Hewlett Packard Enterprise

DPDK

Data Plane Development Kit

Ariel Waizel ariel.waizel@gmail.com

Who Am I?

- Programmer for 12 years
  - Mostly C/C++ for cyber and networks applications.
  - Some higher languages for machine learning and generic DevOps (Python and Java).
- M.sc in Information System Engineering (a.k.a machine learning).
- Currently I’m a solution architect at Contextream (HPE)
  - The company specializes in software defined networks for carrier grade companies (Like Bezek, Partner, etc.).
  - Appeal: Everything is open-source!
  - I work with costumers on end-to-end solutions and create POCs.
- Gaming for fun.

Let’s Talk About DPDK

- What is DPDK?
- Why use it?
- How good is it?
- How does it work?

The Challenge

- Network application, done in software, supporting native network speeds.
- Network speeds:
  – 10 Gb per NIC (Network Interface Card).
  – 40 Gb per NIC.
  – 100 Gb?
- Network applications:
  – Network device implemented in software (Router, switch, http proxy, etc.).
  – Network services for virtual machines (Openstack, etc.).
  – Security. protect your applications from DDOS.

What is DPDK?

- Set of UIO (User IO) drivers and user space libraries.
- Goal: forward network packet to/from NIC (Network Interface Card) from/to user application **at native speed**:
  - 10 or 40 Gb NICs.
  - Speed is the most (only?) important criteria.
  - Only forwards the packets – not a network stack (But there're helpful libraries and examples to use).
- All traffic bypasses the kernel (We'll get to why).
  - When a NIC is controlled by a DPDK driver, it's invisible to the kernel.
- Open source (BSD-3 for most, GPL for Linux Kernel related parts).

Why DPDK Should Interest Kernel Devs?

- Bypassing the kernel is importance because of **performance -** Intriguing by itself.
  - At the very least, know the “competition”.
- DPDK is a very light-weight, low level, performance driven framework – Makes it a good learning ground for kernel developers, to learn performance guidelines.

Why Use DPDK?

Why bypassing the kernel is a necessity.

10Gb – Crunching Numbers

- **Minimum Ethernet packet**: 64 bytes + 20 preamble.
- **Maximum number of pps**: 14,880,952
  - 10^10 bps /(84 bytes * 8)
- **Minimum time to process a single packet**: 67.2 ns
- **Minimum CPU cycles**: 201 cycles on a 3 Ghz CPU (1 Ghz -> 1 cycle per ns).

Time (ish) per Operation (July 2014)

Operation	Time (Expected)
Cache Miss	32 ns
L2 cache access	4.3 ns
L3 cache access	7.9 ns
“LOCK” operation (like Atomic)	8.25 ns (16.5 ns for unlock too)
syscall	41.85 ns (75.34 for audit-syscall)
SLUB Allocator (Linux Kernel buffer allocator)	80 ns for alloc + free
SKB (Linux Kernel packet struct) Memory Manager	77 ns
Qdisc (tx queue descriptor)	48 ns minimum, between 58 to 68 ns average
TLB Miss	Up to several Cache Misses.

Conclusion: Must use batching (Send/rcv several packet together at the same time, amortize costs)

* Red Hat Challenge 10Gbit/s Jesper Dangaard Brouer et al. Netfilter Workshop July 2014.

Time Per Operation – The Big No-Nos

Operation	Average Time
Context Switch	Micro seconds (1000 ns)
Page Fault	Micro seconds

Conclusion: Must pin CPUs and pre-allocate resources

As of Today: DPDK can handle 11X the traffic Linux Kernel Can!

Benchmarks at the end.

