UNIVERSITY OF CALIFORNIA, SAN DIEGO

Fine-Grained End-Host Traffic Control

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Sen Zhang

Committee in charge:

Professor Alex C. Snoeren, Chair
Professor George C. Papen
Professor George M. Porter

2015

The thesis of Sen Zhang is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

Chair

University of California, San Diego

2015

DEDICATION

To Wei, who is always there to support me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

My stay at the graduate program in the University of California, San Diego has been an important and valuable stage of my life. It is the two years when I learned the most. I was very fortune to work with many talented people and I owe to a lot of them for their help and guidance.

I want to thank my advisor Alex Snoeren. Alex got me started in research and guided me throughout the last two years. He always spends time discussing my work with me and provides extremely insightful feedback. I remember the guidance he gave me on how to tackle a difficult problem, the suggestions on how to present data, and the questions he asked about any of my figures.

I am grateful to the faculty members of the REACToR project. I want to especially thank George Porter for the support he gives me and the inspiring discussions and interesting questions. To George Papen and Geoff Voelker, thank you for all the discussions and particularly explanations in the weekly meetings. Special thanks to George Papen for serving on my thesis committe. I also want to thank Rolf Neugebauer and Awanish Verma (Netronome) for supporting my work.

To Alex Forencich, Rishi Kapoor, He Liu, Feng Lu, and Malveeka Tewari, it was a pleasure to work with you on REACToR and I am indebted to all of you.

To my best friends, Louis DeKoven, Xi He, Yiqing Huang, Yuming Liu and Xing Xing, I cannot imagine a graduate life without you.

Finally, I want to thank my fiancée Wei. I could not have made it without you. To my brother and sister, Lei and Yuan, who have been there for me for so many years. To my parents, thank you for your trust and unconditional love.

ABSTRACT OF THE THESIS

Fine-Grained End-Host Traffic Control

by

Sen Zhang

Master of Science in Computer Science

University of California, San Diego, 2015

Professor Alex C. Snoeren, Chair

In this thesis, we look at changes in the end-host network stack that are needed to support REACToR, a hybrid data-center network. We discuss the requirements posed by REACToR and propose a host-traffic control protocol. We then describe three different implementations of the protocol.

To schedule optical circuits on a microsecond scale, a hybrid top-of-rack switch needs to control end-host traffic. The most important performance characteristic of the host-control protocol is its responsiveness, namely how quickly can a flow be paused or unpaused.

The first implementation of the host-control protocol is based on priority-based Ethernet flow control. While it provides good performance by taking advan-

tage of hardware implementations in commodity network cards, the implementation has correctness issues.

The second implementation is the Software-defined Data Plane proposal which uses DPDK to provide a flexible user-land network stack. The host-control protocol can be a subset of the Software-defined Data Plane. However since the stack is implemented in software, there are performance penalties. It also requires the original network stack to be replaced, which might entail a lot of changes in end hosts.

The third implementation is a reprogrammed network card built with a Netronome network flow processor. Being a hardware implementation, it has better performance and does not demand changes in the operating system. Thus it can help REACToR achieve the goal of operating "under the radar".

Chapter 1

Introduction

Traditional computer networks are built after a simple abstraction where end hosts use the network as a water pipe and the network is passive from an end host's perspective. But some next-generation network prototypes are based on much stronger interations between the network and end hosts. The REACToR project [6] builds a data-center network that achieves high bandwdith by exposing optical circuits to end hosts. The efficiency of the optical circuits depends on the switch's ability to control end-host traffic explicitly on a fine-grained time scale.

This thesis explores the host-control protocol as required by REACToR and discusses three implementations of the protocol: an extension of priority-based Ethernet flow control, a software implementation using DPDK and a hardware implementation using a network flow processor.

1.1 Related Work

Our work is motivated by the hybrid switch REACToR [6], which creates an innovative network architecture and poses new requirements for the end-host stack. The Data Center Bridging (DCB) work explored similar fine-grained control of end-host traffic in the Ethernet layer, for example Ethernet PAUSE frame and priority-based flow control (PFC) [13]. TDMA-based Ethernet [12] proposes a TDMA MAC layer that forms a tightly coupled network with a centralized link scheduler. It is also based on Ethernet PFC, similar to our work described in Chapter 2. There

has been a lot of work that aims to provide more control of the end-host stack from the perspective of applications or operating systems. Frameworks like the Data Plane Development Kit (DPDK) [2] and netmap [10] provide libraries for fast packet processing. There are also hardware solutions such as NetFPGA [7] and the Netronome network flow processor [8] (discussed in Chapter 4) that make it easier to explore new network hardware. Software-defined Data Plane (SDD) [11] builds a new end-host stack based on DPDK and allows a centralized network controller to control end-host traffic. We discuss SDD in more detail in Chapter 3.

1.2 Background

REACToR addresses the problem that data-center networks are facing ever-increasing demands but current network technologies are not scalable to support higher and higher bandwidth. For example at 100 Gbps or higher, it is too expensive to use copper cables for anything but intra-rack connections. Optical fibers are much more scalable both in bandwidth and cost, but connecting optical fibers with electrical switches requires two expensive optoelectronic transceivers for each cable and results in huge costs for a typical data-center network [6]. We get the most benefits only if both optical fibers and optical switches are employed on the rack level. But the main challenge with an optical switch is its long switching time. REACToR addresses this problem through a hybrid approach. As shown in Figure 1.1, REACToR is a hybrid top-of-rack switch that connects to both an electrical packet-switched network and an optical circuit-switched network. It thus combines the scalability of optical switching and the flexibility of electrical switching.

Optical circuits have several important characteristics. First, optical switches normally have a very limited number of circuits. At any given time, only a small portion of network flows can be serviced through the optical network. Combined with the fact that network flows do not stay constant, the switch has to pick a changing subset of flows for the circuits; other flows must be either paused or rate limited. Second, the optical switch has a non-trivial reconfiguration time when

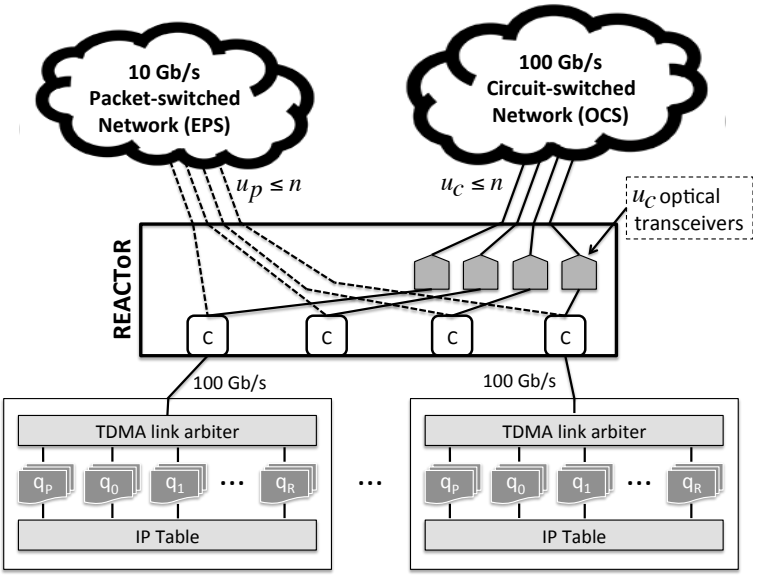


Figure 1.1: REACToR architecture. Each host has a list of separately-controllable queues. Reproduced from [6].

it needs to change circuit assignments. During this time it cannot transmit any traffic, so all flows must be paused and buffered somewhere. Third, the high bandwidth enabled by optical circuits, for example 100 Gbps, means a flow requires a very large queue even if it is paused for a short time period. In REACToR, the optical network is buffer-less and packets are buffered in end-host memory.

These characteristics mean a hybrid REACToR switch needs to control end-host traffic on a fine-grained scale. For example when circuit assignment changes, the switch has to first stop all flows before it can reconfigure the optical switch. After the reconfiguration, it starts a new set of flows on circuits. Such pausing and unpausing must happen as fast as possible, otherwise the utilization of the circuits is impacted. In Section 1.3 we define a host-control protocol that enables such control.

1.3 Host-Control Protocol

To support a hybrid network like REACToR, end hosts need to support a list of operations that are invoked by a top-of-rack switch.

First we define the concept of "flows". For REACToR, a flow is all packets that come in the same switch port and go out another, i.e. packets that are scheduled through the same optical circuit. For an end host, a flow includes all packets with the same IP destination. In our implementations the set of flows are predefined for some hardcoded IP addresses. We might also use IP type of service as the filter when evaluating the prototypes. In a real-world network the IP destinations might need to be updated at runtime, but it is not a primary task of the host-control protocol.

With multiple flows on the network, the switch needs to control each flow individually. Each flow is either stopped by a pause command or resumed by an unpause command. The switch sends commands to end hosts using a schedule packet, which contains a flag for each flow. We refer to the commands as a flow schedule.

1.4 Performance Metrics

When a flow is to be paused, the "off" delay is defined as the time between when the host receives the schedule packet and when the host sends out the last packet from the flow. The "on" delay is defined for the opposite case; when a flow should be unpaused, the time between when the host receives the schedule packet and when the host sends out the first packet from the flow is the on delay. The on and off delays are the offset between switch and end hosts and decide how well a circuit is utilized. Since we can hide the delays by sending out schedule packets ahead of time, the variances of the delays are the most important measurements.

As shown in Figure 1.2, when the variance is 0, we can hide the delays and achieve full utilization of the circuit. But when the variance is not 0, we have to account for the maximum off delay and minimum on delay, leaving the circuit under-utilized.

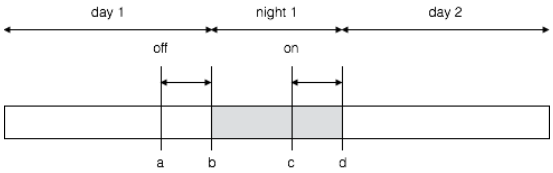


Figure 1.2: On and off delays with 0 variance. The shaded duration is the reconfiguration of the circuit switch. A schedule packet is sent at time a and the traffic stops at b when reconfiguration happens. Another schedule packet is sent at c and the traffic resumes at time d, the same time as the circuit is in place.

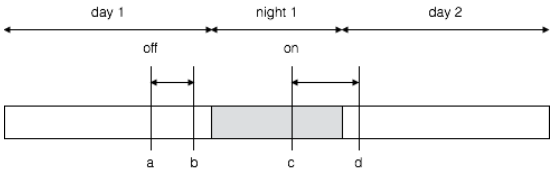


Figure 1.3: On and off delays. When the on and off delays have variance, the traffic might stop earlier (time b) than the reconfiguration, or resume later (time d) than a circuit is turned on

1.5 Summary

In this chapter, we discussed the hybrid REACToR network and introduced the end-host traffic control protocol. We defined the operations of the protocol and its performance requirements. Following chapters of this thesis explore three different implementations of the protocol. The implementation based on the network flow processor is the main part I work on and want to discuss. We compare its performance to the first two implementations and show if it meets the requirements of REACToR.

