École Polytechnique Fédérale de Lausanne

Semester Project

Porting a driver for the Intel XL710 40GbE NIC to the IX Dataplane Operating System

---

Student:
Andy Roulin (216690)

Direct Supervisor:
George Prekas
Academic Supervisor:
Edouard Bugnion

January 2017

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Abstract

This report presents the port of an i40e poll mode driver for the Intel XL710 40GbE network card to the IX dataplane operating system.

Introduction

The goal of the project is to add support for the Intel XL710 40GbE Network Interface Card (NIC) to IX. IX[1] is a dataplane operating system which provides high-throughput and low latency networking for event-driven applications. The driver port leverages the i40e DPDK driver for the card's initialization and then runs on its own as a poll mode driver, accessing the NIC's hardware directly.

This report is organized as follows: section 1 introduces the project by giving an overview of IX, NIC's support in IX and DPDK. Section 2 introduces the Intel XL710 card and its advanced hardware features. Section 3 describes refactoring work done at the beginning of the project to support multiple drivers in IX. The actual i40e driver implementation and related issues are then explained in section 4. Section 5 provides performance results from benchmarks while section 6 describes further work and section 7 concludes.

1 Introduction

This section provides an overview of IX and the different components that are involved in the transmit and receive paths. It first gives an overview of IX and then explains how NICs are supported in IX through DPDK. DPDK itself and its relationship with IX are summarized as well at the end of the section.

1.1 The IX Dataplane Operating System

IX[1] is a protected dataplane operating system for high throughput and low latency network communications. It provides (1) isolation and direct access to hardware through virtualization, (2) fast L2 packet processing through DPDK and (3) a TCP/IP stack making IX a full-stack dataplane OS. IX runs in the same mode as guest OSes, i.e., non-hypervisor ring 0, with applications running in ring 3. Linux is used as the hypervisor and control plane which enables IX to support Linux system calls and services such as file systems and signals. The general IX architecture is shown on Figure 1. The Dune [5] project (consisting of a kernel module and libraries) enables Linux to run as hypervisor and dataplane instances (IX OS and one dataplane application) to run as regular Linux processes.

1.2 NIC Support in IX

As IX interacts with the hardware directly (through virtualization), it must come with its own set of drivers for the different network cards supported. At the beginning of this project, IX only supported the ixgbe NIC family (10GbE Intel cards). This project aims to add support for the Intel 40GbE family of cards focusing first on getting XL710 (one card from that family) to work.

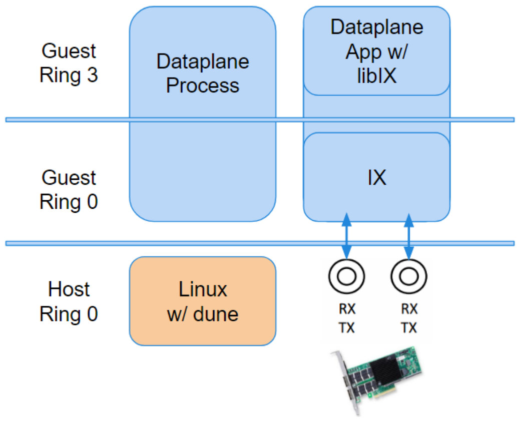

Figure 1: IX OS general architecture

IX uses DPDK[3] (see 1.3 for an overview of DPDK) for NIC's initialization and general configuration. DPDK comes with its own set of drivers (including i40e support) and IX uses them for the initialization and configuration parts of the devices. IX calls the main DPDK initialization function which will initialize the NIC to a working initial state. If IX later needs to change at run-time some card's configuration (e.g., RSS or promiscuous mode), it will call DPDK's functions as well. This way, IX can focus on optimizing the critical paths, i.e., transmit and receive paths, without having to rewrite all the code for the cards' configuration specifics. The three main tasks of an IX driver are thus to:

- Interact with the upper layer in IX to deliver and transmit packets
- Interact with DPDK to initialize the card and change its configuration at run-time
- Provide its own (DPDK-free) transmit and receive path. IX manages the TX and RX descriptor rings (and thus receive and transmit packets) without involving DPDK.

1.3 DPDK

DPDK[3] is an open-source set of layer 2 libraries and drivers for fast user-space packet processing. It has support for a large number of NICs which is one reason why IX uses DPDK for initialization and configuration.

DPDK's main goal[4] is to receive and send packets in a minimum number of cycles (order of 80 CPU cycles) by providing user-space transmit and receive paths that bypass the kernel. Going through the kernel adds overhead in the form of system calls, context switches, data copying and interrupt handling. DPDK provides a user-space alternative path that DMAs packet directly from/to user-space memory (zero-copy), uses polling instead of interrupts, hugepages to prevent TLB misses, cache-aligned structures and other similar optimizations.