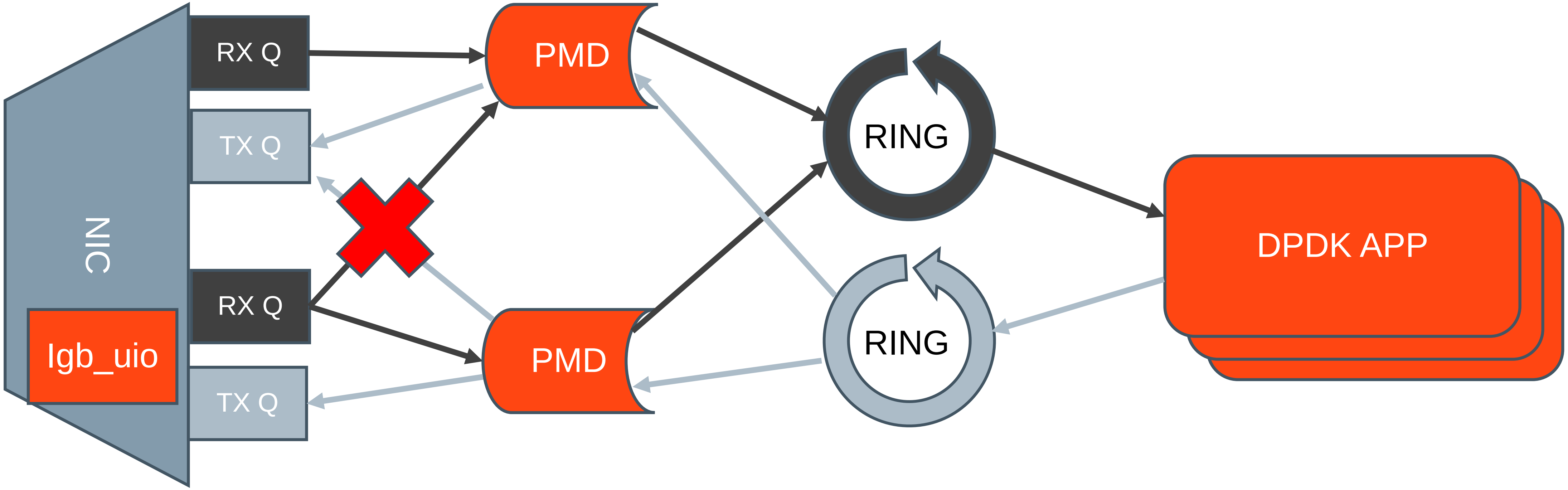
DPDK Architecture

How It Works.

DPDK – What do you get?

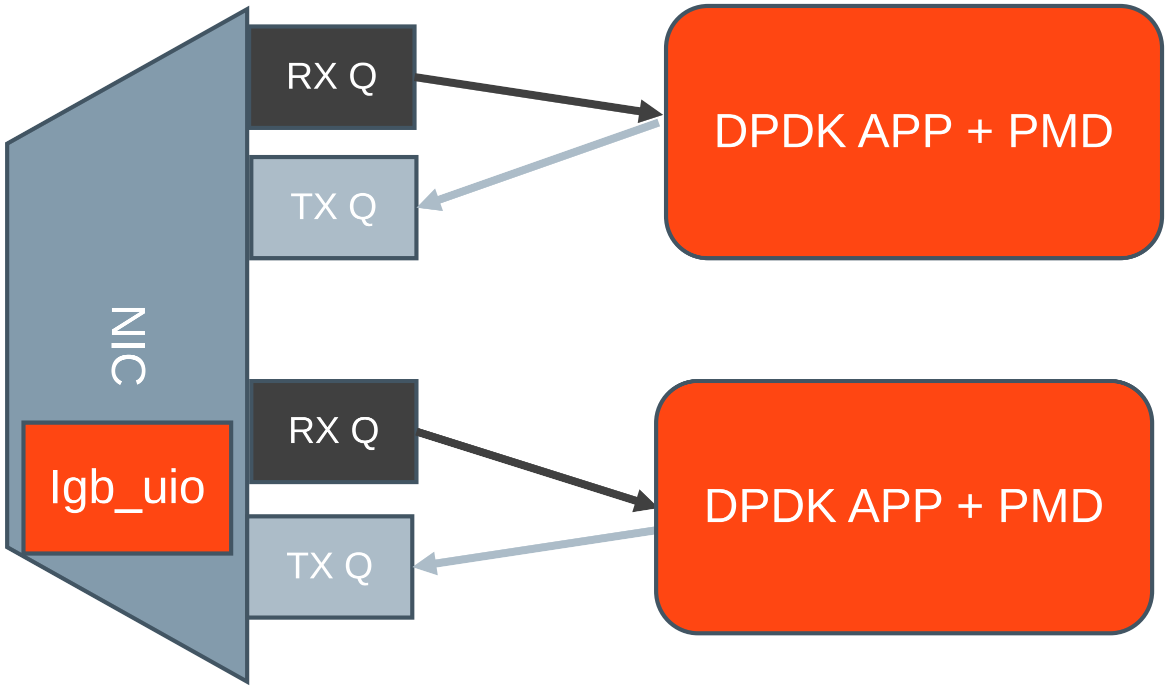
- UIO drivers
- PMD per hardware NIC:
  - PMD (Poll Mode Driver) support for RX/TX (Receive and Transmit).
  - Mapping PCI memory and registers.
  - Mapping user memory (for example: packet memory buffers) into the NIC.
  - Configuration of specific HW accelerations in the NIC.
- User space libraries:
  - Initialize PMDs
  - Threading (builds on pthread).
  - CPU Management
  - Memory Management (Huge pages only!).
  - Hashing, Scheduler, pipeline (packet steering), etc. – High performance support libraries for the application.