Chapter 2

Priority-based Ethernet Flow Control

In this chapter we discuss existing Ethernet Flow Control protocols and show if they can be extended to support fine-grained end-host traffic control. We also talk about the performance of current implementations and establish the baseline that we want to achieve with alternative implementations.

2.1 Ethernet Flow Control

In the traditional network model, Ethernet as a link-layer protocol only has to deliver best-effort service. But some applications require a more reliable Ethernet, such as Fibre Channel over Ethernet which cannot tolerate losses in the link layer. Protocols have been developed for these requirements. One is the PAUSE frame (IEEE 802.3x [13]), which allows an Ethernet station to pause the transmission of the other end of a link, thus avoiding packet losses when congestion happens. The PAUSE frame is extended by IEEE 802.1Qbb, which allows 8 classes of traffic to be paused independently.

802.1Qbb enables priority-based flow control for Ethernet (PFC) which is similar to the traffic control protocol we want to implement. PFC defines 8 priorities and allows an Ethernet station to control another station's traffic on a fine-grained time scale. An example of PFC is shown in Figure 2.1, where station

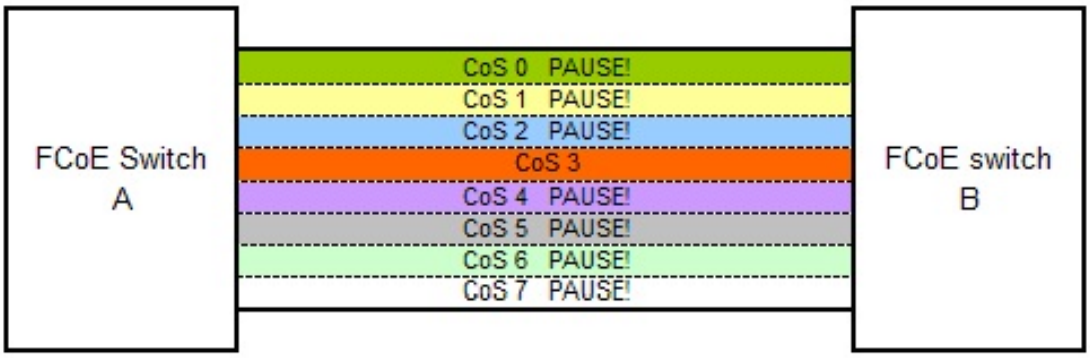


Figure 2.1: A PFC frame is sent specifying all classes except 3 to be paused. Reproduced from [1]

A asks station B to pause all classes except 3. By mapping one network flow to each priority, PFC can be used to control the flows. Section 2.3 discusses the advantages and limitations of this approach.

2.2 PFC performance

PFC controls the 8 traffic classes by specifying a pause duration for each. In the PFC frame, as shown in Table 2.1, there are 8 16-bit time values, each meaning how long a class of traffic should be paused. The pause duration is defined as a number of quanta and each quantum is the time to send 512 bits at the current link speed. On a 10-Gbps link, each quantum is 51.2 ns and the maximum pause duration is about 3.36 ms. A value of 0 means the corresponding class should be unpaused.

The performance of PFC has been well studied. REACToR [6] shows that a 10-Gbps Intel 82599 NIC can pause a flow within 1.0–2.2 μs and unpause a flow within 1.2–1.3 μs. Both the absolute value and the variance of the delay are small. Evaluation results of different implementations are compared to these numbers, because we want to achieve the same performance for our traffic control protocol.

TDMA-based Ethernet [12] shows that there is a significant deviation in how well the requested pause duration is respected. So when using PFC to control flows, we use the timers as binary values: 0 unpauses a flow and 0xFFFF pauses

Table 2.1: PFC frame format

Dest: 01:80:C2:00:00:01
Src MAC addr
Ethertype: 0x8808
opcode: 0x0101
class enable mask
Time (class 0)
Time (class 1)
Time (class 2)
Time (class 3)
Time (class 4)
Time (class 5)
Time (class 6)
Time (class 7)
CRC

a flow for the longest possible period until it is explicitly unpaused. When a flow needs to be stopped for longer than the maximum period, we send PFC frames to renew the pausing.

2.3 Design

The main limitation of PFC is the fixed number of 8 priorities. As shown in Chapter 1, the REACToR switch needs to control all network flows, where each flow is defined as packets with the same IP destination. In a data-center network, there are many more than 8 destinations. For example in a rack of 40 hosts, the top-of-rack switch needs to schedule traffic among at least these 40 hosts. PFC works fine for a testbed with 8 hosts. To support more hosts, we need to reuse the 8 priorities.

In REACToR, circuits are assigned in the units of days, and days are organized into weeks. Within a week, a flow could get a circuit for at most one day. So

it is an intuitive idea that we can limit each week to 8 days and use PFC within each week. When there are more than 8 flows, they can be scheduled in multiple weeks. At the beginning of each week, the host is updated with the set of 8 flows and remap them to the PFC priorities.

The following algorithms show the high-level logic that the switch and the host should implement.

```
switch-control-loop:
    For each week:
      Pick 8 flows and notify end hosts
      For each day:
        Send PFC frames for circuit schedules for the 8 flows

host-pfc-handler:
    If new weekly schedule received:
      Map a set of 8 flows to PFC priorities
    If new PFC received:
      Pause or unpause specified queues
```

2.3.1 Residual packets

However, because of the existence of queues, the mapping of PFC priorities cannot be switched immediately.

In an Intel 82599 NIC, PFC is implemented by 8 hardware queues called packet buffers [3]. Each packet buffer corresponds to a PFC priority and incoming PFC frames control if packets can be sent from the packet buffers. When we change the mapping between flows and packet buffers, we cannot control packets that are already in the packet buffers. In fact, depending on where the remapping happens, there might be more queues in between, thus more of such packets.

We refer to these as residual packets, which are packets sent from a flow that is supposed to be paused. Residual packets are a problem because they violate the traffic control protocol requirements. Essentially they cause a prolonged off delay as the flow takes considerable time to stop. For REACToR, an implementation with such residual packets leads to bad circuit utilization. Our goal in this chapter is to measure the number of residual packets and evaluate the implementation of the traffic control protocol based on remapping PFC priorities.

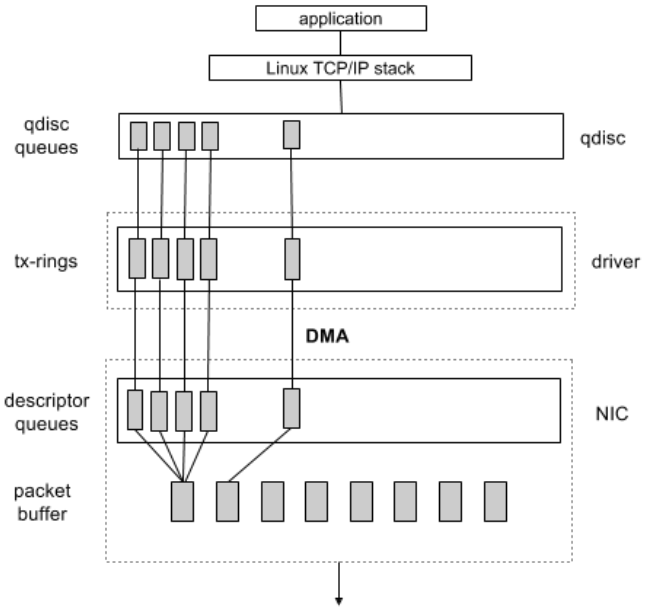


Figure 2.2: Queues in the Linux network subsystem

2.3.2 Queues in Linux networking subsystem

In this section we talk about queues in the Linux networking subsystem and where we can remap flows to different packet buffers.

As shown in Figure 2.2, there are a few layers of queues. The qdisc (queuing discipline) layer classifies packets into flows and each qdisc queue corresponds to a different flow. Tx-rings are the queues in the device driver; descriptor queues and packet buffers are hardware queues.

For outgoing packets, they go though each layer sequentially. There is a mapping scheme between the different layers since they do not have the same number of queues. In the case of an Intel 82599 NIC and the Linux ixgbe driver, there are 64 qdisc queues, 64 tx-rings and 128 descriptor queues. We can change which flow is controlled by each PFC priority by modifying the mapping between any two layers.

But the only reasonable choice is the mapping between qdisc queues and tx-rings. The other mappings are either hardcoded or require too much work to change: the mapping between descriptor queues and packet buffers is fixed in

hardware. Tx-rings and descriptor queues are part of the DMA interface and any change requires a lot of care to ensure DMA performance;

When a qdisc queue is remapped to a different tx-ring, we get residual packets from the tx-ring, descriptor queue and packet buffer. We can remedy the problem using rate limiters in the Intel NIC. Each descriptor queue can be limited to a minimum of 1000th of maximum rate; i.e. 10 Mbps for a 10-Gbps card. Thus we can reduce the number of residual packets from descriptor queues and tx-rings. But residual packets from the packet buffers cannot be controlled.

Since we try to assign flows to separate queues in each layer, the number of flows supported by the entire system is limited to the smallest number of queues. In this case, the above design could support 64 flows as in tx-rings layer.

The qdisc layer also handles new flow schedules. The NIC driver captures schedule packets and pass the schedule to qdisc.

2.4 Implementation

We implement the queue remapping in the Linux kernel. The two main parts are the ixgbe driver and the multiq qdisc.

For any outgoing packets, the driver checks its qdisc queue number and the flow schedule, and puts the packet into the correct tx-ring. When a schedule packet is received, the driver parses the schedule and sends an update to the qdisc layer. The multiq qdisc only sends packets from the enabled queues.

To classify packets into different flows, we create Linux traffic control (tc) filters and put packets into different qdisc queues. So a qdisc queue corresponds to a flow.