1.4 IX and DPDK Complex Relationship

The current relationship between IX and DPDK is complex as IX is both linking against DPDK driver libraries and borrowing (as well as adapting) DPDK's code for e.g., packet processing and queue management. IX also provides its own custom memory management instead of using DPDK's memory management. This relationship mostly results from early implementation decisions but abstracting these implementations details, IX could have been implemented as a DPDK application.

2 The Intel XL710 40GbE NIC

This section describes the card used in this project and its capabilities. It starts with a general overview of it and then describes both the important hardware features supported in this project and the features left for future work.

2.1 General Overview

The Intel XL710 NIC is a 40GbE NIC, i.e., it supports a MAC+PHY bandwidth up to 40 Gb/s. This is done through a QSFP+ physical interface consisting of 4x10Gb channels. This card can actually be used as 4x10Gb interfaces or 1x40Gb interface. An additional mode providing 2x40Gb is also available but the combined throughput of the two interfaces cannot exceed 40 Gbps.

The NIC is connected to the rest of the computer system through PCIe Gen3 on 8 lanes. Each lane provides around 8 Gb/s giving the PCIe link a total bandwidth of around 63.04 Gbits/sec Gbps (which is enough to handle traffic from/to the 40 Gb/s physical channel).

2.2 Main Hardware Features Supported

- **Multiple queues**: the XL710 NIC provides up to 1536 TX/RX queue pairs. Each queue is processed by an individual CPU. These queues can be categorized into different priority sets. Within a set, egress packets are selected from queues using a round-robin scheduler (in hardware). IX only uses one TX/RX queue pair per CPU.
- **Receiver-side Scaling (RSS)**: RSS directs ingress packets to the right RX queue and thus directly to the final destination CPU using hashing. Incoming packets fields (fields from IP and TCP headers such as IP addresses and ports) are hashed to get an entry in a redirection table (see figure 2) containing queue numbers (each queue is managed by one CPU) The XL710 has a redirection table containing 512 entries (called flow groups) while the previous 10GbE cards supported in IX had 128 entries.
- **Checksum Offloading**: Checksum computations can be done in hardware, by enabling flags in ring descriptors. Checksum offloading is supported for IP, TCP and UDP.

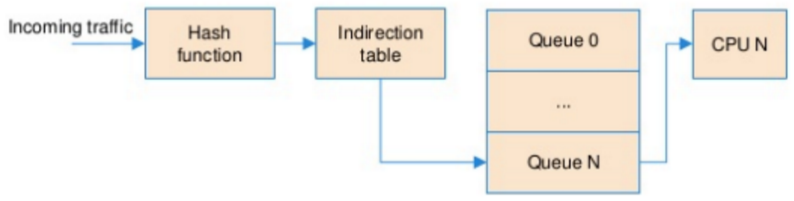

Figure 2: Receiver-side scaling

2.3 Main Hardware Features for Future Work

- **TCP Segmentation Offloading (TSO)**: TCP packets can be much bigger than the Minimum Transmission Unit (MTU) supported by a NIC. TSO offloads the work of breaking packets into several smaller chunks that are transmissible individually by the NIC.
- **Single-root Input/Output Virtualization (SR-IOV)**: SR-IOV, shown in Figure 3 refers to the support in hardware to multiplex physical resources among guests VMs. The XL710 can virtualize physical PCI functions into virtual functions so that one physical NIC can present itself as multiple virtual NICs to guests operating systems.

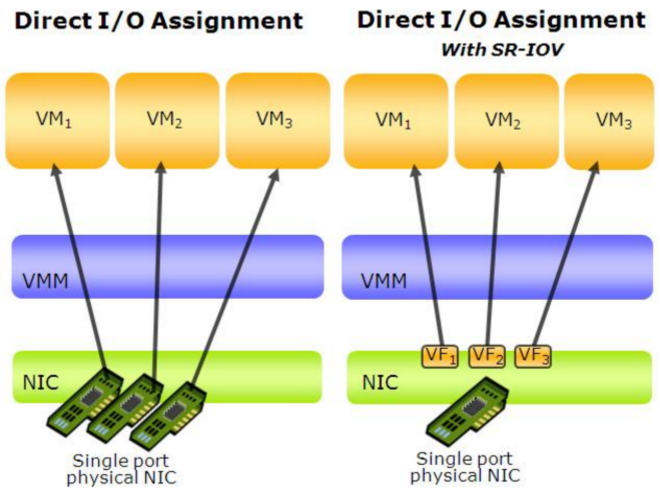

Figure 3: Single-root input/output virtualization

3 Refactoring Needed to Make IX Driver-independent

At the beginning of the project, IX supported only the ixgbe 10GbE family of NICs. The `dp/drivers` directory only contained the ixgbe driver (`ixgbe.c` file) but more importantly some other parts of the code would directly reference ixgbe specific functions. For example, `dp/core/init.c`, which contains IX core initialization code, would directly call `ixgbe_init`.