From NIC to Process – Pipe Line model



* igb_uio is the DPDK standard kernel uio driver for device control plane

From NIC to Process – Run To Completion Model



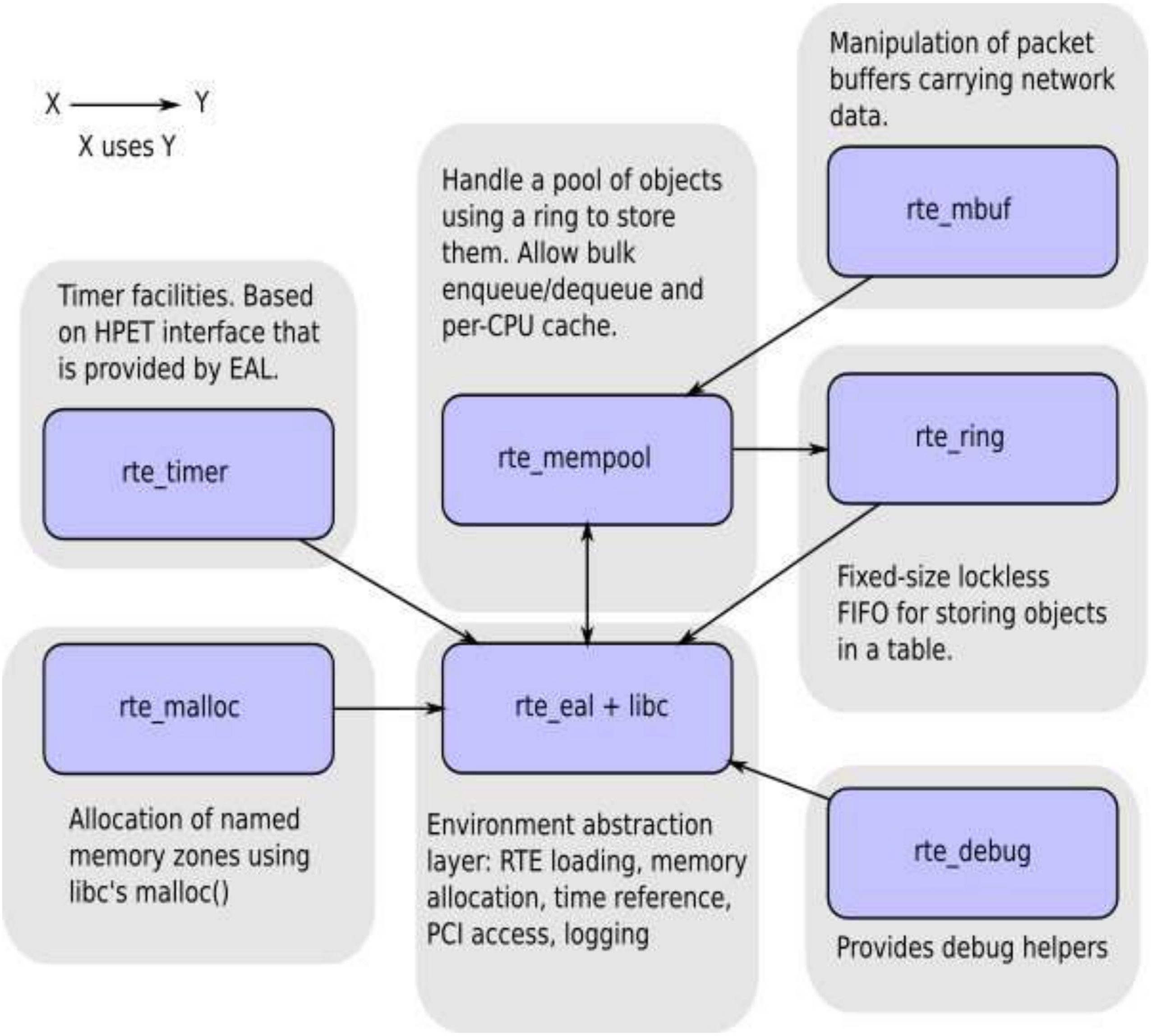
Software Configuration

- C Code
  - GCC 4.5.X or later.
- Required:
  - Kernel version >= 2.6.34
  - Hugetablefs (For best performance use 1G pages, which require GRUB configuration).
- Recommended:
  - isolcpus in GRUB configuration: isolates CPUs from the scheduler.
- Compilation
  - DPDK applications are statically compiled/linked with the DPDK libraries, for best performance.
  - Export RTE_SDK and RTE_TARGET to develop the application from a misc. directory
- Setup:
  - Best to use tools/dpdk-setup.sh to setup/build the environment.
  - Use tools/dpdk-devbind.py
    - -- status let’s you see the available NICs and their corresponding drivers.
    - -bind let’s you bind a NIC to a driver. Igb-uio, for example.
  - Run the application with the appropriate arguments.