2.5 Evaluation

Our implementation has the same on and off delay as Ethernet PFC since we do not change how PFC frames are handled. A PFC frame still controls the packet buffers directly and can pause or unpause a packet buffer with the same

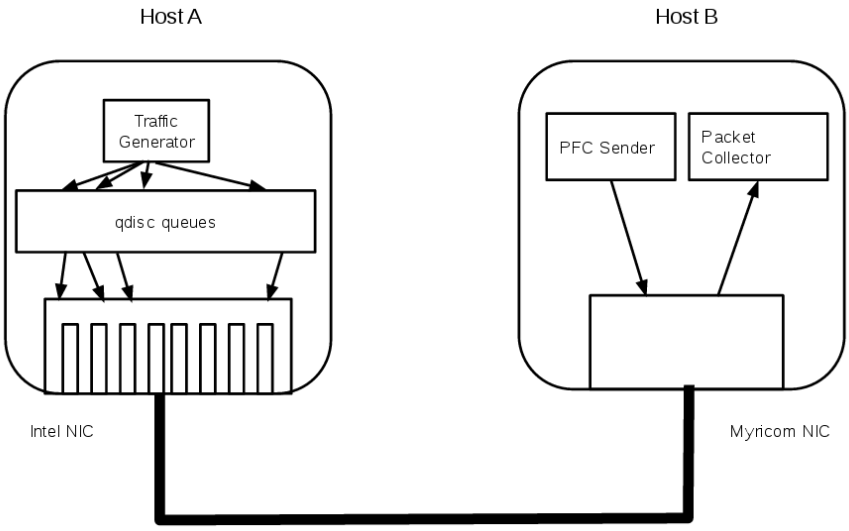


Figure 2.3: Testbed including two servers. Host A acts as a normal server. Host B acts as a switch.

latency.

But because of the existence of residual packets, when a flow is enabled we might be sending packets from a wrong flow. To measure the residual packets and determine where they come from, we evaluate the queue remapping on a testbed as shown in Figure 2.3.

2.5.1 Testbed setup

The testbed has two machines directly connected using an Ethernet cable. Both machines run Debian Linux 7.0 with kernel 3.4.44 x86_64 and Intel Xeon E5520. Host A acts as an end host with an Intel 82599EB 10-Gbps Ethernet card. Host B acts as a top-of-rack switch and is equipped with a Myri-10G Dual-Protocol NIC from Myricom. The Myricom NIC supports kernel-bypassing mode which allows us to send and receive raw Ethernet frames. So Host B is used to send PFC control frames.

Host A runs a traffic generator that sends MTU-sized packets, which are classified into different flows according to the IP type of service. On Host B there is a PFC sender and a packet collector. The PFC sender sends schedules to A,

including the weekly schedule at the start of each week and the daily PFC frames during each week. The packet collector saves all packets sent by A and records the receiving time and index of each packet.