Multiple DPDK libraries are statically linked to IX and provide access to DPDK's Ethernet device abstraction. This library enables IX to use driver-independent DPDK device initialization and configuration functions. Thus initializing a device or configure a general parameter present on all or most devices (such as promiscuous mode) comes down to calling DPDK driver-independent functions.

The first part of the project was thus to make IX core code driver-independent in order to introduce the new i40 driver. This section explains the changes needed starting first with refactoring the driver initialization. Secondly, direct references to ixgbe functions must be changed to driver-independent ones. Finally, in the refactoring process, common code that can be shared between drivers was moved to the new file `dp/drivers/common.c`.

3.1 IX Driver Initialization

Listing 1 shows a new table built to reference the different drivers available in IX. There are two ixgbe drivers available, with and without virtual functions and one driver for the i40e card (virtual function for the XL710 is future work).

```
struct drv_init_tble {
    char *name;
    int (*init_fn)(struct ix_rte_eth_dev *dev, const char *driver_name);
};

struct drv_init_tble drv_init_tbl[] = {
    { "rte_ixgbe_pmd", ixgbe_init },
    { "rte_ixgbevf_pmd", ixgbe_init },
    { "rte_i40e_pmd", i40e_init },
    { NULL, NULL }
};
```

Listing 1: Driver's initialization functions

The common driver initialization function `driver_init` in `dp/drivers/common.c`, shown partially in Listing 2, performs generic driver initialization (which includes common DPDK driver initialization), sets the MAC address of the device and calls the corresponding `init` function from the table in listing 1.

```
int driver_init(struct pci_dev *pci_dev, struct ix_rte_eth_dev **ethp)
{
    // [...] Common DPDK initialization

    // Find DPDK driver that matches the NIC PCI address
    ret = dpdk_devinit(pci_dev, &driver);

    [...]

    // Call the initialization function of IX driver
    for (dpdk_drv = drv_init_tbl; dpdk_drv->name != NULL; dpdk_drv++) {
        if (strcmp(driver->name, dpdk_drv->name))
            continue;
        ret = dpdk_drv->init_fn(dev, dpdk_drv->name);
        if (ret < 0)
            return ret;
        break;
    }

    if (dpdk_drv->name == NULL)
        panic("No suitable DPDK driver found\n");
}
```

Listing 2: Generic driver initialization

3.2 Direct References to ixgbe Functions

A few places in IX core code directly referenced ixgbe data structures or functions. Listing 3 shows how the IX function `eth_rx_idle_wait`, which waits until at least a packet has been received, directly checks an ixgbe ring descriptor field while it should actually be driver-independent. It was replaced with a call to a driver operation `ready` that each driver implements.

```
// dp/core/ethqueue.c: eth_rx_idle_wait(usecs)
if (rxq->ring[idx].wb.status &  cpu_to_le32(IXGBE_RXDADV_DD))
    return true
```

Listing 3: Checking the first ring descriptor for a received packet

3.3 Shared Driver Functions

Examples of such functions are functions that only call the generic DPDK Ethernet layer. Listing 4 shows one such function that was previously in `dp/drivers/ixgbe.c` but is now in `dp/drivers/common.c` and thus is shared between the ixgbe and i40e drivers.

```
void generic_promiscuous_disable(struct ix_rte_eth_dev *dev)
{
    rte_eth_promiscuous_disable(dev->port);
}
```

Listing 4: Shared driver function leveraging DPDK generic Ethernet layer

These functions are referenced through the device operation tables. The device operation table for the i40e driver is shown in Listing 5. Function names that now starts with `generic_` are generic driver functions located in `dp/drivers/common.c`. However, an IX driver can (and in some cases must, if the function is driver-specific) implement its own version and substitute it in the device operation table (see for example the `dev_start` function which has to initialize NIC-specific queue parameters).

```
static struct ix_eth_dev_ops eth_dev_ops = {
    .allmulticast_enable = generic_allmulticast_enable,
    .dev_infos_get = generic_dev_infos_get,
    .dev_start = dev_start,
    .link_update = generic_link_update,
    .promiscuous_disable = generic_promiscuous_disable,
    .reta_update = reta_update,
    .rx_queue_setup = rx_queue_setup,
    .tx_queue_setup = tx_queue_setup,
    .fdir_add_perfect_filter = generic_fdir_add_perfect_filter,
    .fdir_remove_perfect_filter = generic_fdir_remove_perfect_filter,
    .rss_hash_conf_get = generic_rss_hash_conf_get,
    .mac_addr_add = generic_mac_addr_add,
};
```

Listing 5: IX device operation table for the i40 driver

4 Implementation

The section describes the actual i40e driver's implementation in IX. The goals of this driver are to interact with the rest of IX and with the hardware. As DPDK is used for NIC's initialization and configuration, the driver focuses on reimplementing the transmit and receive paths. IX drivers let DPDK create and initialize TX and RX queues which then IX drivers manage by themselves.