Software Architecture

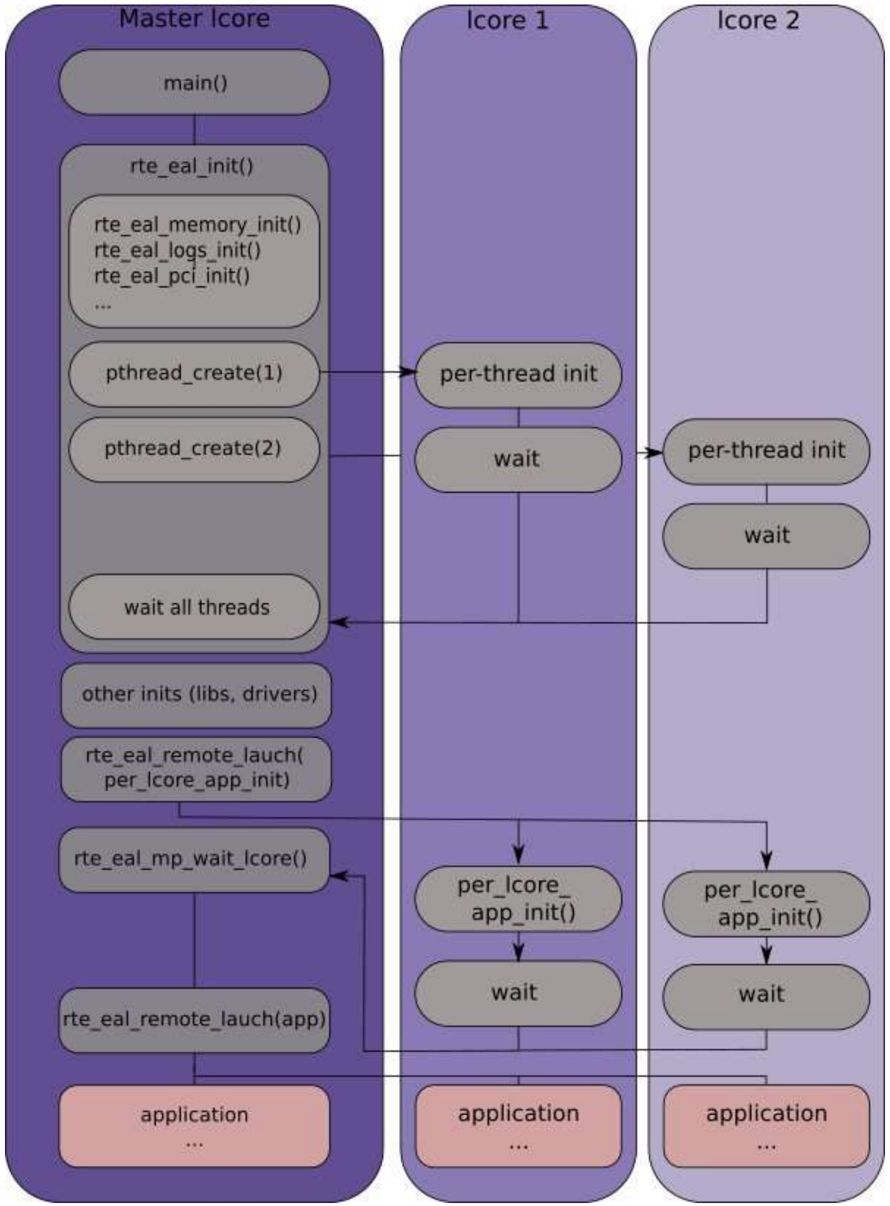
- PMD drivers are just user space pthreads that call specific EAL functions
  - These EAL functions have concrete implementations per NIC, and this costs couple of indirections.
  - Access to RX/TX descriptors is direct.
  - Uses UIO driver for specific control changes (like configuring interrupts).
- Most DPDK libraries are not thread-safe.
  - PMD drivers are non-preemptive: Can’t have 2 PMDs handle the same HW queue on the same NIC.
- All inner-thread communication is based on librte_ring.
  - A mp-mc lockless non-resizable queue ring implementation.
  - Optimized for DPDK purposes.
- All resources like memory (malloc), threads, descriptor queues, etc. are initialized at the start.

Software Architecture



Hewlett Packard Enterprise

Application Bring up



Hewlett Packard
Enterprise

Code Example – Basic Forwarding

```
int
main(int argc, char *argv[])
{
        struct rte_mempool *mbuf_pool;
        unsigned nb_ports;
        uint8_t portid;

        /* Initialize the Environment Abstraction Layer (EAL). */
        int ret = rte_eal_init(argc, argv);
        if (ret < 0)
                rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");

        argc -= ret;
        argv += ret;

        /* Check that there is an even number of ports to send/receive on. */
        nb_ports = rte_eth_dev_count();
        if (nb_ports < 2 || (nb_ports & 1))
                rte_exit(EXIT_FAILURE, "Error: number of ports must be even\n");

        /* Creates a new mempool in memory to hold the mbufs. */
        mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
                MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());

        if (mbuf_pool == NULL)
                rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n");

        /* Initialize all ports. */
        for (portid = 0; portid < nb_ports; portid++)
                if (port_init(portid, mbuf_pool) != 0)
                        rte_exit(EXIT_FAILURE, "Cannot init port %"PRIu8 "\n",
                                        portid);

        if (rte_lcore_count() > 1)
                printf("\nWARNING: Too many lcores enabled. Only 1 used.\n");

        /* Call lcore_main on the master core only. */
        lcore_main();

        return 0;
}
```

DPDK Init

Get all available NICS (binded with igb_uio)

Initialize packet buffers

Initialize NICs.

