);
int idpf_emu_dev_configure(...);
int idpf_emu_dev_close(...);
int idpf_emu_subs_ev(...);
int idpf_emu_unsubs_ev(...);
int idpf_emu_get_attr(...);
int idpf_emu_get_queue_info(...);
int idpf_emu_get_irq_info(...);
int idpf_emu_get_db_info(...);
int idpf_emu_get_mem_table(...);
};
```

Lifecycle

Notify

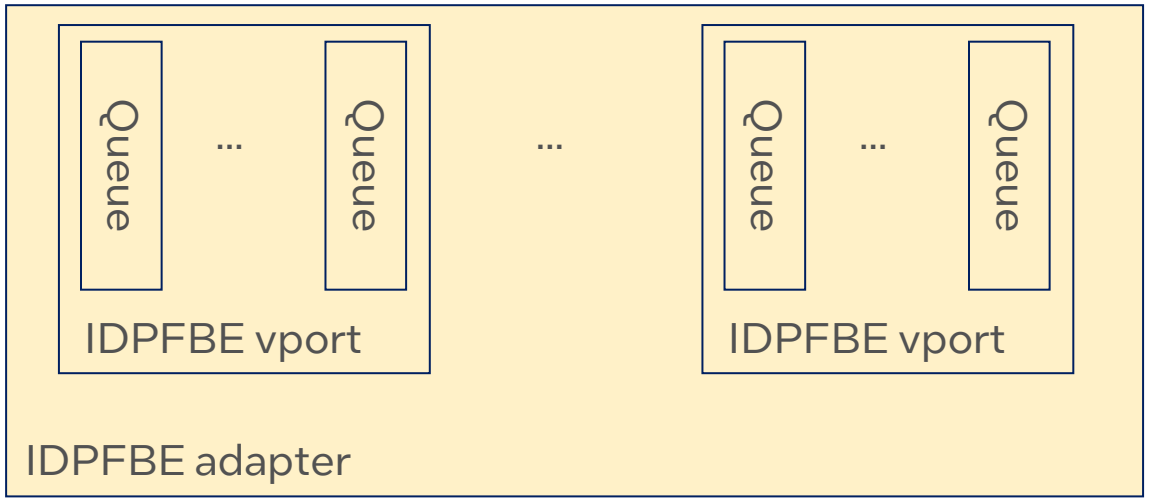
Region

Queue

DMA

- For application
  - Lifecycle management
  - Register notify callback
- For data plane
  - Register notify callback
  - Region read/write
  - Queue and queue notify scheme setup
  - DMA table setup

IDPF data plane: Objects




```
struct idpfbe_adapter {
  struct rte_emudev *emu_dev;
  uint16_t edev_id;
  struct rte_emudev_info dev_info;
  struct rte_idpf_emu_mem *mem_table;
  struct idpfbe_controlq_info cq_info;
  struct virtchnl2_version_info virtchnl_version;
  struct idpfbe_vport **vports;
  ...
};
```

IDPF data plane objects

- Adapter: global information
- Vport: ethdev
- Queues:
  - Single queue mode: rx queue + tx queue
  - Split queue mode: rx queue + tx queue + rx buffer queue + tx completed queue

IDPF data plane: Ops

For application

```
struct eth_dev_ops idpfbe_eth_dev_ops {
    int idpfbe_dev_start(...);
    int idpfbe_dev_stop(...);
    int idpfbe_dev_configure(...);
    int idpfbe_dev_close(...);                    Lifecycle

    int idpfbe_dev_rx_queue_setup(...);
    int idpfbe_dev_tx_queue_setup(...);
    void idpfbe_dev_rx_queue_release(...);
    void idpfbe_dev_tx_queue_release(...);
    void idpfbe_dev_rxq_info_get (...);
    void idpfbe_dev_txq_info_get (...);           Queue

    int idpfbe_dev_link_update (...);             Link
};
```

For IDPF control plane

```
struct rte_idpf_emu_notify_ops idpfbe_notify_ops{
    int idpfbe_new_device(...);
    void idpfbe_destroy_device(...);
    int idpfbe_update_device(...);
    int idpfbe_reset_device(...);                 Device

    int idpfbe_lock_dp (...);                     Lock
};
```

intel®