To ease the work of working directly at the hardware level, the driver uses functions and definitions from the i40e driver. This DPDK driver contains definitions for hardware structures, constants and initialization/configuration functions. Data structures include, for example, ring descriptors and queue descriptors. Queue initialization or RSS tables updates are examples of functions used by IX from DPDK i40e driver library.

4.1 Queue Initialization

The first task is to create TX and RX queues, i.e., descriptor rings. A queue has a number of configurable parameters such as number of descriptors, maximum packet size or descriptor types. Its state is defined by three pointers: the `BASE` pointer who points at the start of the queue in main memory and the `HEAD` and `TAIL` pointers which delimit the descriptors that are to be used by the NIC (descriptors in which packets have to be sent or descriptors in which to place received packets) as shown in Figure 4. The driver setups descriptors and then advances the TAIL pointer to transmit packets or give available descriptors to receive packets to the NIC. The NIC itself also maintains its own version of the three pointers in registers. The `TAIL` pointer is updated by the driver on the NIC through a PCI write when transmitting packets or giving new descriptors for reception. The `HEAD` pointer is updated automatically by the NIC.

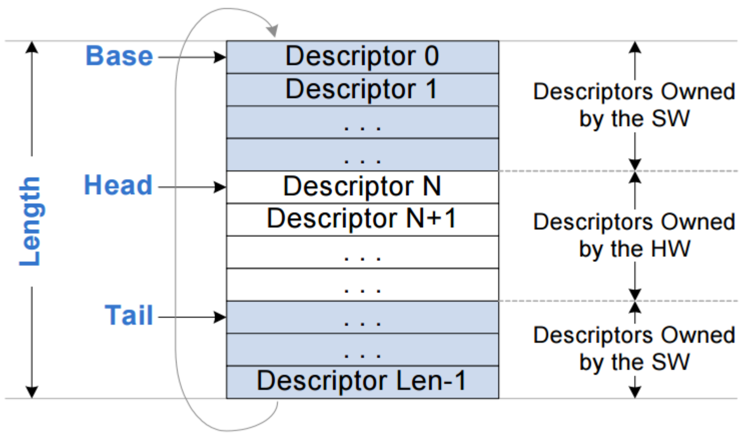

Figure 4: Receive descriptor ring (receive queue)[6]

In order to create these queues, the driver calls the queue setup function from DPDK's Ethernet device abstraction library to create and initialize queues to a working state with hardware configured. IX then allocates main memory (in 2MB pages to reduce TLB misses) and changes the `BASE` pointers of the queues to point to this new area. In other words, queues are created through DPDK but actually the descriptor rings themselves reside in IX's memory.

Some queue parameters are only initialized by DPDK when the starting the queue (call to `rte_eth_dev_start`). Examples include the `TAIL` register address or the maximum size of packets. IX cannot retrieve or initialize them before starting the queue on DPDK's side. Thus IX's own `dev_start` contains first a call to `rte_eth_dev_start` and then retrieves/modifies what is needed.

4.2 Transmit Path

After the TX queue is initialized, all descriptors belong to the driver. In order to transmit packets, the driver fills each transmit descriptor, shown in Figure 5, with the memory address of the packet to send (the packet is never copied to another buffer), the packet length and parameters such as enabling checksum offloading. The final part is to update the `TAIL` register on the NIC by the number of descriptors filled, i.e., number of packets to transmitted. This update is done with a PCI write. Listing 6 gives a simplified outline of the transmit path.