Code Example – port_init

```c
static inline int
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
{
        struct rte_eth_conf port_conf = port_conf_default;
        const uint16_t rx_rings = 1, tx_rings = 1;
        int retval;
        uint16_t q;

        if (port >= rte_eth_dev_count())
                return -1;

        /* Configure the Ethernet device. */
        retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
        if (retval != 0)
                return retval;

        /* Allocate and set up 1 RX queue per Ethernet port. */
        for (q = 0; q < rx_rings; q++) {
                retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
                if (retval < 0)
                        return retval;
        }

        /* Allocate and set up 1 TX queue per Ethernet port. */
        for (q = 0; q < tx_rings; q++) {
                retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                                rte_eth_dev_socket_id(port), NULL);
                if (retval < 0)
                        return retval;
        }

        /* Start the Ethernet port. */
        retval = rte_eth_dev_start(port);
        if (retval < 0)
                return retval;

        /* Display the port MAC address. */
        struct ether_addr addr;
        rte_eth_macaddr_get(port, &addr);
        printf("Port %u MAC: %02" PRIx8 " %02" PRIx8 " %02" PRIx8
                   " %02" PRIx8 " %02" PRIx8 " %02" PRIx8 "\n",
                        (unsigned)port,
                        addr.addr_bytes[0], addr.addr_bytes[1],
                        addr.addr_bytes[2], addr.addr_bytes[3],
                        addr.addr_bytes[4], addr.addr_bytes[5]);

        /* Enable RX in promiscuous mode for the Ethernet device. */
        rte_eth_promiscuous_enable(port);

        return 0;
}
```

NIC init: Set number of queues

Rx queue init: Set packet buffer pool for queue
* Uses librte_ring to be thread safe

Tx queue init: No need for buffer pool

Start Getting Packets

Code Example – lcore_main

```
static __attribute__((noreturn)) void
lcore_main(void)
{
        const uint8_t nb_ports = rte_eth_dev_count();
        uint8_t port;

        /*
         * Check that the port is on the same NUMA node as the polling thread
         * for best performance.
         */
        for (port = 0; port < nb_ports; port++)
                if (rte_eth_dev_socket_id(port) > 0 &&
                                rte_eth_dev_socket_id(port) !=
                                                (int)rte_socket_id())
                        printf("WARNING, port %u is on remote NUMA node to "
                                        "polling thread.\n\tPerformance will "
                                        "not be optimal.\n", port);

        printf("\nCore %u forwarding packets. [Ctrl+C to quit]\n",
                        rte_lcore_id());

        /* Run until the application is quit or killed. */
        for (;;) {
                /*
                 * Receive packets on a port and forward them on the paired
                 * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
                 */
                for (port = 0; port < nb_ports; port++) {

                        /* Get burst of RX packets, from first port of pair. */
                        struct rte_mbuf *bufs[BURST_SIZE];
                        const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                                        bufs, BURST_SIZE);

                        if (unlikely(nb_rx == 0))
                                continue;

                        /* Send burst of TX packets, to second port of pair. */
                        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                                        bufs, nb_rx);

                        /* Free any unsent packets. */
                        if (unlikely(nb_tx < nb_rx)) {
                                uint16_t buf;
                                for (buf = nb_tx; buf < nb_rx; buf++)
                                        rte_pktmbuf_free(bufs[buf]);
                        }
                }
        }
}
```