An example schedule is shown below, which is equivalent to assigning a circuit to 16 flows sequentially and achieving a staircase traffic pattern.

```
schedule: 0 1 2 3 4 5 6 7
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
schedule: 8 9 10 11 12 13 14 15
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

We vary the day length and schedule for different experiments in order to show the minimal schedule interval that can be supported, the number of residual packets, and how different schedules can be supported.

2.5.2 Residual packets

Figure 2.4 shows the trace of 16 flows using the above schedule. In the bottom right of the figure are the residual packets. Table 2.2 shows the packet trace when the second week starts. What we want to see is flow 0-7 are disabled and 8-15 enabled. But before packets for flow 8 come in, we first receive a few packets from flow 0. After that, we get 1-2 packets from flow 0 after every 500-800 packets from flow 8.

These are the residual packets and the reason is that flow 0 and 8 share

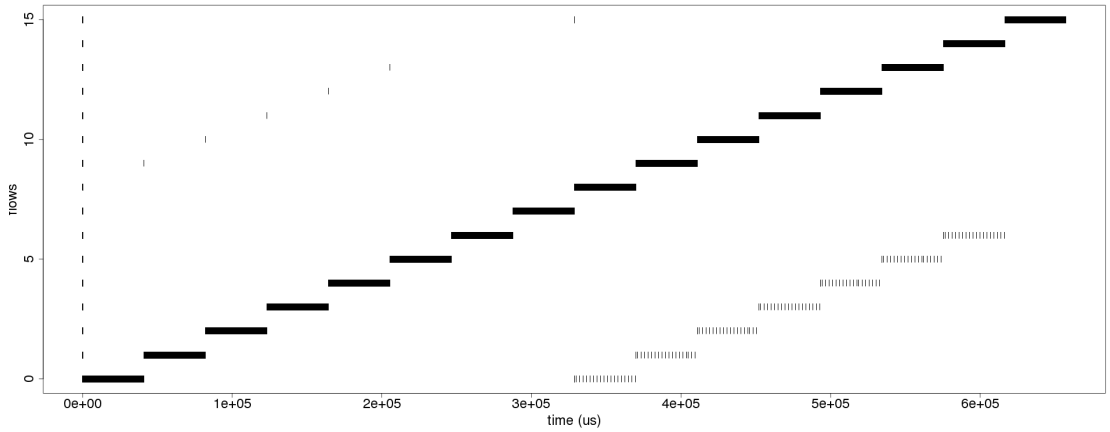


Figure 2.4: Packet trace for 16 flows and day length 40 ms.

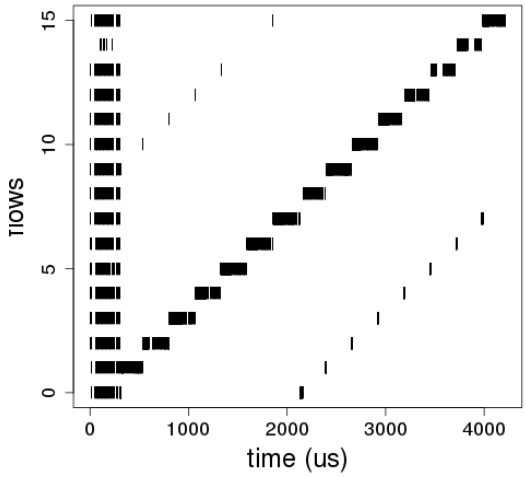


Figure 2.5: Packet trace for 16 flows and day length 200 μs.

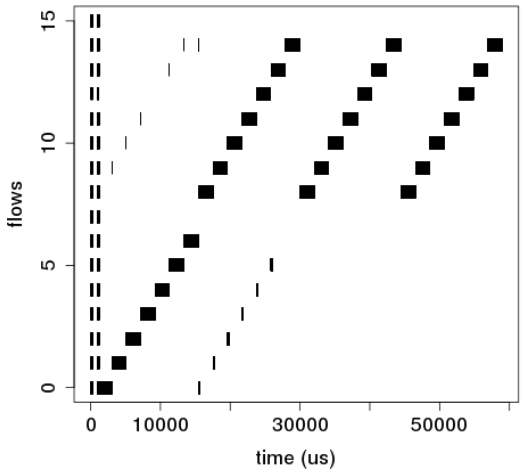


Figure 2.6: Packet trace for 16 flows and day length 2 ms.

the same packet buffer, descriptor queue and tx-ring in the NIC. The initial 8-9 packets come from the packet buffer because that is roughly the size of a packet buffer. The later ones are from the rate limited descriptor queue. Packets in the descriptor queues for flow 0 and 8 are sent to the same packet buffer, which is why the two flows are interleaved and flow 0 is at a much lower rate.

Figure 2.5 shows the same schedule but with a shorter day length. Since

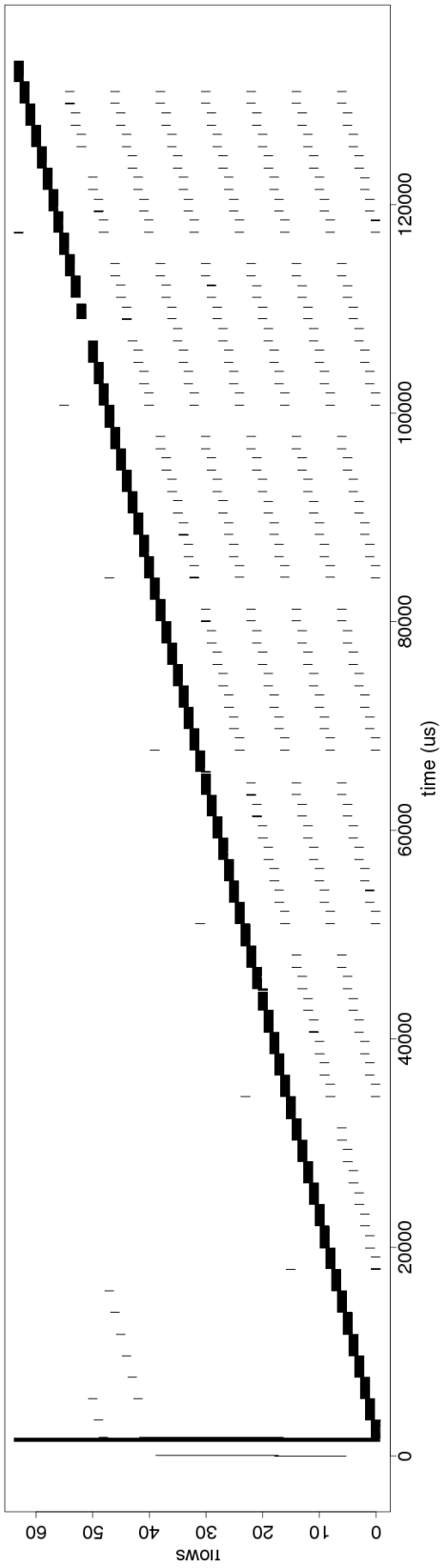


Figure 2.7: Packet trace for 64 flows and day length 2 ms.

Table 2.2: Packet trace when new week starts. The index column shows packet sequence numbers within each flow. The timestamp column shows when each packet is received by the packet collector.

flow	**index**	**timestamp** (μs)
0	42783	18049
0	42784	18050
0	42785	18052
0	42786	18053
0	42787	18054
0	42788	18055
0	42789	18056
0	42790	18057
8	38907	18058
0	42791	18060
8	38908	18061
8	38909	18062
8	38910	18063
8	38911	18064

each flow is enabled for a short time, we only get the first batch of residual packets from the packet buffer.

Figure 2.6 shows the same schedule but with flow 8-15 enabled for a few more weeks, and wihout rate limiting on the descriptor queue. We can see that during the second week, all residual packets are sent out when a day starts. They do not appear in later weeks because the tx-ring, descriptor queue and packet buffer have been flushed and qdisc does not send down more packets.

For completeness, Figure 2.7 shows a schedule with 64 flows enabled.

2.6 Summary

As described in Section 2.3, residual packets are the primary problem when reusing the PFC priorities. The residual packets come in two batches, one from the packet buffers and the other from rate limited descriptor queues. The first batch can be hidden by viewing it as part of the variance, as shown in Section 1.4. However the second batch cannot be hidden as easily. Even though the number of residual packets in the second batch is limited and the cost in the context of REACToR is a small utilization drop, they violate the requirements of the traffic protocol. Hence we look for alternative implementations without the residual packets problem.

Chapter 3

Software Implementation with DPDK

Packet processing frameworks like the Data Plane Development Kit (DPDK) [2] and netmap [10] make it possible to implement a host-control protocol in software. In fact we can implement the entire end-host stack with these libraries.

A significant advantage of a packet processing framework is its flexibility. These frameworks provide direct access to the network and allow novel features to be included in the network stack implementation. However, a possible drawback and the main issue we examine is performance penalty, since flow schedules can only be handled on a higher software level. Another downside of building everything on top of such frameworks is the necessity to rewrite the entire stack, including all network protocols. User applications might have to be recompiled or modified too.

In this chapter we discuss the design of the Data Plane Development Kit and a data-plane implementation that supports end-host traffic control.

3.1 Data Plane Development Kit

DPDK is a framework that supports fast packet processing on commodity hardware. It includes a device driver and a software library that are optimized to reduce the overhead of processing packets from Linux userland. It delivers good

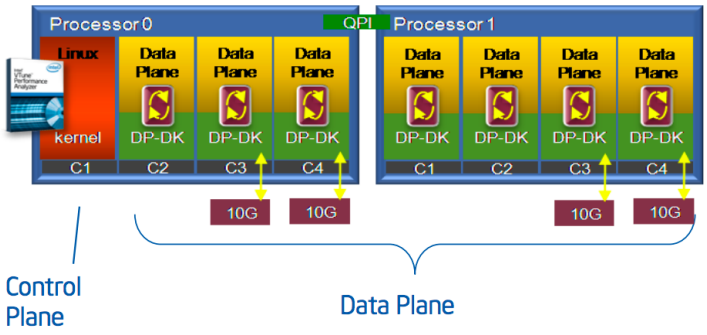


Figure 3.1: Data plane based on DPDK. Reproduced from [4].

performance by taking advantage of multi-core processors and their increasing speed; see Figure 3.1 where the control plane is the original operating system while the data plane uses DPDK and runs on dedicated cores. Using DPDK an application can send packets to and receive packets from the wire directly, with minimum latency caused by the DPDK framework in between. With such flexibility, DPDK can be used to to implement a network stack and the host-control protocol.

DPDK provides a generic interface to applications called the Environment Abstraction Layer (EAL) [5]. EAL includes optimizations for specific operating systems and hardware but hides the details from applications. As shown in Figure 3.2, applications are linked against the abstraction layer and userland libraries.

EAL supports multi-threading with EAL threads, called "logical cores" (*lcores*), which are implemented by Linux pthreads. The EAL threads are typically pinned to different CPU cores. For example an application might have a TX *lcore* and an RX *lcore*. Using multiple physical cores eliminates context switching and ensures fast packet processing.

DPDK also provides several libraries optimized for packet processing, among which the most important ones are the ring, mempool and mbuf libraries. To reduce memory overhead, DPDK requires dedicated memory and uses huge pages to minimize TLB misses. To allocate and free memory for packets, the mempool library provides an API to manage fixed size objects, with various optimizations such as per-core local cache. Packets are stored in fixed size buffers called mes-

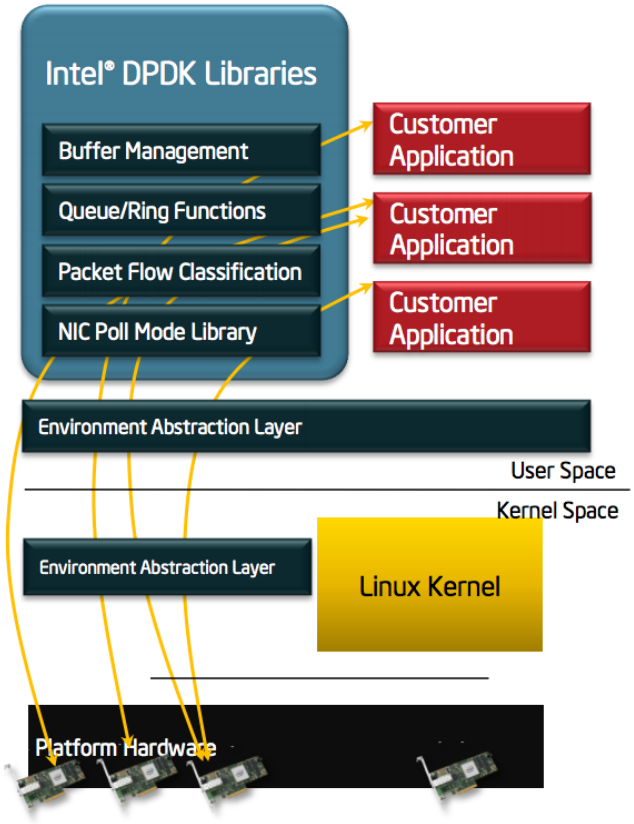


Figure 3.2: DPDK components. Reproduced from [4].

sage buffers and are allocated and freed by the mbuf library from a memory pool. Each mbuf contains metadata and payload of a packet and might link to additional mbufs. The ring library provides support for lock-free FIFOs that can be used as TX and RX queues.

Besides CPU cores and memory, DPDK also needs to optimize accesses to the network card and does so with Poll Mode Drivers (PMD). A PMD enables packet transmission and reception without interrupts. Instead, the application has to poll the driver for incoming packets. It sends and receives packets in bursts where a small burst size can reduce latency and a large size can improve throughput.

A poll mode driver allows packets to be processed in a simple run-to-completion model, where each *lcore* retrieves a burst of packets and completes the processing before handling the next burst. Packets do not have to be passed

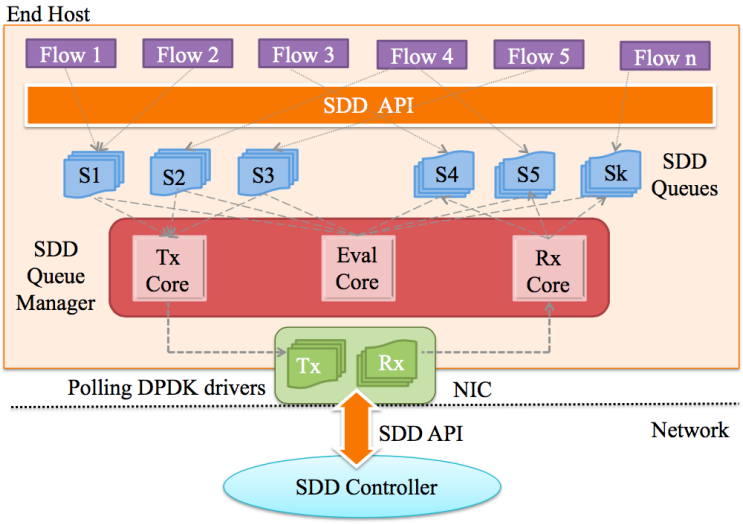


Figure 3.3: System overview of the SDD architecture. Reproduced from [11].

through intermediate queues. A pipeline model is also possible with a PMD, where each *lcore* is responsible for a different stage of the processing and packets are passed between them through queues. Obviously the run-to-completion model is simpler and might have a better performance by avoiding the queue overhead and having better cache locality.

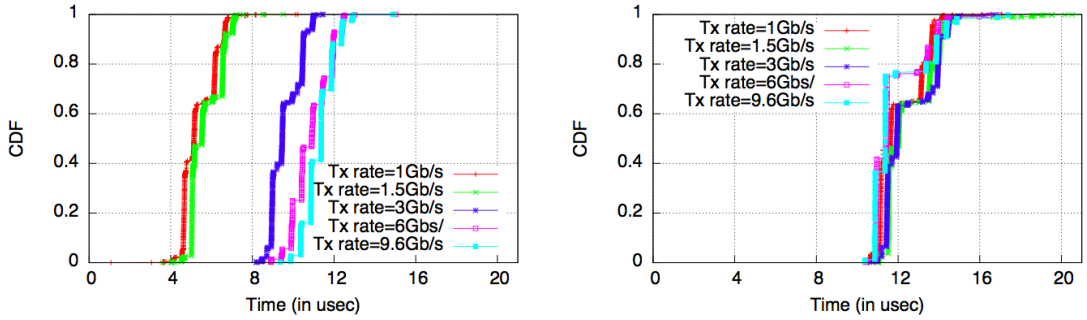
3.2 Software-defined Data Plane

Software-defined Data Plane (SDD) [11] uses DPDK to build an end-host stack that supports centralized network controllers. In this section, we discuss the design of SDD and how its implementation covers the host-control protocol.

SDD addresses the problem that while modern data-center networks are becoming increasingly centralized, end hosts are not part of the centralized control. It implements both the control plane and data plane on top of DPDK in order to provide responsive control of network flows.

As shown in Figure 3.3, the key components of SDD are SDDQueues and an API that controls when traffic can be transmitted from an SDDQueue.

An SDDQueue is a shared memory region that are accessible to SDD and a userland network stack. An SDDQueue can be associated with one or more network