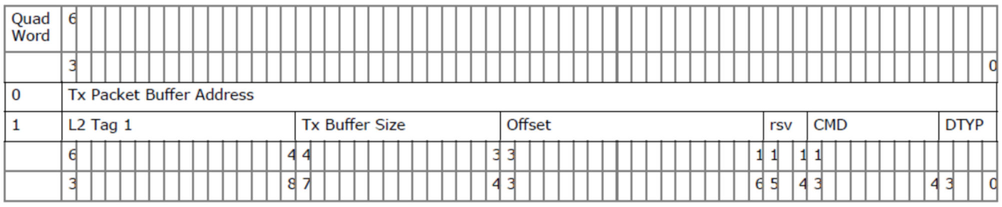

Figure 5: Transmit descriptor format[6]

```
// Write the memory address of the packet (mbuf) to the descriptor
// The i40e card doesn't use the IOMMU for now, so the address must be a machine address
maddr = mbuf_get_data_machaddr(mbuf);
tx_descriptor->buffer_addr = rte_cpu_to_le_64(maddr);

// Construct and write the various flags, .e.g. checksum,
// and length of the packet to the descriptor
txdp->cmd_type_offset_bsz = i40e_build_ctob((uint32_t)td_cmd, td_offset, mbuf->len, 0);

// Update driver's tail pointer
txq->tail++;

// Update the tail pointer on the NIC
// As soon as the PCI write finishes, the NIC can decide to transmit the packet
I40E_PCI_REG_WRITE(txq->tdt_reg_addr, txq->tail & (txq->len - 1));
```

Listing 6: Transmit path

The NIC continuously transmits packets from descriptors it owns, updates its `HEAD` pointer to return this descriptors to the driver and writes back to the descriptors with a special flag telling the packet was transmitted (`DONE` flag). The driver can then look at that flag and update its version of the `HEAD` pointer accordingly (avoiding a PCI read) to reuse descriptors. The process of reusing used descriptors is outlined in Listing 7.

```c
// Reclaiming used TX descriptors
while (txq->head + idx != txq->tail) {
        // Look at the done flag of each descriptor to see if packet has been sent
        struct i40e_tx_desc *txdp = txq->ring[(txq->head + idx) & (txq->len - 1)];
        if ((txdp->cmd_type_offset_bsz &
                rte_cpu_to_le_64(I40E_TXD_QW1_DTYPE_MASK)) !=
                rte_cpu_to_le_64(I40E_TX_DESC_DTYPE_DESC_DONE))
            break;

        idx++;
        nb_desc = idx;
    }

// Update the driver's head pointer
// The NIC automatically updates its version of the pointer
txq->head += nb_desc;
```

Listing 7: Reclaiming descriptors

4.3 Receive Path

Similarly after the RX queue is initialized, all descriptors belong to the driver. The driver fills at initialization time each transmit descriptor, shown in Figure 6, with the memory address of a buffer to receive a packet (the `Header Buffer Address` field is not used. This is done by the `i40e_alloc_rx_mbufs` function, shown in Listing 8 for general understanding.

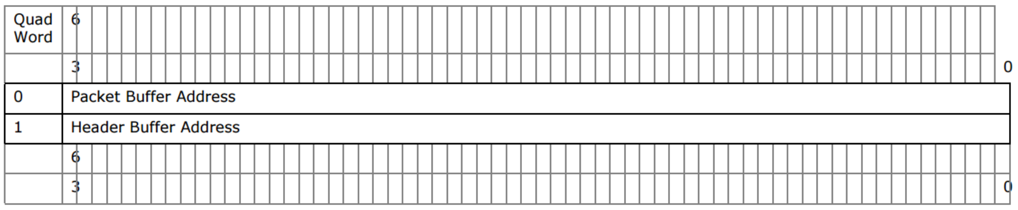

Figure 6: Receive descriptor format

```c
static int i40e_alloc_rx_mbufs(struct rx_queue *rxq)
{
    int i;
    for (i = 0; i < rxq->len; i++) {
        machaddr_t maddr;
        struct mbuf *b = mbuf_alloc_local();
        if (!b)
            return -ENOMEM;

        maddr = mbuf_get_data_machaddr(b);
        rxq->ring[i].read.pkt_addr = rte_cpu_to_le_64(maddr);
    }
}
```

Listing 8: Filling the descriptor ring at initialization time

The NIC places received packets into descriptors it owns, updates its `HEAD` pointer to return these descriptors to the driver and writes back to them with a special flag telling a packet was received. The NIC also writes back in the descriptor the length of the packet. The driver can then look at that flag and the length and then retrieve the packet. After retrieving all received packets, the driver updates its version of the `HEAD` pointer to signal new available descriptors for reception. The execution flow for polling the receive description ring is shown in Listing 9.

```
while (1) {
        volatile union i40e_rx_desc rx_desc = rxq->ring[rxq->head & (rxq->len - 1)];
        qword1 = rte_le_to_cpu_64(rxdp->wb.qword1.status_error_len);
        rx_status = (qword1 & I40E_RXD_QW1_STATUS_MASK) >> I40E_RXD_QW1_STATUS_SHIFT;

        // Check that there is at least one packet to receive
        if (!(rx_status & (1 << I40E_RX_DESC_STATUS_DD_SHIFT))) {
            break;
        }

        // Check checksums calculated by hardware and drop packets in case of error
        // [...]

        // translate descriptor info into mbuf parameters, read length from descriptor field
        struct mbuf *b = get_mbuf(rx_desc);
        b->len = ((qword1 & I40E_RXD_QW1_LENGTH_PBUF_MASK) >> I40E_RXD_QW1_LENGTH_PBUF_SHIFT)
    ;

        // Refill descriptor with newly allocated mbuf for next packet that comes here
        new_b = mbuf_alloc_local();
        maddr = mbuf_get_data_machaddr(new_b);
        rx_desc->read.pkt_addr = rte_cpu_to_le_64(maddr);

        // Advance head pointer as we received the packet and descriptor can be reused
        rxq->head++;

        // Deliver packet
        if (unlikely(eth_recv(rx, b)))
            mbuf_free(b);
    }
```