PMD

Igb_uio

- For Intel Gigabit NICs.
  - Simple enough to work for most NICs, nonetheless.
- Basically:
  - Calls pci_enable_device.
  - Enables bus mastering on the device (pci_set_master).
  - Requests all BARs and mas them using ioremap.
  - Setups ioports.
  - Sets the dma mask to 64-bit.
- Code to support SRIOV and xen.

rte_ring

- Fixed sized, “lockless”, queue ring
- Non Preemptive.
- Supports multiple/single producer/consumer, and bulk actions.
- Uses:
  - Single array of pointers.
  - Head/tail pointers for both producer and consumer (total 4 pointers).
- To enqueue (Just like dequeue):
  - Until successful:
    - Save in local variable the current head_ptr.
    - head_next = head_ptr + num_objects
    - CAS the head_ptr to head_next
  - Insert objects.
  - Until successful:
    - Update tail_ptr = head_next + 1 when tail_ptr == head_ptr
- Analysis:
  - Light weight.
  - In theory, both loops are costly.
  - In practice, as all threads are cpu bound, the amortized cost is low for the first loop, and very unlikely at the second loop.

rte_mempool

- Smart memory allocation
- Allocate the start of each memory buffer at a different memory channel/rank:
  - Most applications only want to see the first 64 bytes of the packet (Ether+ip header).
  - Requests for memory at different channels, same rank, are done concurrently by the memory controller.
  - Requests for memory at different ranks can be managed effectively by the memory controller.
  - Pads objects until gcd(obj_size,num_ranks*num_channels) == 1.
- Maintain a per-core cache, bulk requests to the mempool ring.
- Allocate memory based on:
  - NUMA
  - Contiguous virtual memory (Means also contiguous physical memory for huge pages).
- Non- preemptive.
  - Same lcore must not context switch to another task using the same mempool.

Benchmarks

Linux kernel Vs. DPDK

Linux Kernel Benchmarks, single core

Benchmark	July 2014	Feb. 2016
TX	4 Mpps	14.8 Mpps
RX (Dump at driver)	6.4 Mpps	12 Mpps (experimental)
L3 Forwarding (RX+Filter+TX)	1 Mpps	2 Mpps
L3 forwarding (Multi Core)	6 Mpps	12 Mpps

* Red Hat The 100Gbit/s Challenge Jesper Dangaard Brouer et al. DevConf Feb. 2016.

DPDK Benchmarks (March 2016)

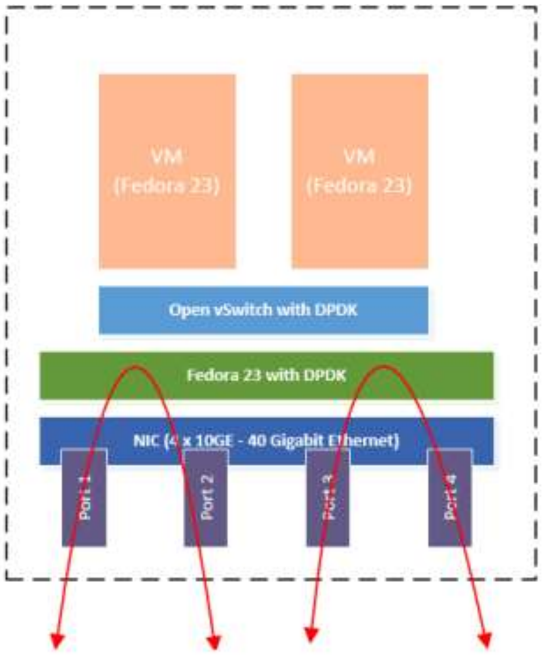
Benchmark	Single Core	Multi Core
L3 forwarding (PHY-PHY)	22 Mpps	Linear Increase
Switch Forwarding (PHY-OVS-PHY)	11 Mpps	Linear Increase
VM Forwarding (PHY-VM-PHY)	3.4 Mpps	Near Linear Increase
VM to VM	2 Mpps	Linear Increase