(a) Distribution of *RemoveDequeueCondition*() latency

(b) Distribution of *AddDequeueCondition*() latency

Figure 3.4: Characterizing the responsiveness of adding and removing conditional dequeue commands. Reproduced from [11].

flows and its "dequeue condition" determines how packets should be transmitted from the queue, such as the time and the rate. The SDDQueue API allows a network controller to manage the queues, for example to change how flows are mapped to SDDQueues or set a timer of when a queue is drained. One of the design goals of SDD is to allow fast control of the queues. It exploits the performance and features of DPDK such as using separate cores for Tx and Rx and a third core for handling API calls.

Our host-control protocol can be seen as a subset of SDD features. To implement the protocol, each flow is bound to a separate SDDQueue based on their destinations. Whenever a flow needs to be paused or unpaused, the switch sends a new dequeue condition to the end host. The on and off delays are equivalent to how fast SDD can carry out new dequeue conditions.

3.3 Evaluation

The implementation of the host-control protocol described above is essentially the same as the "tightly-coupled TDMA" experiment of SDD. The RemoveDequeueCondition call is used to stop a queue and the AddDequeueCondition call to start a queue. Their response times are the off delay and on delay respectively.

Figure 3.4(a) shows the distribution of the off delay, i.e. the time between

when RemoveDequeueCondition is invoked and the flow is stopped. The mean of the off delay increases as transmission rate is higher, possibly because of heavier queuing below DPDK. The CDFs have long tails. While most data points fall in a range of 4 μs, the range of all points are considerably longer. For example when Tx rate is 1 Gb/s, the range is close to 8 μs.

Figure 3.4(b) shows the distribution of the on delay, which stays the same regardless of the transmission rate. Most data points are still in a range of 4 μs but there is a long tail especially when Tx rate is 1.5 Gb/s.

3.4 Summary

In this chapter we first explored the design of a packet processing framework DPDK. We also talked about SDD, a data plane based on DPDK that supports remote controllers, and how it can be used to implement our host-control protocol. We included the evaluation results of SDD. Compared to the PFC implementation in Chapter 2, where the commodity hardware achieves on/off delay with a range of less than 2 μs, the software-based implementation has significantly longer delay. A data plane on top of software packet processing libraries also requires dedicated CPU cores and memory which might not be desired by a data center. Another drawback with such an approach is that it requires the entire network stack to be replaced. A lot of engineering effort might be needed before real applications can be evaluated.

Chapter 4

Hardware Implementation with Network Flow Processor

Since the software stack using DPDK has a much larger variance of off/on delay than PFC, we explore the possibility of a hardware implementation. If achievable, it is likely to have much better performance, close to PFC implementations. But modifying or creating hardware demands substantial work. A potential platform is the network flow processor from Netronome, which is a network interface card that can be reprogrammed with a high-level language. Another disadvantage with hardware solutions is that they require hardware investments, of course. But compared to rewriting the entire network stack and dedicating CPU cores to network processing, it is possible that a hardware solution provides better performance guarantee and may be a better choice for data centers whose scale warrants custom hardware.

This chapter discusses a third implementation of a fine-grained end-host traffic control protocol using a Netronome network flow processor. By reprogramming a network card and evaluating its ability to do traffic control, we show that fine-grained traffic control can be implemented efficiently in the NIC hardware.

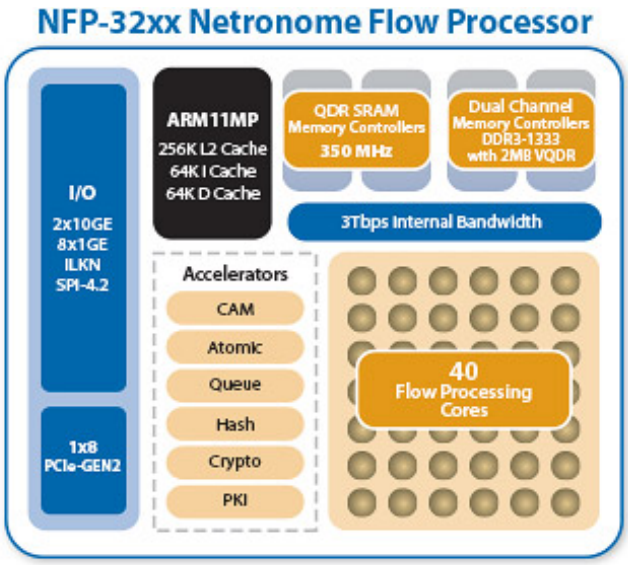


Figure 4.1: NFP-32xx block diagram. Reproduced from Netronome website [8]

4.1 Network Flow Processor

We use a network flow processor (NFP) from Netronome [8] as the base platform. The NFP is optimized for networking applications and is programmable as a network card. In this section we briefly describe the features of the NFP-3240 that relate to our case.

4.1.1 NFP-3240

The NFP-3240 has 40 cores (also called microengines or MEs) at 1.4 GHz, each of which has 8 hardware threads (also called contexts); see Figure 4.1. They are 32-bit RISC cores and organized into 5 clusters. Resources on the NFP are allocated to one of four different levels: thread, ME, cluster or the entire device.

Besides MEs, the other class of key components of the NFP are functional units [9]. They can be seen as peripherals or accelerators with specialized microengines that handle networking-related tasks. For example, the PCIe controller provides high-speed data transfers between the NFP and the host, and the memory unit provides access to the card's internal DRAM.

The NFP has a lot of data storage units, both registers and memory [9]. Each ME has 256 general purpose registers (GRP), which are used in either context-relative mode where each context owns a subset of the GPRs, or absolute mode where a register is visible to all contexts. Each ME also has 4 KB of local memory (LM), which provides more general purpose registers from a programmer's perspective. To communicate with functional units, such as reading or writing memory, every ME has access to 512 transfer registers, half of which are used for reading and half for writing. There are also 128 next-neighbor (NN) registers in each ME which are used for inter-ME communication or as more general purpose registers.

Besides these different registers, the NFP has several other types of storage units. Each ME cluster is equipped with a cluster-local scratch (CLS) which is 64 KB of SRAM and used for inter-ME communciation within a cluster. The device has 4 GB of DRAM that is accessible by all MEs through the Memory Unit. There is also an SRAM component but the storage is actually backed by DRAM.

As mentioned above, each ME on the NFP is multi-threaded. The 8 contexts are scheduled in a cooperative and round-robin fashion. A context keeps running until it yields control voluntarily and only one context can be running at any time.

Signals are the primary mechanism in the NFP to achieve asynchronous operations. They are usually raised by functional units to notify an ME of I/O completion, but they can also be used for inter-context or inter-ME communication. Contexts in the same ME communicate through either shared registers or signals.

The many registers and threads on the NFP define a very special programming model. Programmers have access to a large number of registers and write programs that are similar to high-level C code. Even though the MEs are multi-threaded, programmers do not have to use locks since there is no preemption. But they do need to take multi-threading into account when issuing long-running I/O operations and must wait for signals explicitly. Programmers must also consider what registers a variable is allocated to, for example if a variable is only visible to each context or shared between all contexts.

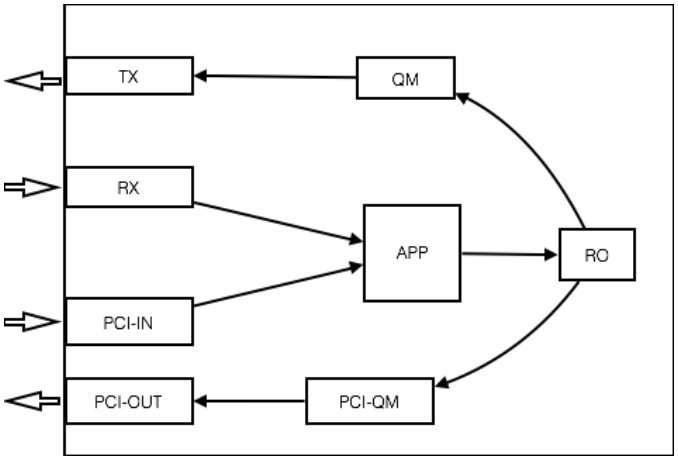


Figure 4.2: Architecture of NIC implementation on NFP-3240. Each rectangle is a "block" in the firmware. There are two blocks (PCIE-SVC and another PCI-OUT) not shown in the graph.

4.1.2 NIC implementation on NFP

The NFP-3240 already works as a network card. The current implementation enables one of the two ports on the card and makes it a 10-Gbps NIC.

As shown in Figure 4.2, the NIC implementation is made up of a few blocks that run on different MEs. Most of the blocks run on one ME while the application block uses 16 MEs. A total of 25 MEs are used by the current firmware.

The application block is the central piece that implements the logic of the processor. As a network card, the application block takes packets from the PCIe interface (PCI-IN), and sends them to the Ethernet link (TX). On the receiving path, it gets incoming packets from RX and sends them to PCI-OUT. The RO block (RO meaning reorder) ensures the packets are not out of order. There is also a Queue Management (QM) block on both the incoming and outgoing path; they provide buffers to avoid packet losses. All blocks other than application are called infrastructure blocks, since they perform general tasks and may be used for other purposes of the NFP, for example as a switch instead of a NIC.

4.2 Design

The NIC implementation described in the previous section essentially supports a single stream of packets. Because we want to control each flow independently, we need a number of parallel queues. We also need to classify packets of each flow into different queues and control these queues according to a flow schedule.

Two potential blocks in which to implement the queues are QM and TX, since they give us the most control of packets. When flow schedules change, we only have control over packets in the queues. Once a packet leaves a queue, it can no longer be stopped. Similarly when we want to restart a flow, its packets can only reach the wire following previous packets. So to minimize the off and on delay, the queues should be put as close to the physical link as possible.

The last software block in the packet pipeline is the TX block, where packets are handed to the hardware Ethernet controller. We can theoretically achieve minimal delay variance by putting the queues in TX. But TX does not allow such modifications because it has to process packets under a strict time requirement to achieve a desired throughput. The QM block gives us much more freedom since it works with packet descriptors instead of actual packet data. We choose to implement the queues in QM and Section 4.4 evaluates the throughput of the NIC.

With the different queues, we associate each flow with a queue and need to classify packets into flows. All packets must be inspected and marked with a flow number, which has to be accessible by the QM block so that it knows the queue each packet should go to. The classification is based on packet data, such as destination IP address or Type of Service. There are two places in the current NIC implementation where we can afford to look at the data of each packet, the kernel driver or the application block. The kernel driver can easily look at each packet, but it has to pass the flow number within packet descriptors all the way to QM, through the DMA interface, PCI-IN block, application block and RO block. Much less change is needed if the classification is done in the application block. It is much closer to QM and already reads some bytes of each packet to check

the header fields. So our implementation modifies the application block to classify packets.

To pass the flow information to QM, the application block embeds flow numbers in packet descriptors, which contain several unused fields.

The last part of the design is how to handle new schedules from the switch. Again our goal is to minimize off/on latency, so it is best if the schedules are processed as early as possible. RX is the first block that receives incoming packets, but like TX it is heavily optimized and any change can easily affect the NFP performance. The current RX does not look into packet data either, so it is impossible to recognize and parse a schedule packet without issuing expensive memory reads. The next block in the pipeline is the application block, which runs on many threads and is able to look at each packet. So the application block is where we capture schedule packets. To communicate a schedule to the QM block, any of the inter-ME communication mechanisms can be used.

4.3 Implementation

We implement the new NIC firmware based on a software release for NFP-3240 from Netronome. This section talks about the implementation details of each design decision mentioned in Section 4.2.

4.3.1 Classifying packets

As mentioned in Section 4.2, the infrastructure blocks work with packet descriptors. There are two parts to each descriptor. The first part is a 32-bit handle that refers to a *packet buffer* that stores packet data. The second part is a much larger data structure called metadata, which contains more information about each packet including packet length as well as application-specific data.

One of the application-specific fields can be used to save flow numbers. Metadata is initialized by the application block and to classify packets, it looks at packet headers and marks the flow number in metadata. The metadata is stored in DRAM and read later by the QM block.

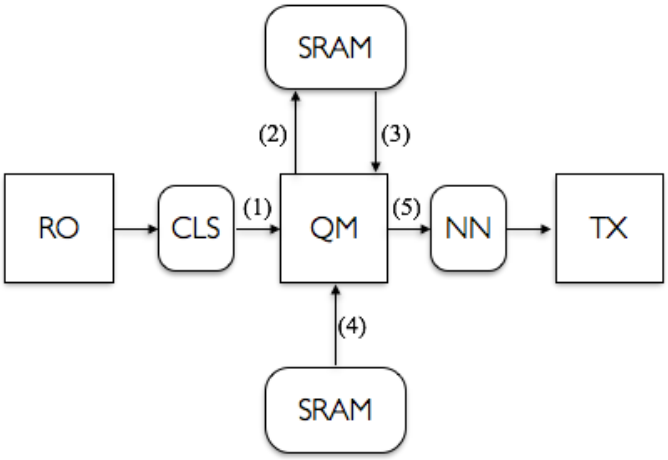


Figure 4.3: Architecture of original QM block

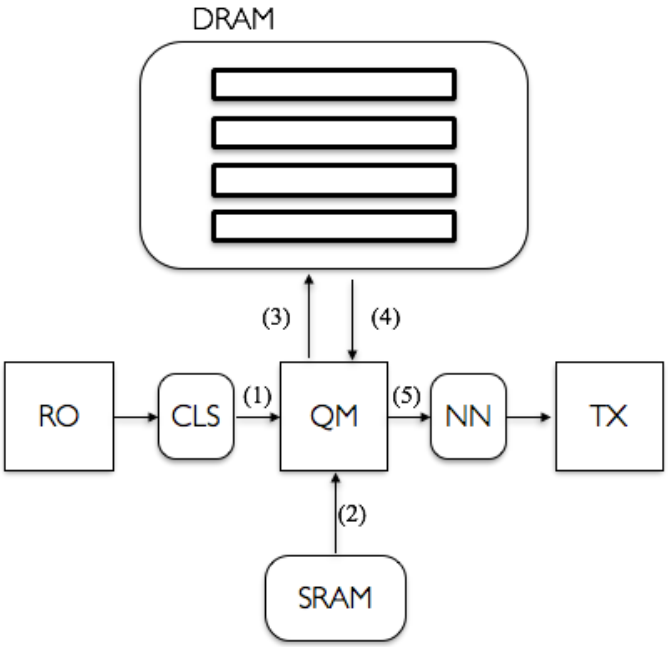


Figure 4.4: Architecture of new QM block

4.3.2 Multiple queues

The queues are implemented in the QM block. To make further changes easier we first translate the original implementation from assembly to C.

The main function of QM is to receive packet buffer handles from RO and pass the corresponding packet metadata to TX. The processing is divided into 5

steps, as shown in Figure 4.3: (1) QM reads a packet handle from RO through the CLS ring; (2) QM stores the handle in an SRAM queue (which is actually backed by DRAM); (3) QM dequeues a handle from the SRAM queue; (4) QM reads the corresponding metadata from the SRAM metadata queue (also backed by DRAM); (5) QM sends the handle and metadata to TX through the NN ring.

The original QM block essentially passes all packets through a single queue in step (2) and (3). We need to replace it so that packets from different flows are put in different queues and then dequeued according to the queue schedule.

The new QM implementation has the same 5 steps but in a different order. (1) When QM gets a packet handle, (2) it reads the corresponding metadata right away, which contains the flow number marked by the application block. (3) QM then puts both the handle and metadata in the correct DRAM queue. QM also keeps a local copy of the queue schedule; if a flow is enabled, (4) QM reads from its queue and (5) sends the handle and metadata to TX. If a flow is paused, no dequeue is done so packets from the flow are not transmitted.

The main change is that the single SRAM queue in Figure 4.3 is replaced by multiple DRAM queues. Since the SRAM interface is backed by DRAM, there is no performance penalty. Of course we now read and write more data because metadata is also stored in the queues, but the impact is negligible. In our prototype implementation, we have 16 queues.

The assembly implementation of the original QM runs a single loop that finishes all 5 steps. Our new implementation in C breaks the loop into two parts, one of which handles the enqueue steps (1) - (3) and the other handles the dequeue steps (4) and (5). This results in shorter and simpler code since each part of the loop is executed by different contexts. To further simplify the code, the I/O operations are done in a synchronous fashion (wait for each I/O operation to complete before starting another). The NFP compiler parses such code better and we can avoid heavy annotation to ensure correct register allocation.

4.3.3 Receiving schedules

The last part is to supply the queue schedule to QM used in step (4). As mentioned in the Section 4.2, the schedule packets are received in the application block. It looks at the headers of each incoming packet and recognizes schedule packets.

To send a schedule to QM, the application block writes to a transfer register in QM. Since they run on different MEs, we have to declare the transfer register as remotely accessible and import the register in the applicaiton block. Then it can be written to through the CAP interface (CSR Access Proxy; CSR meaning Control and Status Registers).

4.3.4 Format of schedule packets

We use a custom IP protocol number to designate schedule packets. The payload of the packet is similar to PFC frames as described in Chapter 2, containing a simple 16-bit mask that specifies if each flow should be pause or unpaused.

4.4 Evaluation

We evaluate two characteristics of the NIC: its throughput as a normal NIC and the off/on delay it can achieve for the host-control protocol. We use an NFP-3240 that is installed on an HP server with Intel Xeon E5520 and Linux 3.2.57 (Host A). Another server is used to send flow schedules and receive packets (Host B). It is equipped with a Myricom Myri-10G Dual-Protocol NIC. The servers are connected through a Cisco Nexus switch.

Before we discuss performance, Figure 4.5 shows the 16 flows controlled independently, similar to experiments in Chapter 2, Compared to Chapter 2, here we can see the individual packets because the days are shorter and as shown later, the throughput is below line rate. When each day starts, there is a burst of packets because of the queue; once the queue is drained, the flow gets back to the normal speed. The longer a flow has been paused, the bigger a burst. Finally when all flows are enabled, we see a similar pattern of bursts.

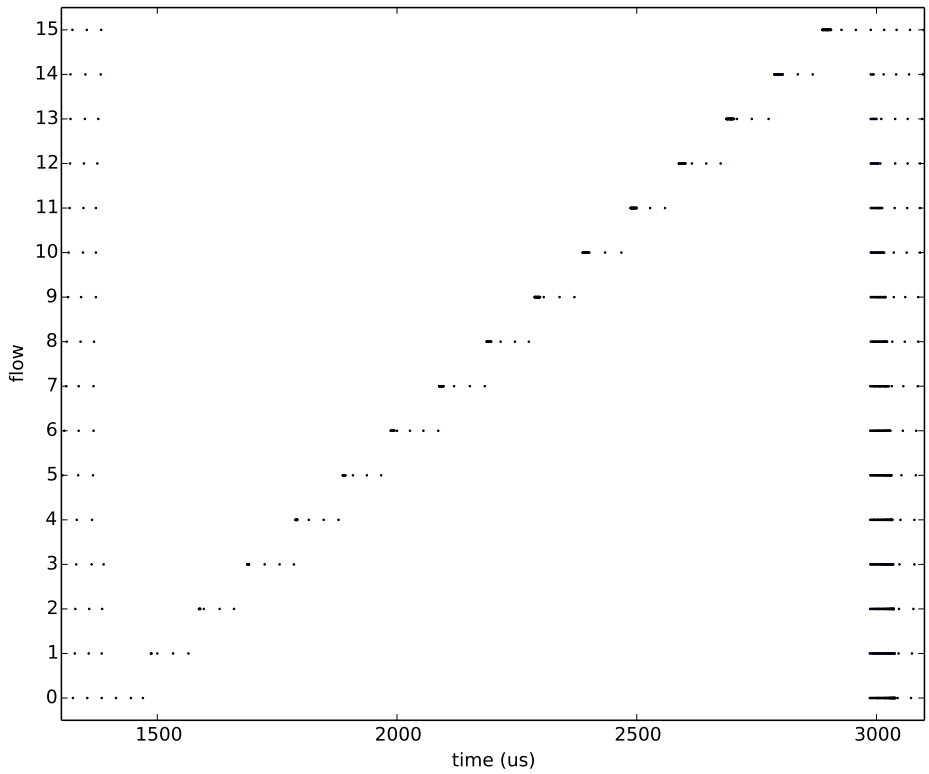


Figure 4.5: Packet trace for 16 flows and day length 100 μs.

4.4.1 Throughput

We measure the throughput of the NIC with both the original firmware and the new firmware with our modifications. By comparing the results, we make sure our modifications do not result in performance degradation. We use iperf to measure both sending and receiving throughput. For sending throughput, Host A with the NFP runs iperf client and Host B with the Myricom card runs iperf server. Receiving throughput is measured the other way around.

Figure 4.6 shows the throughput of the original NFP firmware. We measure different number of flows (as different number of iperf client threads). For each flow count, iperf is run for 10 times, each time for 10 seconds. The box plots show the distribution of the evaluation results, including median, first and third

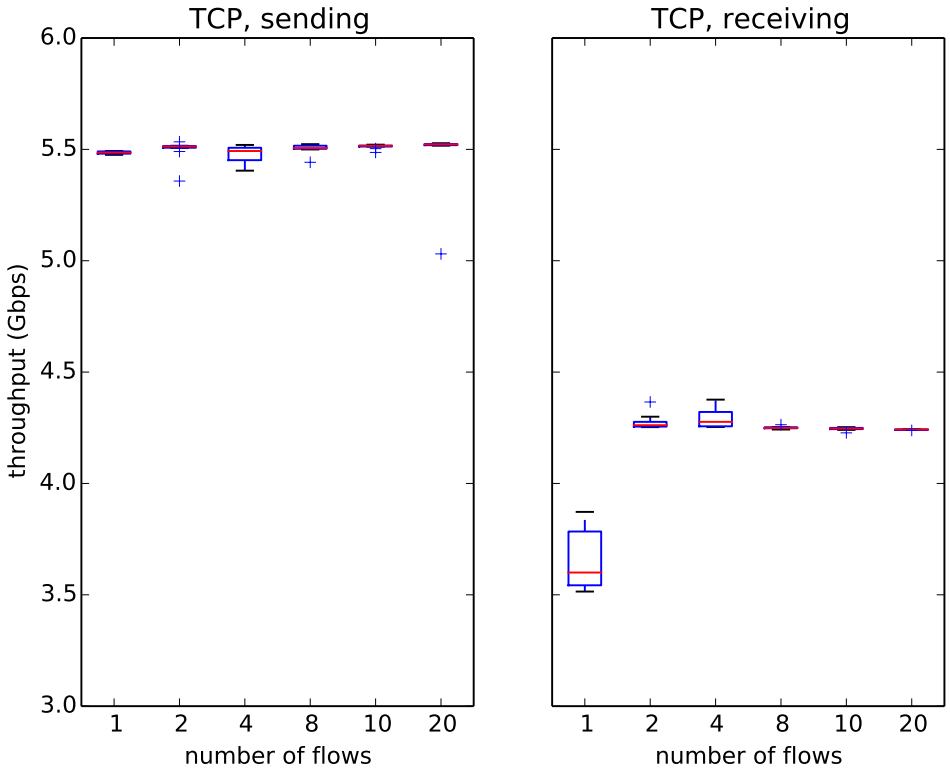


Figure 4.6: NFP throughput with original firmware

quartiles, minimum and maximum, and outliers. The NFP achieves 5.5 Gbps for sending with any number of flows and 4.2 Gbps for receiving with more than 2 flows. When there is only 1 flow, the receiving throughput is lower (3.6 Gbps) and has much larger variance. However the NFP is supposed to work as a 10-Gbps NIC. Bottlenecks potentially exist in the kernel driver or the firmware.

Figure 4.7 shows the throughput after our modifications. It is about the same as Figure 4.6 for both sending and receiving throughputs. For sending with 20 flows, the box plot shows a larger variance (with smaller third quartile) while in Figure 4.6 the lowest data point is considered an outlier. The reason is that a throughput of 5.0 Gbps happens once in Figure 4.6 and three times in Figure 4.7. However the median values are close. We conclude that the NIC has the same performance even with extra queues and handling of schedule packets.

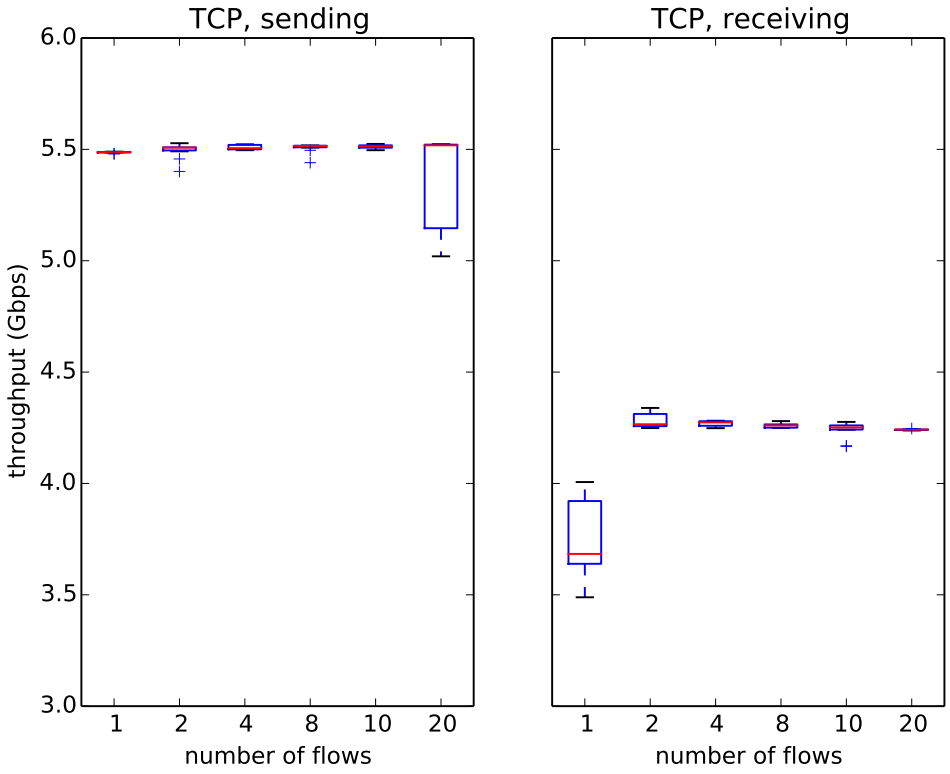


Figure 4.7: NFP throughput with new firmware

4.4.2 Off/on delay

To measure on and of delays, we have to capture the timestamps of both flow schedule packets and data packets in order to know how long it takes for a flow to start or stop. The Myricom NIC on Host B gives us the timestamp of each data packet, but not timestamps of schedule packets because we only know when packets are sent to the outgoing queue, not when they are place on the wire.

To obtain the exact timing between data packets and schedule packets, we use the SPAN (Switch Port Analyzer) feature of the Cisco Nexus switch. SPAN monitors a specified port on the switch and mirrors all traffic to another port. We need a third server (Host C or sniffer) which receives the duplicated packets of those sent and received by Host A. Ideally we would have used ERSPAN (Encapsulated Remote SPAN) that provides nanosecond-scale timestamps, but it is not supported

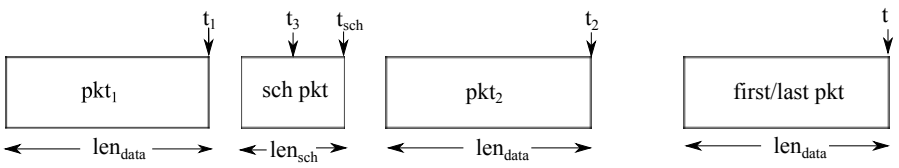


Figure 4.8: Timestamps of schedule packets.

on our version of hardware.

We restore the timestamps of schedule packets by combining packet traces from Host B and Host C. As shown in Figure 4.8, the timestamp of a schedule packet t_{sch} can be inferred from two preceding and following data packets. From the packet trace captured by Host C, we find the two data packets, pkt_1 and pkt_2. The packet trace from Host B gives us their timestamps t_1 and t_2. We calculate t_{sch} as $t_{sch} = t_3 + \frac{len_{sch}}{2}$ where $t_3 = \frac{t_1+(t_2-len_{data})}{2}$. To compute off delay, we search for the last packet of the paused flow in the packet trace from Host B, which has timestamp t as in Figure 4.8. The off delay is $t - t_{sch}$. For on delay, we search for the first packet in the flow and the delay is $(t - len_{data}) - t_{sch}$. We subtract len_{data} from t because the start of a flow is marked by the beginning of the packet. In our experiments, schedule packets are 64 bytes while data packets can be configured to different sizes.

The precision of off/on delay depends on how accurate is t_{sch}, which depends on several factors such as packet gaps and if SPAN can even capture all packets. We discuss these issues later.

In our first experiment, Host A sends two flows (with two different IP types of service) to Host B, which receives the packets and saves their sequence numbers and timestamps. Host B sends schedule packets to Host A to pause and unpause flow 0 for 10,000 times. In total there are 20,000 schedule packets. Each time flow 0 is paused or unpaused for 100 μs (day length). The switch mirrors all packets to Host C which records their sequence numbers, including those of the schedule packets. We need more than one flows because otherwise it is impossible to reconstruct timestamps of *on* schedules. If there is only one flow and it is paused, no packet is transmitted before the schedule packet. We refer to these extra flows as background flows.

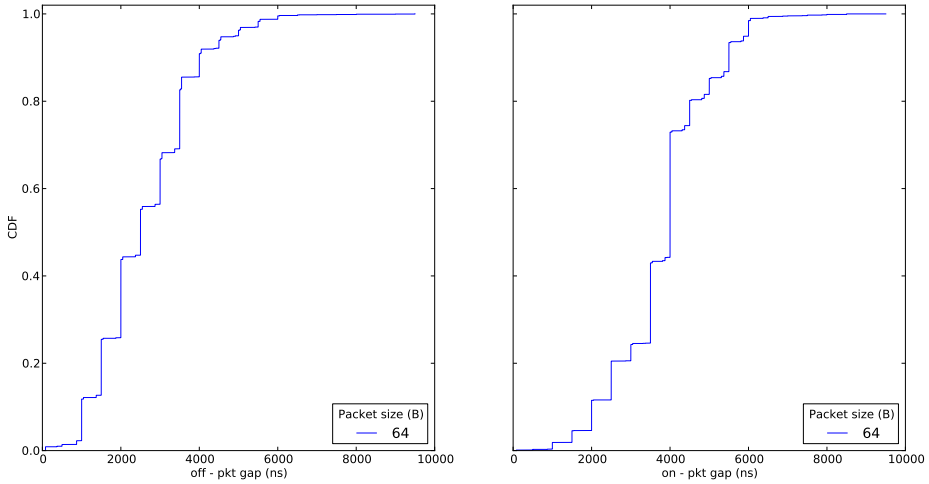


Figure 4.9: CDF of packet gaps around each each schedule packet

The sniffer captures 19,998 schedule packets, however not all are useful. 8379 packets are received back to back (or with only one data packet in between) even though they are sent with 100 μs intervals. This problem might be caused by bad timing of the receiver or sending schedule packets through a switch. Data-packet gaps (between t_1 and t_2) are unreasonably large (over 10 μs) for 17 schedule packets. 10 of the gaps are between 80-173 μs which is similar to a clock problem we observed on these servers before, where the clock jumps ahead 120 μs every 1.2 milliseconds. 5 of the rest are close to 10 μs, 1 is 600 μs and the last one is 3 ms. No packets are lost in all of these cases, so again the cause might be in the switch or the Myricom NIC.

After removing these problematic schedule packets, we are left with 5802 *off* schedules and 5780 *on* schedules. Figure 4.9 plots the CDF of the packet gaps. The majority of the gaps are below 6 μs. The median is 3 μs for *off* schedules and 4 μs for *on* schedules. This matches the data rate reported by the packet generator, which can only send at a similar packet rate regardless of packet size. With packet size of 64 B, it can generate traffic at 680 K packets per second, or 350 Mbps. The average packet gap is 1.47 μs. Since the packet gap plotted in Figure 4.9 includes two packets ($t_2 - t_1$ includes a data packet and a schedule packet), a median of 3

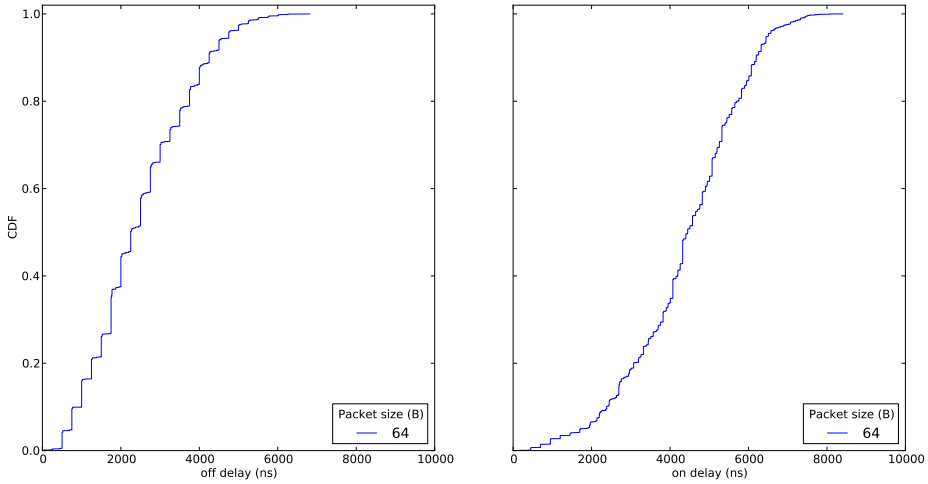


Figure 4.10: CDF of off and on delays

μs makes sense. The packet gaps for *on* schedules are slightly longer because when they are sent, only one flow is enabled and thus the data packets have a larger gap.

Table 4.1: Off and on delays (μs)

	mean	std	range	min	max
off delay	2.48	1.26	6.77	0.04	6.82
on delay	4.40	1.47	8.27	0.14	8.40

We calculate off delay and on delay as described above, after taking out those with unreasonably large gaps. Figure 4.10 shows the CDF of the off and on delays. See Table 4.1 for the mean, standard deviation and range. On delay has a negative value because By comparing range to the results in Chapter 3, the NFP has a slightly better performance man SDD but still much worse than PFC implementations. However if we take into consideration of the inaccuracy of the packet gaps, the actual performance should be more desirable than shown here. But we defer it to future improvements to the evaluation testbed.

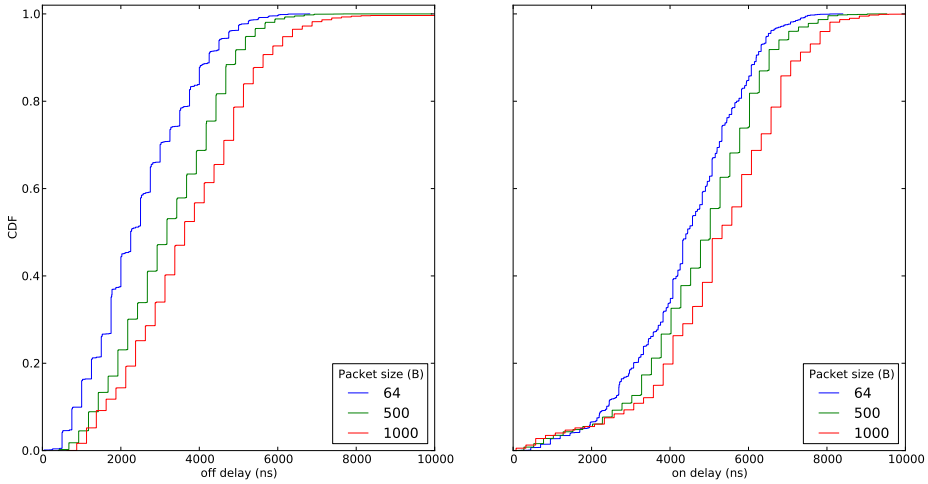


Figure 4.11: CDF of off and on delays for different packet sizes

More experiments

The above experiment has a few hard-coded settings, for example data rate, data packet size, number of background flows and which flow is paused/unpaused.

We did not evaluate different data rates due to limitations in our traffic generator, but due to bottlenecks in the NFP, changing packet size is equivalent to changing data rate.

Figure 4.11 shows off and on delays for different packet sizes. The delays generally fall below 10 μs, although there is a long tail as the size increases, possibly due to SPAN dropping packets. We omit results for 1400 B because too many packets are dropped.

Both off delay and on delay follow a similar distribution for different packet sizes, with similar variance. The mean values grow with packet size, which can be explained by that larger packets take longer to transmit. For off delay, larger packet size means more data to transmit before a flow is stopped. Similarly for on delay, before a flow is restarted, more data has to be transmitted from the background flows.

Table 4.2 and Table 4.3 show evaluation results for different number of flows. Table 4.4 and Table 4.5 show results for controlling different flows when

Table 4.2: Off delay for different packet size and number of flows. Each cell shows the mean and standard deviation (μs).

	packet size		
nflow	**64B**	**500B**	**1000B**
2	2.81 (1.28)	3.63 (1.48)	4.66 (10.83)
4	2.08 (1.32)	2.37 (2.06)	3.01 (3.83)
8	1.52 (1.25)	1.78 (2.97)	2.12 (3.03)
16	1.25 (0.87)	1.43 (1.22)	1.75 (1.19)

Table 4.3: On delay for different packet size and number of flows. Each cell shows the mean and standard deviation (μs).

	packet size		
nflow	**64B**	**500B**	**1000B**
2	4.32 (1.53)	4.99 (1.17)	5.33 (1.68)
4	4.30 (1.01)	4.61 (1.20)	5.47 (1.25)
8	4.44 (1.58)	4.84 (0.90)	5.57 (1.61)
16	4.31 (1.24)	5.39 (3.10)	5.47 (1.28)

Table 4.4: Off delay for pausing different flows. Each cell shows the mean and standard deviation (μs).

	packet size		
controlled flow	**64B**	**500B**	**1000B**
0	2.01 (1.45)	2.53 (2.21)	2.83 (2.61)
1	2.04 (1.30)	2.32 (1.55)	2.83 (1.66)
2	1.96 (1.41)	2.44 (2.01)	2.78 (3.67)
3	2.01 (1.67)	2.35 (2.02)	3.00 (3.74)

there are 4 flows in total. Each experiment sends 100,000 *off* schedules and *on* schedules.

When the number of background flows increases, off delay becomes smaller because the traffic generator sends packets in a round-robin fashion and packets from different flows are perfectly interleaved. With more packets from background

Table 4.5: On delay for pausing different flows. Each cell shows the mean and standard deviation (μs).

controlled flow	packet size		
	64B	**500B**	**1000B**
0	4.18 (1.18)	4.81 (0.93)	5.26 (1.50)
1	4.20 (0.99)	4.53 (1.34)	5.39 (0.96)
2	4.08 (1.12)	4.69 (0.91)	5.21 (1.70)
3	4.04 (1.15)	4.49 (1.18)	5.36 (1.15)

flows in the queue, the controlled flow can be stopped faster. On delay stays largely the same regardless of the number of flows which is expected. Table 4.4 and Table 4.5 show that off delay and on delay do not change no matter which flow is controlled by schedule packets. Here each column should contain the same value, which is also equal to the corresponding cell in row $nflow = 4$ in Table 4.2 and Table 4.3. The results are close to our expectations.

4.5 Summary

In this chapter we discussed the design and implementation of a host-control protocol in the network interface card. Using a programmable network flow processor, we added multiple queues in the NIC with support for fine-grained flow schedules. We evaluated the performance of the NIC and compared it to alternative implementations including PFC and SDD. Its performance is slightly better than SDD, but not yet close to PFC in commodity NICs.

Due to limitations in the evaluation testbed, we devised a way to infer timestamps of schedule packets from two packet traces. But the testbed still has a lot of imperfections, preventing us from running some experiments like large packet sizes.

Our conclusion is that it is viable to implement a fine-grained end-host traffic control protocol in the NIC hardware. It has better performance than a software implementation and does not require changes in the operating system or

network stack. The card can act like a normal NIC to conventional switches, but can also work with a hybrid switch like REACToR.

Chapter 5

Conclusion

In this chapter, we review the different implementations of the host traffic control protocol, discuss future directions and conclude the thesis.

5.1 Host-Control Protocol and the Three Implementations

The end-host traffic control protocol is motivated by the hybrid switch REACToR. We describe details of the protocol and its performance requirements. What we care most is the on delay, off delay and their variance.

There are three possible implementations of the protocol. The first is based on the priority-based Ethernet flow control protocol. To support the use case in REACToR where there are more than 8 flows, the PFC priorities need to be reused. However to reuse the priorities we have to remap queues in the system which causes the problem of residual packets. We measure the residual packets and show that such an implementation does not meet the requirements of the traffic control protocol, because packets from the wrong flows are sent for considerable time. But the original PFC implementations in commodity NICs have excellent performance with regard to on and off delays. They set the baseline for evaluating future implementations.

We then discuss the second implementation of the traffic control protocol

Table 5.1: Comparison of the three implementations. Each row shows different statistics but the numbers are all in μs.

	off-delay	on-delay
PFC	min (max): 1.0 (2.2)	min (max): 1.2 (1.3)
SDD	mean (range): 10 (4)	mean (range): 12 (4)
NFP	mean (std): 2.48 (1.26)	mean (std): 4.40 (1.47)

based on DPDK. With the flexibility and performance of DPDK, it can be used to implement the whole network stack. A recent proposal by Software-defined Data Plane [11] provides an end-host stack that supports centralized network controllers. We talk about how SDD can be used as an implementation of the host-traffic control protocol with reasonable performance. The primary disadvantage of a solution on top of DPDK is that it requires a lot of changes in end hosts, including replacement of the network stack and recompiling of applications.

The main part of this thesis is a hardware implementation of the protocol that hides the changes in the network from end hosts. Using a Netronome network flow processor, we reprogram a network interface card to support fine-grained traffic control. Such an implementation allows REACToR to achieve the goal of operating "under the radar", with applications and the network stack unaware of circuits in the network. We describe details of the new NIC design and evaluate various performance characteristics. Its on and off delays are better than the SDD implementation, but still worse than PFC. However given the inaccuracy in the evaluation setup, better experiments are needed to provide more precise measurements. Since a hardware implementation makes host-traffic control transparent to the operating system, the network flow processor provides a feasible solution to implementing the control protocol.

Table 5.1 shows the known statistics of the delay for each implementation. Since the published results on the first two do not include mean and standard deviation numbers, we can only compare the available statistics.

5.2 Future Work

The hardware implementation based on NFP supports 16 flows. A future improvement is to extend to more flows and evaluate the scalability of the NIC. Right now a list of counters are associated with each queue which represent the number of packet descriptors. Three microengine threads serve the 16 queues by searching through them sequentially. Queues can be added by simply declaring more DRAM queues in the QM block, but the sequential search should be evaluated to test its scalability.

The kernel driver for the NFP can also be improved. The current implementation only supports a single queue in the kernel, while having multiple QM queues on the card. The status of the QM queues is not reported to the kernel driver, thus when any QM queue is full, packets coming into that queue are silently dropped. A multi-queue kernel driver will work better with the new firmware. The kernel driver needs to declare multiple queues to the upper layers. The firmware (PCI-IN block) then pulls packets from these queues according to QM-queue status. When a QM queue is full, packets for other queues are still transmitted.

Another thing we would like to have done is more precise measurements of the NIC implementation. As mentioned in Section 4.4.2, the timestamps of schedule packets are only inferred from packet traces. Ideally we could use something like ERSPAN to observe the timestamps directly. With better timestamps, we can get a more accurate evaluation of on and off delays and the performance of the hardware implementation.

Bibliography

[1] Cisco. Priority flow control: Build reliable layer 2 infrastructure. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809_ns783_Networking_Solutions_White_Paper.html, 2009.

[2] Intel Corporation. Data plane development kit. http://dpdk.org.

[3] Intel Corporation. Intel 82599 10 gigabit ethernet controller: Datasheet, 2013.

[4] Intel. Intel data plane development kit overview. http://www.intel.com/content/dam/www/public/us/en/documents/presentation/dpdk-packet-processing-ia-overview-presentation.pdf, 2012.

[5] Intel. DPDK programmer's guide. http://dpdk.org/doc/pdf-guides/prog_guide-2.0.pdf, 2015.

[6] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M Voelker, George Papen, Alex C Snoeren, and George Porter. Circuit switching under the radar with reactor. In *Proceedings of the 11th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA*, 2014.

[7] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*, pages 160–161. IEEE, 2007.

[8] Netronome. NFP-32xx. http://www.netronome.com/product/nfp-32xx/.

[9] Netronome. NFP programmer's guide, 2015.

[10] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[11] Malveeka Tewari, Alex Snoeren, and George Porter. Practical, centralized control of programmable end-host data planes, In submission.

[12] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C Snoeren. Practical tdma for datacenter ethernet. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 225–238. ACM, 2012.

[13] Wikipedia. Ethernet flow control — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Ethernet_flow_control`.