Listing 9: Receiving packets

5 Results

The performance of the IX i40e driver with the XL710 NIC was evaluated on one NUMA node with 8 hyper-threaded *Intel(R) Xeon(R) CPU E5-2650 v2* cores running at 2.60 GHz (16 processing units in total). Figure 7 shows the execution environment. Performance benchmarks were compared to the original ixgbe IX driver for 10Gb cards. Comparison with other cards or 4x10Gb cards is for future work.

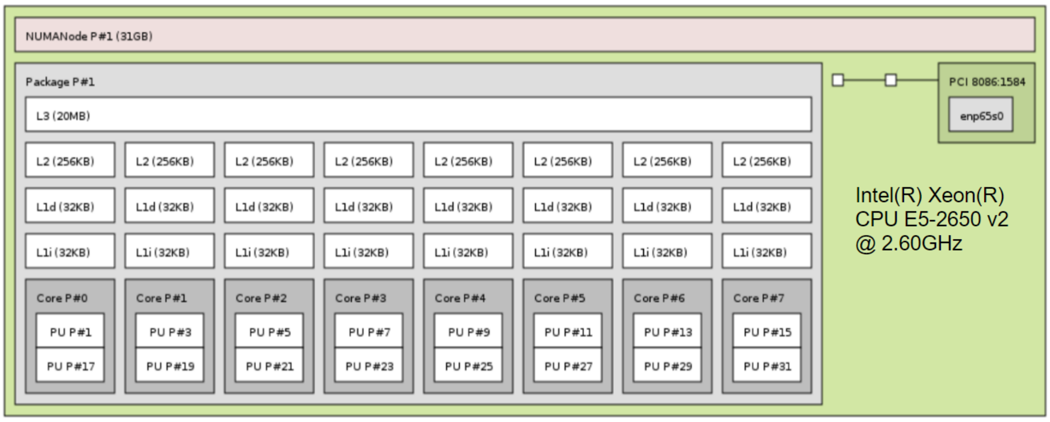

Figure 7: Evaluation environment

The first three performance tests are the same that were used to initially evaluate IX. They are described in more details in section *5.2.2. Throughput and Scalability* in IX TOCS' paper[2]. Figures 8, 9 and 10 show the message rate or goodput for both the ixgbe driver (10GbE card) and the new i40e driver (40GbE card) as we vary the number of cores used, the number of round-trip messages per connection, and the message size, respectively. They involve eighteen clients connecting to one server running IX on a single port with n round-trips (n two-way messages) per connection and messages of size s bytes. The results in the present project could unfortunately not be compared to the results in the TOCS' paper because the processors used were different. This comparison is for future work. Additionally, two `memcached` benchmarks were also run to measure latency as a function of throughput.

Figure 8, with $n = 1$ and $s = 64$B, shows that IX needs only four cores to saturate the 10GbE link while it scales up to at least eight cores with the XL710. We could not test if it scales beyond 8 cores due to limitations in the testing infrastructure. Additional cores would have resided on another NUMA node thus the PCI access time and memory latency would not be symmetric between all cores anymore.