- All tests with:
  - 4X40Gb ports
  - E5-2695 V4 2.1Ghz Processor
  - 16X1GB Huge Pages, 2048X2MB Huge Pages

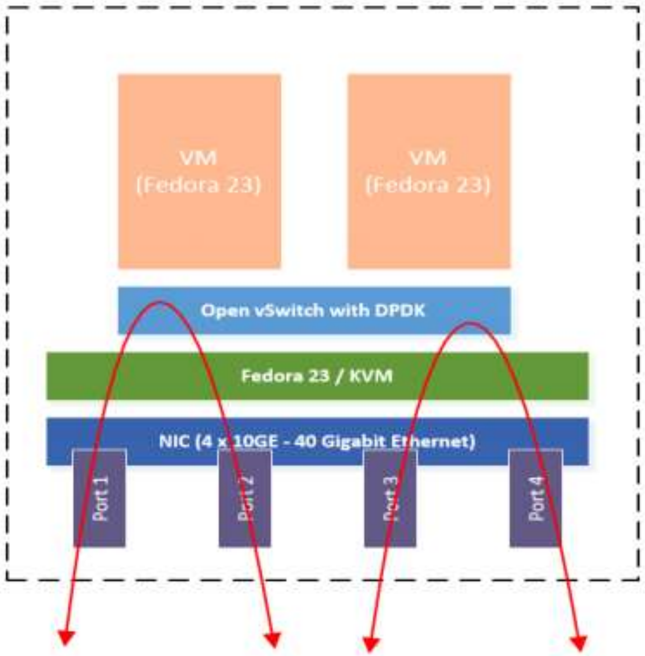
* Intel Open Network Platform Release 2.1 Performance Test Report March 2016.

DPDK Benchmarks Figures

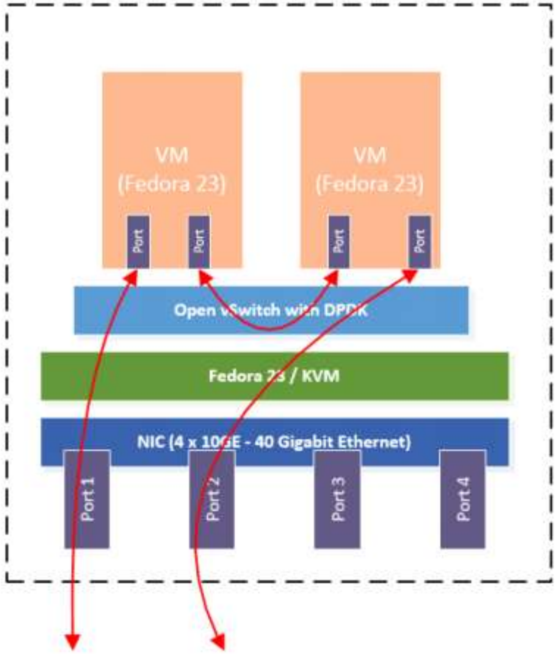
PHY-PHY



PHY-OVS-PHY



VM-VM



DPDK Pros and Cons

DPDK Advantages

- Best forwarding performance solution to/from PHY/process/VM to date.
  - Best single core performance.
  - Scales: linear performance increase per core.
- Active and longstanding community (from 2012).
  - DPDK.org
  - Full of tutorials, examples and complementary features.
- Active popular products
  - OVS-DPDK is the main solution for high speed networking in openstack.
  - 6WIND.
  - TRex.
- Great virtualization support.
  - Deploy at the host of the guest environment.

DPDK Disadvantages

- Security
- Isolated ecosystem:
  – Hard to use linux kernel infrastructure (While there are precedents).
- Requires modified code in applications to use:
  – DPDK processes use a very specific API.
  – DPDK application can’t interact transparently with Linux processes (important for transparent networking applications like Firewall , DDOS mitigation, etc.).
    - Solved for interaction with VMs by the vhost Library.
- Requires Huge pages (XDP doesn’t).