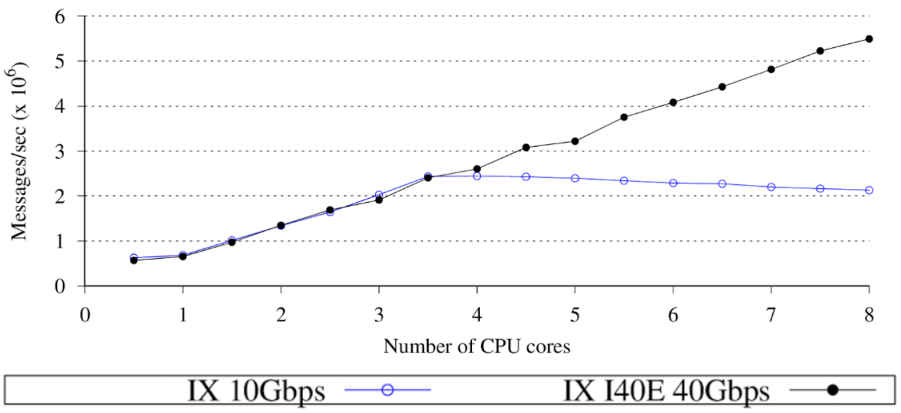

Figure 8: Scalability w.r.t. number of cores

Varying the number of round-trips per connection, n, is shown in Figure 9 with $s = 64B$. i40e's throughput is close to 2X compared to ixgbe's.

Figure 10 shows the goodput when varying the size of messages, with $n = 1$. 8K messages saturate the link with a goodput close to 35Gbps.

The two next performance tests are memcached benchmarks. They were also used to evaluate IX initially and are described in more details in section `5.2.4. Memcached Performance` in IX TOCS' paper[2]. Memcached is an in-memory, key-value used as a high-throughput, low-latency cache for database servers. The graphs show the average and 99th percentile latency, for the ixgbe driver and the new i40e driver, as a function of throughput with a tail latency SLA of 500μs.

The first benchmark, memcached ETC, measures the latency in a scenario where 75% of requests are GET requests with different sizes for keys (20B to 70B) and values (1B to 1KB).

The second memcached benchmark, memcached USR, measures the latency as well in an alternative scenario where 99% of requests are GET requests with short keys (<20B) and 2B values. In other words, almost all traffic involves minimum-sized TCP packets.

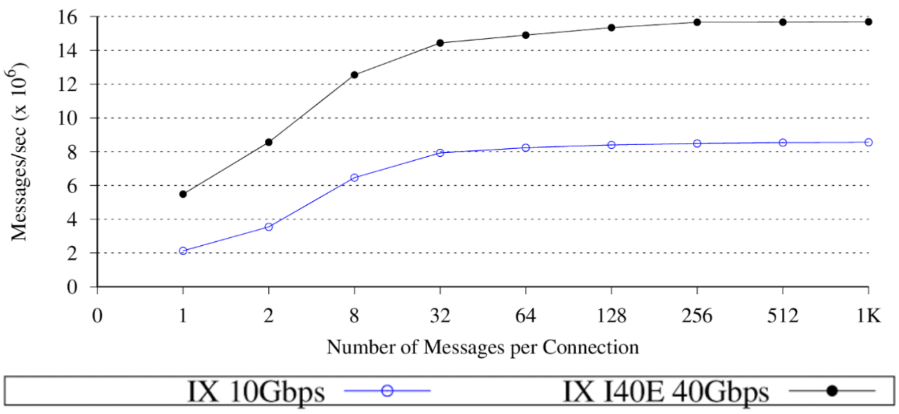

Figure 9: Scalability w.r.t. number of messages per connection

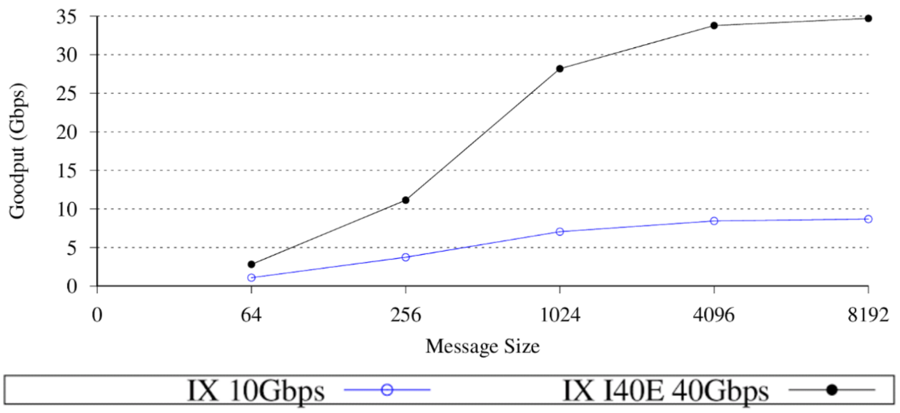

Figure 10: Scalability w.r.t. message size

Memcached ETC shows that latency doesn't degrade as fast with the i40e compared to ixgbe while USR shows the reverse. The reasons for this needs to be investigated in future work.

6 Further Work

6.1 Bugs

There is still one bug that prevents sometimes (roughly a quarter of the time) benchmarks to be run when using more than four cores (eight processing units) and more than eight clients. It happens at initialization time when connecting to all clients. We didn't have time to investigate further but still could run the benchmarks. Our current hypothesis is a race condition in IX memory allocator when allocating memory for the i40e driver.

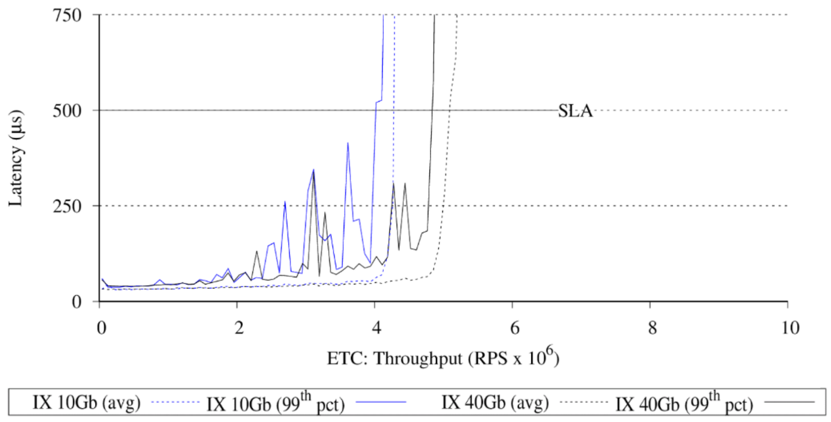

Figure 11: Memcached ETC

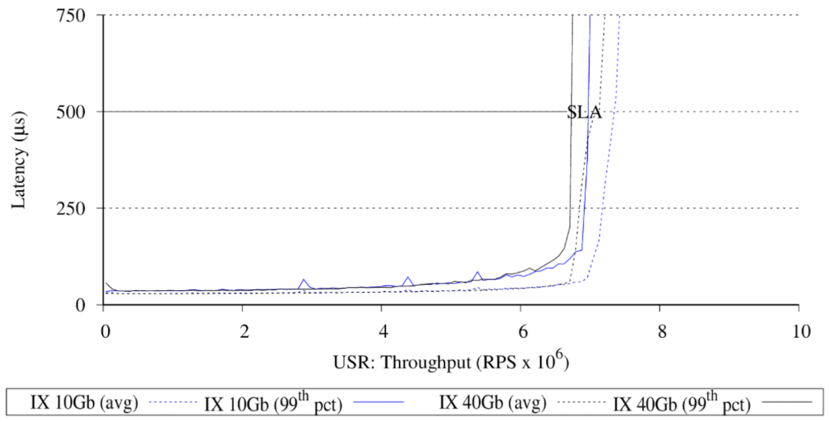

Figure 12: Memcached USR

6.2 Additional Features

The most important missing feature is SR-IOV support. To enable that, the driver first needs to support the IOMMU and virtual functions. This could be a good topic for a new semester project.

Another important missing feature supported by the XL710 is segmentation offload for TCP and UDP (refer to section 2.3). Compared to SR-IOV support, this should not take as much time.

6.3 Benchmarking and Optimization

More benchmarks should be run to isolate critical paths and optimize them. No particular optimization were applied to the RX and TX paths but if they happen to be the bottleneck then they could be optimized. DPDK i40e driver has additional optimizations such as SIMD support to read and write descriptors in batch.

7 Conclusion

This semester project report described adding a new i40e driver to IX to support the Intel XL710 40GbE card. It explained how drivers are supported in IX and how DPDK is involved. IX drivers rely on DPDK for initialization and general configuration while rewriting their own transmit and receive paths to have complete control over the critical paths.

Refactoring needed to make IX driver-independent and separate ixgbe code from IX core code were first described followed by the proper i40e driver implementation. For more details, readers should look at the open-source IX repository on Github `https://github.com/ix-project/ix`. They can check the pull requests that introduced the changes and/or check drivers in the `dp/driver` folder.

The XL710 being a 40GbE card and the comparison was with 10GbE cards, the performance tests were expected to show an increase in throughput and goodput. Latency improvement is not that clear yet and its analysis should be refined in future work. Leveraging more of the advanced hardware features of the card should also be done in future work.

References

[1] Belay, Adam, et al. "IX: A protected dataplane operating system for high throughput and low latency." 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014.

[2] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. ACM Trans. Comput. Syst. 34, 4, Article 11 (December 2016), 39 pages. DOI: https://doi.org/10.1145/2997641

[3] DPDK Data Plane Development Kit Official Website
http://dpdk.org/

[4] Understanding DPDK Presentation
http://www.slideshare.net/garyachy/dpdk-44585840

[5] Belay, Adam, et al. "Dune: Safe user-level access to privileged CPU features." Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). 2012.

[6] Intel® Ethernet Controller 710 Series Datasheet. Revision: 2.9 April 2017
http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf