Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Bachelor's Thesis Nr. 95b

Systems Group, Department of Computer Science, ETH Zurich

Dedicating NIC hardware queues to applications on the Barrelfish OS

by

Roni Häcki

Supervised by

Dr. Kornilios Kourtis, Pravin Shinde

March 2013–September 2013

inf | Informatik
Computer Science

# Contents


**1 Introduction** **3**

**2 Related Work** **4**

2.1 User-level networking . . . 4
2.1.1 Arsenic . . . 4
2.1.2 Xok/ExOS's application-level networking . . . 5
2.1.3 The Packet Filter . . . 5
2.2 Offloading . . . 5
2.2.1 NICOS . . . 5
2.3 Operating system architecture . . . 6
2.3.1 x-Kernel . . . 6
2.3.2 Dragonet . . . 6

**3 Background** **7**

3.1 Barrelfish . . . 7
3.1.1 Capabilities . . . 7
3.1.2 Inter-dispatcher communication . . . 7
3.1.3 Network architecture . . . 8
3.1.4 Dedicating queues . . . 13
3.2 Mackerel . . . 15
3.3 Network Controller . . . 15
3.3.1 Buffer Address Translation . . . 15
3.3.2 Multiple Queues . . . 15
3.3.3 Virtualization (SR-IOV) . . . 18
3.3.4 Interrupts . . . 18
3.3.5 Management-Controller-to-Driver Interface (MCDI) . 19
3.3.6 Offloading . . . 20
3.3.7 User-level networking . . . 20

**4 Approach** **21**
4.1 Driver . . . . . 21
4.1.1 I/O . . . . . 21
4.1.2 Single queue driver . . . . . 22
4.1.3 Multiple queues and filters . . . . . 22
4.1.4 User-level networking . . . . . 22

**5 Implementation** **24**
5.1 Driver implementation . . . . . 24
5.1.1 Single queue . . . . . 24
5.1.2 Multiple queues and filters . . . . . 25
5.1.3 User-level networking . . . . . 26
5.2 Problems during implementation . . . . . 27
5.2.1 Writing registers of the card . . . . . 27
5.2.2 Reading registers of the card . . . . . 28
5.2.3 Packet transmission . . . . . 28
5.3 Unresolved Issues . . . . . 30
5.3.1 Gottardo . . . . . 30
5.3.2 Filters . . . . . 30
5.3.3 User-level networking . . . . . 31

**6 Evaluation** **32**
6.1 Comparison with Linux . . . . . 32
6.2 OpenOnload . . . . . 33

**7 Conclusion** **35**
7.1 Further work . . . . . 35

# Chapter 1

# Introduction

Concurrency is one of the keywords of the last years, regarding hardware and software for Network Interface Cards (NICs). The increasing number of applications accessing the card, can lead to low performance with traditional NICs that offer only a single hardware queue. Following the trend of increasing the core count, hardware designers of modern high-performance cards raise the number of queues that can send and receive, opening up several possibilities on how to assign them to applications. One of the most promising options is the dedication of a hardware queue to an application. Additionally to the increased queue count, features like TCP offload, virtualization extension etc. were introduced to reduce the load onto the cores and improving isolation for applications using the network. Having such complex cards leads to a trend to remove the operating system from the datapath and giving applications more flexible ways to use the network.

As a basis for this thesis a high performance card, in particular the Solarflare SFN5122F dual-port 10GbE server adapter, is used to investigate the benefits regarding performance and isolation on the Barrelfish operating system. In this project, a driver for the Barrelfish operating system was developed. Currently there is a large effort around Barrelfish to make a step forward from traditional operating system structure that still have single queue NICs in mind and building an infrastructure to manage today's high performance network cards.

# Chapter 2

# Related Work

There is not much research into the direction of resource dedication, but still a lot of research is targeting the efficient use of hardware resources and features to not let them go to waste. Looking at what modern network cards are capable of today, having multiple hardware queues and also more functionality in these queues, software seems to be lacking somehow. There are a lot of different approaches to use hardware resources more efficiently. First of all, there are the user-level approaches to networking that may reduce latency tremendously. Further, ideas were introduced to offload functionality to the network card or the reverse strategy by onloading more onto the CPU. Additionally, a lot of concepts were presented in respect to how the operating systems needs to be built up, to efficiently implement network protocols.

## 2.1 User-level networking

### 2.1.1 Arsenic

Arsenic [13] is a Gigabit Ethernet NIC which exports an extended interface to both the operating system and application. The NIC implements functions that demultiplex packets (based on filters) and provides memory protection. Through transmit traffic shaping and scheduling mechanisms it is possible to control the bandwidth for each application. All the functionality provided by the NIC, enables applications to send and receive packets without the operating system's interaction.

### 2.1.2 Xok/ExOS's application-level networking

In the paper "Fast and Flexible Application-Level Networking on Exokernel Systems" [5] Ganger et al. describe the Xok/ExOS's exokernel [4] system and their approach to networking. The main idea is to reduce the abstraction i.e. FreeBSD sockets and give more flexibility to the application to specialize how the network is used. As a result, applications have an increased inclusion in the process of sending and receiving data.

### 2.1.3 The Packet Filter

Having networking protocols implemented in user space can lead to an overhead compared to a kernel implementation. Mogul, Rashid and Acetta [12] believe that the key to a good performance is the mechanism used to demultiplex packets to user applications. In their early paper they describe the Packet Filter, a packet demultiplexer in the kernel with the aim to reduce system calls and context switches and improve the performance using a user-level implementation of networking protocols.

## 2.2 Offloading

The more complex NICs get, the more functionality can be offloaded from the CPU to the NICs. Even the most rudimentary cards support offloading features like verifying checksums. One of the most controversial topics in regard to offloading is a TCP offload engine [11], short TOE. In order to reduce CPU utilization, the TCP/IP network stack processing is moved to the network card. It is called a full TCP offload if also the connection management is maintained by the TOE.

### 2.2.1 NICOS

As far as offloading goes there are actually attempts to modify or replace the operating system of a programmable high-end NIC [22], so that the resources given by the NIC can be used directly by an application. In this manner, applications can directly give tasks to the NIC and use the onboard CPU to increase performance. As a part of NICOS Weisenberg et al. implemented a scheduler that can preempt tasks running on the NIC.

## 2.3 Operating system architecture

### 2.3.1 x-Kernel

The x-Kernel [6] is an operating system kernel tailored towards efficient construction and composition of network protocols. It defines three primitive communication objects: protocols, sessions and messages. Each of the protocol objects describes a network protocol e.g. TCP, UDP. The relationship between protocols is defined when the kernel is configured. Session objects contain an instance of a protocol object and other data structures that represent the state of a network connection. Messages represent the protocol headers and the user data that is required by the session and protocol objects.

### 2.3.2 Dragonet

As a part of Barrelfish, Dragonet [17] is developed. Having modern NICs in mind, Dragonet is trying to allocate and dedicate hardware resources based on a Physical Resource Graph (PRG). The PRG lets the operating system reason about abilities of the card. The second critical part of Dragonet is the Logical Protocol Graph (LPG). Similar to the x-Kernel [6], the network stack is seen as a graph of protocol operations. Using the PRG and LPG, Barrelfish can get the global view of the systems state that is missed in other operating systems.

# Chapter 3

# Background

## 3.1 Barrelfish

Barrelfish [2] is a research operating system built from scratch to explore how the structure of operating systems need to be changed to efficiently use multi- and many core systems. It is based on a multikernel approach. Each core runs a so called CPU driver (or kernel) in privileged-mode and an user-mode monitor process. Because Barrelfish is based on a shared nothing architecture the different CPU drivers do not share any state. Communication is based on message passing and thus the system itself can be seen as a network of cores. Through the multikernel architecture, Barrelfish can support heterogeneous systems.

### 3.1.1 Capabilities

Following the microkernel [10] philosophy and thus reducing the functionality of the kernel to a minimum, Barrelfish uses a capability [18] approach for managing memory and system resources. In other words, not only does user space allocate and manage memory for their own objects, but for the kernel as well. Operations on capabilities are carried out by a system call to the CPU driver.

### 3.1.2 Inter-dispatcher communication

Similar to L4 kernel's IPC [9], Barrelfish uses inter-dispatcher communication or short IDC [1] to implement communication between different services and applications. IDC is based on message passing. The message format can

be defined and thus the number and type of arguments can vary. The possibility to transfer capabilities enables the sharing of memory regions. Various forms of communication are realized by using different back ends. The main criteria for choosing the back end is based on the cores that the services or applications are running on. If they are running on the same core (core-local communication), the local message passing (LMP) back end is used. Communication between dispatchers (kernel threads) on different cores is handled by the user-level message passing (UMP), which is based on sharing memory regions mainly in user space. In order to establish a connection that the two dispatchers can communicate, a process called binding is required. Binding basically assigns an object to each of the endpoints of the communication channel. For the binding itself, objects called interface references (or iref) are obtained by using a name service query so each service can be identified. As a result of the binding process an object is returned that represents the interface exposed by the service. If one of these functions of the interface is called, the opposite endpoint responds to the call based on which function was used.

### 3.1.3 Network architecture

As in exokernel [4] or microkernel systems, Barrelfish has most of the network functionality provided by an user space networking library (lwIP [3] for Barrelfish). Building on the services provided by lwIP, netd (networking daemon)[14] is responsible for running DHCP, thus getting an IP address for the NIC. Netd also handles ARP lookups and the traffic which no other application is responsible for. For each network device an instance of the device manager is running. The device manager is responsible for the port management, which includes software filters and if supported hardware filters. Additionally, the driver starts for each hardware queue a queue manager by calling its main function (`ethersrv_init()`).

#### LwIP

LwIP is a lightweight implementation of the TCP/IP stack and is currently used by Barrelfish. LwIP not only supports TCP and UDP but also a bunch of other protocols like DHCP, DNS, ARP and ICMP. The client side supports the well known BSD socket interface and two lower level APIs for integrating applications more with the TCP/IP code. To receive/send packets the application and netd connect to lwIP. Internally lwIP uses pbufs to represent the memory used for receiving and sending. Normally the pbufs would

have a size of 1600 bytes but due to the current implementation the buffers have a size of 2 KB. The service that provides the functionality, is running in user space and thereby in combination with Barrelfish a good basis for enabling an easy implementation of user-level networking when compared to Linux/FreeBSD. As an example, for the SFN5122F user-level networking is implemented through OpenOnload [19]. To provide a working solution for Linux they implement the network protocol stack themselves [15] and run it in user space.

### Device manager

For each NIC an instance of the device manager is running. The device manager is responsible for managing application access to ports, software and hardware filtering (if supported by the card). The type of filtering i.e. hardware or software can be chosen through an argument given to the device manager. If more than one hardware queue is allocated, the device manager has an argument called "totalqueues" that is required. The important parts of the interface for software filters are described in listing 3.1.

There are two types of filters: ARP and port filters. Currently hardware filters use a similar interface format but exposed by the card. To implement hardware filtering two parts are needed: a hardware specific part in the driver and a part that communicates from the device manager to the driver.

```
interface net_soft_filters
"Software based filter Interface" {
    call register_filter_request(uint64 id,
            uint64 len_rx,
            uint64 len_tx,
            uint64 buffer_id_rx,
            uint64 buffer_id_tx,
            uint64 filter_type,
            uint64 paused);
    response register_filter_response(uint64 id,
            errval err,
            uint64 filter_id,
            uint64 buffer_id_rx,
            uint64 buffer_id_tx,
            uint64 filter_type);

    call register_arp_filter_request(uint64 id,
            uint64 len_rx,
            uint64 len_tx);
    response register_arp_filter_response(uint64 id,
            errval err);

    call deregister_filter_request(uint64 filter_id);
    response deregister_filter_response(errval err,
            uint64 filter_id);
    ...
}
```

Listing 3.1: Interface exposed by device manager

### Netd

Netd is normally started at boot time. It is responsible for the setup and maintenance for networking. There is only a single instance of the netd service. Netd connects to all device managers and their shared queues and runs DHCP on them. Furthermore, for each NIC an ARP cache is maintained to run ARP lookups if requested by an application. If there is no application responsible for a packet, it is handled by netd. Netd takes various arguments. At least the card name is needed for which it is responsible for. Optional arguments are there to assigning a static IP to the card, define a gateway,

set the network mask and if DHCP should be run or not.

### Queue manager

A queue manager is responsible for managing a single hardware queue, thus for each hardware queue a queue manager is running. A part of a queue manager is the function `ethersrv_init()`. Listing 3.2 provides an overview of `ethersrv_init()` and the functions a driver needs to implement.

```
void ethersrv_init(
  char *service_name, uint64_t queueid,
  ether_get_mac_address_t get_mac_ptr,
  ether_terminate_queue terminate_queue_ptr,
  ether_transmit_pbuf_list_t transmit_ptr,
  ether_get_tx_free_slots tx_free_slots_ptr,
  ether_handle_free_TX_slot handle_free_tx_slot_ptr,
  size_t rx_bufsz,
  ether_rx_register_buffer rx_register_buffer_ptr,
  ether_rx_get_free_slots rx_get_free_slots_ptr)

void (*ether_get_mac_address_t)(uint8_t *mac);
void (*ether_terminate_queue)(void);
errval_t (*ether_transmit_pbuf_list_t)(
struct driver_buffer *buffers,
size_t count,
void *opaque);
uint64_t (*ether_get_tx_free_slots)(void);
bool (*ether_handle_free_tx_slot)(void);
errval_t (*ether_rx_register_buffer)(
                                 uintptr_t paddr,
                                 void *vaddr,
                                 void *opaque);
uint64_t (*ether_rx_get_free_slots)(void);
```

Listing 3.2: `ethersrv_init()` function and signature for the function pointer

`Ethersrv_init()` is the main function of a queue manager. In the current implementation each queue needs to register the functions to provide the following functionality: get the MAC address, terminate a queue, transmit a buffer list, get the number of free transmit/receive slots, register a free receive buffer and a function to clean up the transmit queue of used descriptors.

Additionally, a queue manager exposes an interface that is used as a part of the datapath. Listing 3.3 only shows the part of the interface that is relevant to the datapath and the initialization.

```
interface net_queue_manager
"Ethernet hardware RX/TX queue manager" {
    call register_buffer(cap buf,
                         cap sp,
                         uint64 queueid,
                         uint64 slots,
                         uint8 role);
    response new_buffer_id(errval err,
                           uint64 queueid,
                           uint64 idx);

    call raw_add_buffer(uint64 offset,
                        uint64 length,
                        uint64 more);
    response raw_xmit_done(uint64 offset,
                           uint64 length);

    call terminate_queue();
    ...
};
```

Listing 3.3: Queue manager interface

### Initialization and datapath

Let us have a closer look at how the initialization is performed. As part of netd's main function it starts an instance of lwIP and the ARP lookup service. When lwIP is initialized, it connects to the ARP lookup service from netd and sets up the connection. Likewise, it initialize all the services lwIP provides and allocates a big chunk of memory that internally is split up into pbufs. The memory is allocated for both receiving and sending. Besides these things, there is a more Barrelfish specific part. LwIP calls `net_if_init()` of the raw interface that indirectly connects to the driver through the queue manager. Performing an IDC call (`register_buffer()`), the memory allocated for receiving and sending is registered to the queue manager, yielding back a queue ID and a buffer ID (`new_buffer_id()`). For each empty

buffer lwIP calls `buffer_rx_add()` of the raw interface. Then an IDC call (`raw_add_buffer()`) to the queue manager is executed, which in turn adds these buffers as descriptors into the receive queue managed by the queue manager. The function to add descriptors to the receive queue is part of the information given to the queue manager by the function `ethersrv_init()`. At this stage, all the buffers that are needed at the beginning are defined.

#### Receiving datapath

If the card receives a packet, the following information is known: the offset into the memory chunk (allocated by lwIP), the length of the data and if the packet is broken up into more than one buffer. The driver hands over the information through a function call (`process_received_packet()`) to the queue manager. The queue manager on his part does some housekeeping and propagates the length and an offset to the raw interface. The raw interface computes the pbuf ID based on the offset and calls `handle_incoming()`. According to the information given to lwIP, it processes the buffer. Each used buffer is replaced by a new free buffer and is added in the same way as the receive queue is populated at the beginning i.e. an IDC call `raw_add_buffer()`.

#### Transmitting datapath

Whenever a packet must be sent, lwIP propagates the information required to the raw interface (`buffer_tx_add()`) and from the raw interface to the queue manager using an IDC call (`raw_add_buffer()`). From lwIP the following information is known: the ID of the pbuf used for the data to be sent, the offset into the pbuf and the length of the data sent. The queue manager converts the information to a data struct called `driver_buffer` and updates the state of the buffers. The `driver_buffer` stores the length, the physical address and the virtual address of the buffer. The struct is then handed over to the function `transmit_pbuf_list()` that was registered calling `ethersrv_init()`. When the driver signals that a transmission is completed, it calls the function `handle_tx_done()` from the queue manager, which does the housekeeping. The queue manager performs an IDC call (`raw_xmit_done()`) to the raw interface which in turn then calls `handle_tx_done()` from lwIP.

### 3.1.4 Dedicating queues

Before an application can use the network in Barrelfish, it should be initialized. The initialization is handled by either calling `lwip_init(char`

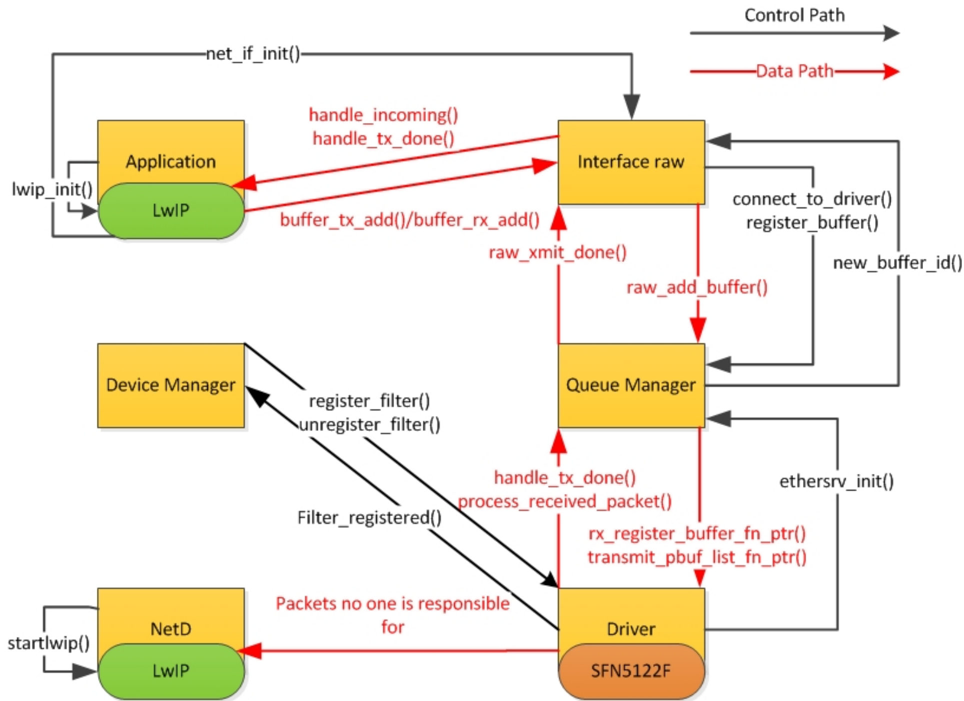


Figure 3.1: Overview of datapath and initialization

`*card_name, uint64_t qid)` or `lwip_init_auto()` from lwIP. Since the arguments to the function call are a card name and a queue ID, it is transparent to the user which hardware resources are used. In the case of `lwip_init_auto()` the card name is guessed automatically and the queue ID is set to zero. A queue ID of zero is assumed a safe option, but may have drawbacks because it is a shared queue. In this way, a queue is bound to an application. There is currently no other way in Barrelfish than to dedicate a queue to an application.

What are the reasons for dedicating queues to applications? First of all improved performance. By dedicating a queue to an application, one can also imagine having customization options for the network stack per queue. Additionally to the performance benefits, a better quality of service for applications is expected.

## 3.2 Mackerel

Mackerel [16] is a hardware description language for devices that use memory mapped registers. Even data structures like descriptors or tables can be described. Mackerel is designed for an easy translation of hardware data books or programmers reference manuals etc.. The description in Mackerel is compiled to an output C header file that contains a large number of C inline functions which can be called to manipulate the content of a register. In order to use these functions, the device must be initialized with a base address (in the case of a NIC from a PCI base address registers or short PCI BAR).

## 3.3 Network Controller

This section provides an overview of the Solarflare SFN5122F network adapter [20] and some of the features it provides. The SFN5122F is part of the SFx902x controller family, which supports 10 Gigabit Ethernet, PCI express 2.0, one PCI function per port which can be increased if virtualization is enabled. The communication from the driver to the card is based on memory mapped registers. The registers are written to configure the card and access the tables where the state of hardware queues and filters are stored.

### 3.3.1 Buffer Address Translation

The SFN5122F implements a buffer address translation, which allows the safe use of virtual NICs in combination with untrusted software. The basics behind the address translation is to match attributes that identify a buffer uniquely and as a result of the translation, output the physical address of this buffer. The card uses IDs to identify the buffer and its owner. This prevents applications from using buffers they do not have the permission to access, are not allocated yet or are out of range. The address translation is used for user-level networking, for conventional kernel mode drivers it is not necessary.

### 3.3.2 Multiple Queues

The network controller supports up to 1024 hardware queues. Each hardware queue consists of a receive queue, a transmit queue and an event queue. Analogous to other modern network cards the queues use descriptor rings (event entry rings in the case of event queues) with a size of up to 4096

descriptors or 32768 event entries. The format and size are dependant on the mode the queues operate in and the type of the queue. There are two modes: physical mode and buffer mode.

### Receive Queues

For receive queues in physical mode the descriptors have a size of 8 bytes and contain the information about the size of the received data, the physical address and the address region. When updating a queue tail there is the possibility to push an additional descriptor with the tail update. In physical mode the receive queue can be set into header split mode. If header split mode is enabled, the packets are received by two queues called header queue and payload queue. The header queue only receives the header of the packet, while the payload queue receives the data of the packet. In buffer mode the descriptor size is reduced to 4 bytes and the information stored is an offset from the base address of the buffer and a buffer ID. The rest of the information to process the descriptors is encoded in RX events (see Event Queues).

### Transmit Queues

Transmit queues in physical mode have a descriptor size of 8 bytes. The descriptors contain the information about the physical address, the address region, the length and a bit that indicates if a packet is split over several descriptors. More than 16 descriptors per packet is considered a software error. As receive queues, transmit queues have the same possibility to push a descriptor with the tail update. In buffer mode the size of the descriptor is reduced to 6 byte. The descriptors contain information about the length, the buffer ID, offset into the buffer and the same bit that indicates continuation of the packet as in physical mode. Alike to the receive descriptors more information is encoded in TX events (see Event Queues). In both modes transmit pacing can be enabled to limit the pace at which a queue sends packets.

### Event Queues

Event queues are circular ring buffer data structures similar to receive/transmit queues. The Solarflare card has 1024 event queues. Events report status asynchronously to the driver and are encoded into a 64 bit `EVENT_ENTRY`. Sizes of up to 32768 entries are supported. There are six different possible events:

| Event | Code | Description |
|---|---|---|
| RX | 0 | Event when a packet is received. Has enough information to free processed descriptors and may signal if a packet has an error |
| TX | 2 | Event when a packet is received. Has enough information to free processed descriptors and may signal an error when sending a packet |
| Driver | 5 | Event if e.g. the queues are flushed, SRAM updates are done, timer events etc. |
| Global | 6 | Basically only physical layer (PHY) events |
| Driver Generated | 7 | Event that is generated from the driver |
| MCDI | 12 | Event when the link changes, the card is rebooted etc. |

RX and TX events encode more information about receiving and sending than there is in the descriptors itself. As an example, the descriptors do not have any information about their state i.e. if they are already processed or not. Interrupts for an event queue can be handled in two ways. Either the interrupt occurs immediately or it can be hold for an user defined time.

**Filters**

Filter for both sending and receiving are implemented in hardware. The filters are kept track of using four filter tables. There are 512 entries for MAC based filters and 8192 for IP based filters. The filters provide demultiplexing for the queues on the following criteria:

- **Ethernet filtering**
  - Ethernet frame filtering based on the MAC destination address and VLAN ID
  - Ethernet frame filtering based on the MAC destination address
- **IPv4 filtering**
  - TCP wildcard match using destination IPv4 address and port number
  - TCP full match using source and destination IPv4 addresses and port number

- UDP wildcard match using destination IPv4 address
- UDP full match using source and destination IPv4 addresses and port number

The depth to which the filter tables are looked up can be configured for both wildcard and full matches.

#### Receive side scaling (RSS)

If receive side scaling is enabled the data received can be spread over multiple queues. Each queue can then be processed by a different CPU core. To not disturb network flows, hashing is performed to always target the same queue. The card takes a 32 byte hash that may be used in combination with different hashing methods.

### 3.3.3 Virtualization (SR-IOV)

The NIC features the possibility to enable virtualization and use up to 127 virtual functions per physical function. Virtual NICs can be bound to virtual functions, therefore allowing a partition of resources. While physical functions provide an interface to control the whole card, virtual functions provide an interface to control only a part of a virtual NIC. Virtual functions only support MSI-X whereas physical functions support all three interrupt modes.

### 3.3.4 Interrupts

There are different sources that can raise an interrupt. First of all, event queues can cause an interrupt if new events are written into memory. Next, the controller itself can report a fatal interrupt which signals that an error like memory parity error or buffer translation error occurred. The interrupts can be raised in different modes. On one hand, we have the PCI-legacy mode that maps the two physical functions to IntA and IntB of the PCI express bus. A limitation of legacy interrupts is that not all queues generate interrupts. On the other hand, we have MSI and MSI-X which are message based. To support MSI-X an additional PCI BAR needs to be mapped to access the MSI-X vector table. Each queue has one entry in this table and can also raise interrupts.

### 3.3.5 Management-Controller-to-Driver Interface (MCDI)

The Solarflare card implements a Management-Controller-to-Driver Interface short MCDI, which is a simple remote procedure call protocol. The protocol is implemented using a shared memory region called `MC_TREG_SMEM`. There is a wide variety of commands the protocol can execute e.g. setting the MAC settings, setting the link settings, let the card DMA MAC statistics into host memory. When mapping the card into the virtual space of the driver, the registers mapped do not contain the registers that are required to configure the physical layer (PHY) and the media access control (MAC). Consequently it is mandatory to implement this protocol to get a working driver. The protocol works the following way:

1. The request to execute a command starts with the MCDI header, which is a 32 bit structure with a format according to:

| Bits | Name | Description |
|---|---|---|
| 6:0 | Code | The code for the command to be executed |
| 7:7 | Resync | The resync bit is always set |
| 15:8 | Len | The length of the input buffer used for arguments to the command |
| 19:16 | Seq | Sequence number |
| 21:19 | Rsvd | Reserved bits |
| 22:22 | Err | Set if an error occurred while executing |
| 23:23 | Response | Set if the command is finished executing |
| 31:24 | Xflags | Set if the response should be through an event |

2. Write the command and the input buffer into the shared memory region according to an offset depending on the port number.

3. Write a distinct value into the shared memory region according to an offset depending on the port number. This is called "Ringing the Doorbell".

4. Wait until the response bit is set, but caution needs to be taken if the memory of the card is reset. In this case, the response bit is set but the command is not yet executed.

5. Extract from the MCDI header if an error occurred and the length of the output.

6. Retrieve the output of the RPC call.

### 3.3.6 Offloading

To reduce the CPU cycles utilized for packet processing, the controller implements three packet checking functions in hardware:

- IPv4 header checksum
- TCP/UDP checksum
- iSCSI header and data digest

The results of the hardware functions are encapsulated in the RX event that is generated on reception. If a received packet failed one of the verification functions, a bit is set that indicates an error. To narrow down the cause of the failure various other bits indicate which of the hardware verification functions failed.

### 3.3.7 User-level networking

By bypassing the kernel, user-level networking can reduce the latency by a great margin. To guarantee safety of accesses to memory, the card itself is required to implement similar features for memory protection as a kernel. The basic feature the card implements to guarantee safe memory access, is a possibility to translate something similar to virtual addresses to physical address while not violating access permissions. The translation should prevent malicious software from reading/writing memory locations that are either not allocated or the software has no permission to access. The Solarflare card implements this property with the buffer address translation. Additionally, the translation also introduces a form of isolation for applications. For queues allocated in buffer mode i.e. used for user-level networking, it is mandatory to assign it a number to identify the owner of the queue and as well define the buffers that the queue may used to receive and transmit packets.

# Chapter 4

# Approach

## 4.1 Driver

Driver implementations are always a special thing. The hardware can not be controlled directly (just through registers) and the response to certain configurations may be unpredictable. Even a single bit that is set wrong, can lead to the driver not working correctly. The huge configuration space does not make it easier to write a driver for a network card. Furthermore, even having the documentation for the card, may not be enough for implementing a functional driver. There always might be some missing parts or mistakes in the documentation. One of the positive aspect that is helpful when writing a driver is, when the source code for another operating system is available. For the Solarflare card there are two existing drivers for Linux and FreeBSD for which the source code is available.

### 4.1.1 I/O

At first, we had the intention to port the BSD-licensed HAL (hardware abstraction layer) library from the FreeBSD driver to Barrelfish to get the I/O to the card working. Browsing through the code of the library, it looked a bit overwhelming for the first task of this project. Having Barrelfish as an operating system, the implementation of the I/O to the card could be realized with Mackerel. Looking through the documentation of Mackerel, it seemed more natural to implement the I/O in this manner and it is more the "Barrelfish style" to do it. So the first actual task was to translate the documentation which described the hardware registers into Mackerel. To confirm that the Mackerel bindings were correct, it is the easiest to read a constant register. For writing registers, the best way to verify is to first write

the register and then read it. After knowing that the Mackerel bindings are working, the next step was to implement a single queue driver.

### 4.1.2 Single queue driver

The simplest driver for a network card is one that only uses a single queue and legacy interrupts. There are several steps on the way to a functional single queue driver. The first step is to initialize the hardware to a known state. Normally when writing a simple driver a good part of the code is dealing with the initialization of the device. Since this part is one of the most error prone, it is a source for a lot of debugging hours. Part of the initialization is reading the MAC address that is necessary as a part to register the card to the operating system. When the card is configured, it can only be tested by setting up hardware queues. After that, the different functions for processing a queue may be registered to the operating system. The card should now be ready to receive and send some packets. At first, assuming the receive queue is working, a lot of packets should be received that are flying on the wire. To verify that transmitting is working tools like Tcpdump [21] and Wireshark [23] come in handy.

### 4.1.3 Multiple queues and filters

The main decisions to make, for converting the single queue driver into a multi-queue driver, is how to get the event loops running that check for new packets. For each queue, a queue manager must be started through the `ethersrv_init()` function. Likewise, for demultiplexing packets sent/received to different queues, hardware filtering and the communication to the device manager responsible for filtering need to be implemented. We decided to implement a driver similar to the Intel 82599 [7] from Barrelfish [8] and splitting up the driver into two parts: a card driver and a queue driver.

### 4.1.4 User-level networking

User-level networking has the advantage of a certain isolation from other applications and not to mention the performance benefits. Hence, the implementation of user-level networking with the SFN5122F was the next step to take. On the hardware side there are only two parts to change. The memory that is used for the receiving/sending needs to be predefined before using the queues in buffer mode. Therefore, the card requires a big chunk of memory from somewhere that can be added as 4 KB buffers to the buffer table of the card. The other part is a way to identify for which queue and

for which application the memory is allocated, so safety of memory access through the buffer address translation can be guaranteed.

On the side of the operating system, there is more to think about. The main effort to enable user-level networking is in removing the queue manager from the datapath. This implies that the functions that are registered with `ethersrv_init()` are no longer on the datapath. Basically, we want lwIP to almost directly add descriptors to the receive/transmit queues. To let lwIP communicate with the rest of the networking code, there is currently a raw interface. This interface connects indirectly to the driver through the queue manager. In the end, we need to remove or avoid this connection and use a direct path to the driver using an IDC interface. The information we get from lwIP are a pbuf ID, an offset into the pbuf and a length for the sending side. For the receiving side there is only a pbuf ID. Having two different structures at hand, we need a way to translate the pbuf IDs to buffer table IDs of the card. These are the changes that are required to remove the queue manager from the datapath from lwIP to the card, but there is still the other direction. When user-level networking is not enabled, the driver responds to the queue manager when a packet is sent or received. Eventually we need to remove the call to the queue manager and do it more directly.

# Chapter 5

# Implementation

This chapter presents the implementation of the driver, the changes done to the Barrelfish network stack to enable user-level networking.

## 5.1 Driver implementation

The basis for the implementation of the driver is an internal document from Solarflare that is under a non-disclosure agreement. Additionally, two drivers (Linux and FreeBSD) and their source code are available. With the two drivers, we were able to fill in the missing information from the internal documentation.

### 5.1.1 Single queue

The first problem we encountered on the way to a single queue driver was with Mackerel. Mackerel does not support 128 bit registers, so we split up the registers into two 64 bit registers. Sometimes, a dummy register was added even though there were no bits that could have been written. For an explanation to this, see section 5.2.1. The documentation seemed to miss some parts, so we were left with the two drivers for which the source code was available. Looking through the drivers we first noticed the MCDI requests to the card. Using the drivers as a reference, we were able to implement a working version of the protocol. The same approach was taken for the initialization of the card. One of the differences to traditional cards is, that the SFN5122F has event queues. The event queues change the way the driver is checking for new packets. With the Solarflare card, the interaction is event based. When initializing event queues, all of the bits making up the queue

are set to one (all zeros is a valid event). In the current implementation the driver is polling for new events. Each event has a code and for every event code a function is implemented that handles a specific event type. As an example for cleaning up the descriptor queue when a packet is sent, there is a function that is registered to the queue manager via `ethsrv_init()`. When we implemented the function for the Solarflare card it can only return false i.e. there is nothing to clean up, because the driver is actually waiting for an TX event that indicates when a descriptor is done and the packet is sent. When implementing the single queue driver, we lost a lot of time debugging, and ensuring that the packets were sent/received correctly (see section 5.2.3).

### 5.1.2 Multiple queues and filters

For the implementation of a driver that makes use of multiple queues, we took the current Intel 82599 [7] [8] as a reference. Basically, the driver is split up into two parts.

#### Card driver

One part is the card driver that is responsible for setting up the I/O and initializing the card and hardware queues. The card driver has all the information about the queues i.e. memory location, size etc. and about all the filters currently enabled. It exposes an interface to which every queue driver connects when starting. Furthermore, the interface exposes functions to register port filters. The card actually has four tables for hardware filters: MAC filters and IPv4 filters for both sending and receiving. In the current implementation there are only filters for ports or in other terms "wildcard" filters (no IP address required) for the receiving side. The filters are implemented by an IDC call from the device manager to the card driver containing the information to fill in a table entry. The MAC filter table could be used to implement ARP filters, but we thought it makes sense to leave it at port filters for the beginning.

#### Queue driver

The queue driver has all the functions implemented to manage a queue on the hardware level. A queue driver is static, hence either it is defined that it should be started at the beginning or it is not started at all. It connects to the interface exposed by the card driver. Using the interface it performs an IDC call to the card driver to get the capability of the registers, thus making

it possible for the queue driver to write the registers. The queue driver then allocates a receive, transmit and event queue and registers the capabilities of the queues to the card driver where the queues get initialized in hardware. After the queues are initialized in hardware, a queue manager for the queue is started by calling `ethersrv_init()`. At this stage, the queue driver is polling for events and handling events according to their type. A queue driver can only be terminated by the card driver.

### 5.1.3 User-level networking

User-level networking is currently not working, but most of the changes described here are implemented. On the driver side there is not that much to change to enable user-level networking. User-level networking can be enabled by an argument. If this argument is set, the queues are allocated with the reduced size (smaller descriptors) in buffer mode. To start up the queues in buffer mode there is only a single bit to change and the owner ID must have a non zero value. Furthermore, the functions registered to the queue manager are mainly dummy functions.

Getting the memory needed for user-level networking is our next concern. When lwIP registers the big memory chunk used for either sending or receiving to the queue manager, it seemed the easiest to propagate this information further to the driver. We decided to add a function pointer to `ethersrv_init()` that adds entries to the buffer table having the owner ID set to the queue ID plus one. As a return value of this function the queue manager gets the ID of the first buffer table entry making up the memory used. The queue manager then propagates this offset into the buffer table to the raw interface. The 2 KB pbufs of lwIP can now be directly translated to 4 KB buffer table entries of the card. To translate a pbuf ID to a buffer table ID, the pbuf ID is divide by two, round up and the initial offset is added. The offset into the pbuf is the same for the buffer table entry, except when the pbuf ID is odd. Then 2048 is added for the offset into the buffer table entry.

To remove the queue manager from the datapath, the raw interface needs to add descriptors directly into the receive/transmit queues. Either we could add descriptors by IDC calls or the queue could be set up in the raw interface itself. We decided on the second approach to remove the latency added by IDC. We added a function to the IDC interface of the card (`get_queue()`) that sends the capabilities of the transmit/receive queues and the registers of

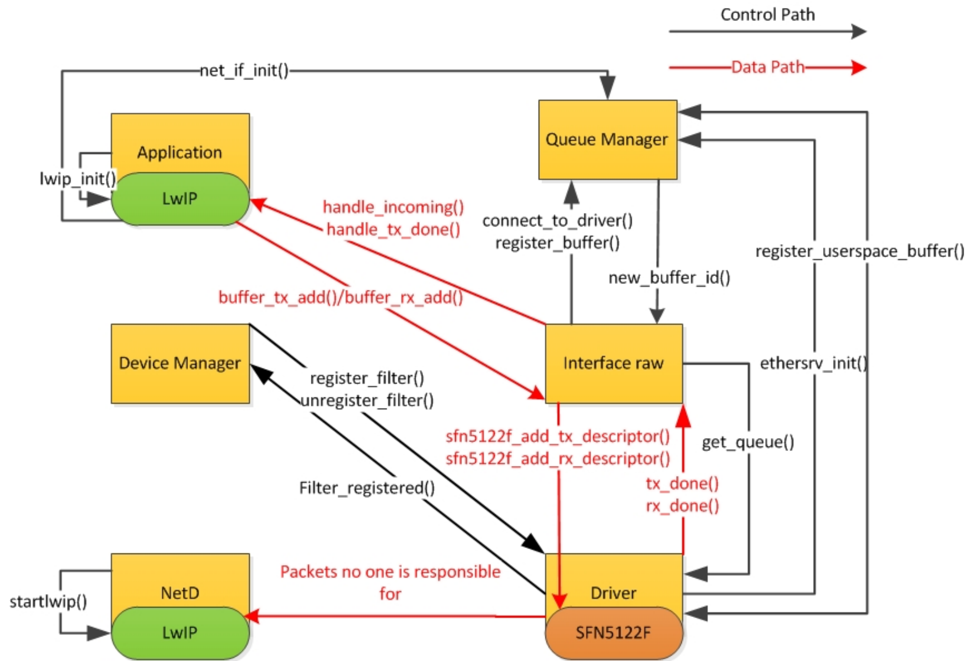


Figure 5.1: Overview of revised implementation

the card. With the virtual address of the base of the queue, descriptors can be added directly from the raw interface (`sfn5122f_add_tx_descriptor()`). After the descriptor is added, the queue tail is written to notify the card that there is work to do. For the other direction lwIP is informed from the driver that a packet is sent/received by translating a buffer ID to a pbuf ID and using the functions from the raw interface (`tx_done()/rx_done()`).

## 5.2 Problems during implementation

This section presents the most important problems faced during implementation and their attempted solution.

### 5.2.1 Writing registers of the card

When we used Mackerel for describing the hardware, there were some problems because Mackerel does not support 128 bit registers. The easy solution was to split up the register into two 64 bit registers. The splitting of registers led to another problem when writing only the first or the second half

of the 128 bit register. Reading the value from the register after a write, it seemed like the write did not happen at all. Due to a hardware specific implementation that first accumulates the writes before they are written down to the register, the value of the register stays unchanged. As a result of the accumulation, some of the registers are only written (with no difference) just to fulfill the constraint that all parts of the register are written.

### 5.2.2 Reading registers of the card

In some cases, reads may return a value that is out of date. The SFx902x controller family performs reads using a shadow register, which works like a single entry cache. On a read of the first 64 bits, the current value is fetched and placed in the shadow register. All subsequent reads take the value from the shadow register. Only after the whole register was read, the subsequent reads return the current value of the register.

### 5.2.3 Packet transmission

Shortly after getting the hardware queues up and running, we tried to verify that the packets are sent and received correctly. The receiving side looked fine, broadcasted packets were received correctly (and some other packets that needed to be discarded). Monitoring the traffic using tools like Tcpdump [21] and Whireshark [23] there was no indication of any packets that were sent by the card. Additionally, a flag in the TX event was set indicating that the transmission should be completed. The first thing in mind was to further compare our implementation with the Linux driver if something was missing.

After finding small differences, which did not lead to the solution of the problem, we compiled the Linux driver ourselves. By adding debugging statements we could get further insight. On one of the development servers we tried to compile the latest version of the driver without success. Looking at the kernel version of the server, we concluded that it may be easier to compile an older version of the driver. To get as near as possible to the version running on the servers, we chose the version from the Linux kernel 2.6.33. After compilation without any problems, the server crashed when the driver was inserted as a kernel module. The server crashed because the traffic to maintain the connection to the server was by default running on one of the card's interfaces. When inserting the module, the connection to the server failed and thus the only way to proceed was to restart the server.

After some brainstorming we tried to first give the same IP to another port of another card and then taking the two ports of the Solarflare card down. This was the solution to a running Linux driver. After getting the driver to work, we printed out all registers that are written during the initialization. Comparing the values, there were no major mismatches that could have prevented the card from sending packets. Looking at the MCDI requests, there seemed to be no difference at all in terms of the input buffers. To further assimilate the initial situation, we forced the driver to work with legacy interrupts. There was almost no change of the values that were written to the registers or the driver's behavior. In turn to reduce debugging efforts, we connected two servers directly. If anything was on the wire, we would see it now without any other traffic. To further rule out possible sources of an error, we inspected the different flags that had something to do with the MAC or the link. We obtained the flags by a MCDI request we added to the Linux driver. There was no difference at all, even though some of these flags seemed strange. In conclusion after looking over the cards configuration everything seemed fine.

We looked more carefully at the values written into the descriptor rings. The descriptors itself seemed all right i.e. the continuation flag was not set and the physical address and the size seemed reasonable. We were not sure about the physical address. In the Linux driver there is a comment that indicates that buffers need to be aligned to 4 KB boundaries. To embrace all the possibilities that the physical address may be a source for the errors, it seemed reasonable to allocate the buffers ourselves. We copied the data into the buffers and then gave this physical address to the card. To give a changing physical address to the card, the buffers were allocated similar to the descriptor queues as a ring buffer. Having noticed no difference, we could eliminate the physical address as a source for an error.

Knowing that the driver sends a DHCP request at the beginning and retries because there is no response, we tried to send more than a single packet at a time. We allocated a buffer and sent its content repeatedly. After trying to send other packet types like ICMP or ARP packets with no success, there seemed to be no problem with the contents or the quantity of the packets.

To avoid sending a DHCP request, we set the card's IP address statically. When starting Netd there is a possibility to give it an IP address, a network mask, a gateway and a flag if DHCP should be run or not. The first packet that is sent when not running DHCP, is a single ARP packet. It did not

yield any additional information to what could be wrong.

Having in mind that there was some sort of completion event for the sending side, we tried to get the MAC statistics up and running. From the MAC statistics we could infer, if the completion event is counted as a sent packet or if it is not counted at all. In order to get the statics, it required a MCDI request that takes a physical address pointing to a buffer, and some other arguments regarding the frequency of DMA of the statistics. Using the same arguments as for the Linux driver, we got an error from the MCDI request indicating that the arguments were invalid.

After a lot of debugging on other parts of our code, we had a closer look at the implementation of the MCDI protocol. Most of the time, we assumed it worked correctly because all of the calls succeeded except the one for the DMA statistics. Hence, we were looking for the error in the arguments to the call and not in the implementation of the MCDI protocol. Still there was an error that was unnoticed a long time and prevented writing out all arguments from the input buffer to the card. Fixing this bug solved most of the problems and the driver was finally working.

## 5.3 Unresolved Issues

Due to time being a constraint, there are still unresolved issues we could not approach regarding the driver itself. The driver is not tested thoroughly and more issues may arise.

### 5.3.1 Gottardo

When the card driver is started on gottardo, there may occur a page fault on the first line of code. We did not find the cause for this yet, but the driver seems to function correctly.

### 5.3.2 Filters

The hardware filters from the SFN5122F may require a destination IP address, but at the moment there is only the information available for port filters that do not require an IP address.

### 5.3.3 User-level networking

As mentioned before in section 5.1.3 the user-level networking part of the driver needs debugging. When a queue is started in buffer mode it seems like the event queues are not working correctly. There are some events that are reported at the beginning but after a certain time no events occur.

# Chapter 6

# Evaluation

Due to the time we have lost debugging the driver, there was only time left for an initial performance evaluation. The performance benefits of dedicating queues to applications would need more complex benchmarking. The complexity comes from the assumption that the effects of dedicating queues become more visible, when having multiple network flows in the system. In Barrelfish there is currently no other alternative than to dedicate a queue to an application, which leaves us with no option to compare to except for Linux.

## 6.1 Comparison with Linux

First, we compared the performance of the SFN5122F driver from Barrelfish to the performance of the Linux driver. The setup is as following: On gottardo a server is running that just echoes back the packets. On the other side, there is ziger2 running the client side sending packets of different sizes. We measured the time on the client side until the response is received. The figures below are the average over 1000 packets sent/received for different payload sizes. In comparison with Linux, the performance does not look that bad. For bigger payload sizes the latency increases more for Barrelfish than for Linux.

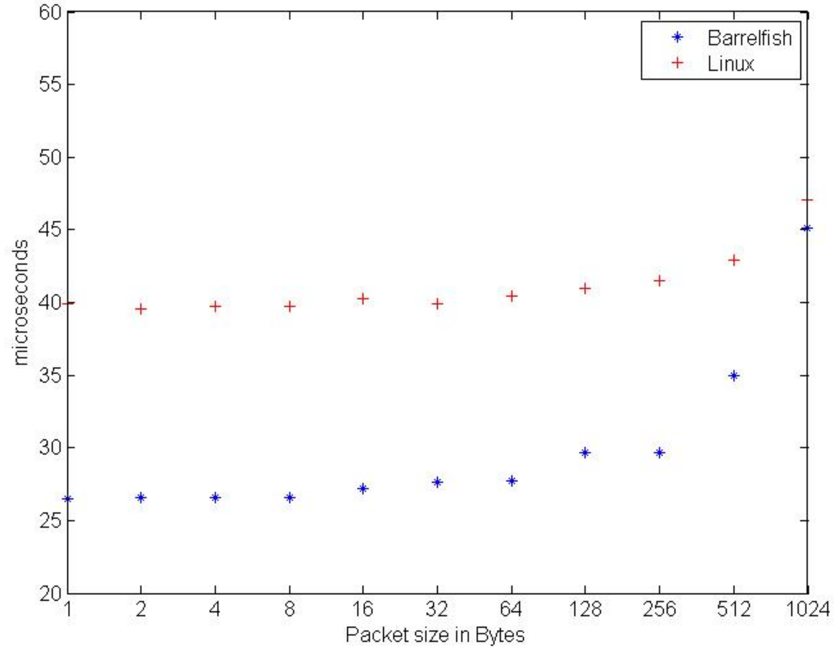


Figure 6.1: Comparison Barrelfish vs. Linux

## 6.2 OpenOnload

To give a performance hint of what we expect from our implementation of user-level networking, we did a second set of measurements using OpenOnload [19] from Solarflare. OpenOnload removes the Operating System from the datapath and implements a high performance network protocol stack in user space.

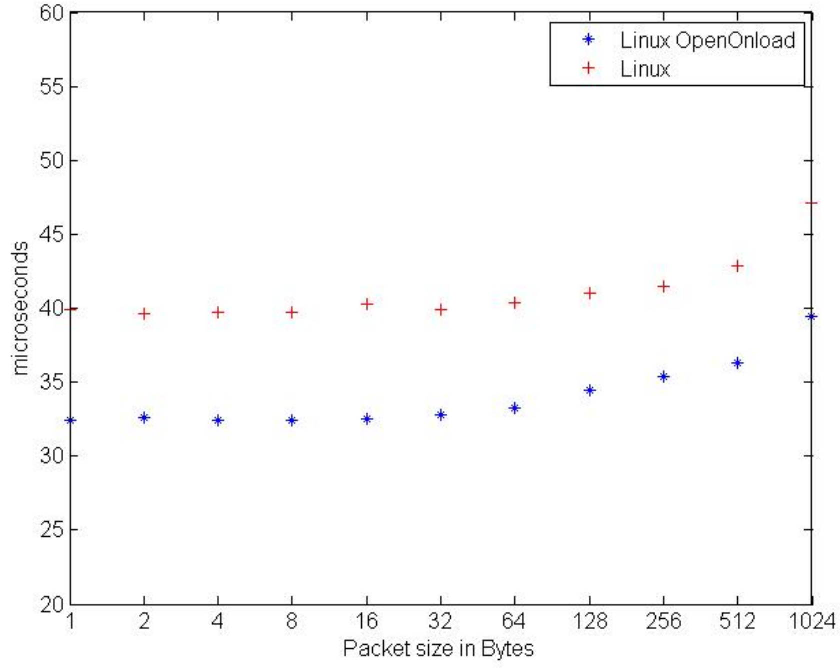


Figure 6.2: Comparison Linux vs. Linux using OpenOnload

With the usage of OpenOnload, the round trip time can be reduced by up to $8\mu s$. When converting the results from Linux to Barrelfish, we can expect machine to machine latency of less than $10\mu s$.

# Chapter 7

# Conclusion

Having spent most of the time debugging the driver (around 200 hours) there was not that much time left for anything other than a simple latency performance test. Nevertheless, I learned a lot of things during the implementation of the driver and obtained a good knowledge about the Solarflare card, but also about the Barrelfish infrastructure. Looking at the different ways to implement a network card driver from the different operating systems (Linux, FreeBSD and Barrelfish) was interesting. This showed some of the limitations of the networking architectures of the current operating systems that still have their network stack in kernel space.

## 7.1 Further work

There is still a lot of work regarding the driver. First of all, the part that handles the user-level networking needs debugging. This would be the most beneficial part for Barrelfish and it would also make the best use of the multikernel architecture of Barrelfish. The driver uses a similar architecture like the driver for the Intel 82599 [7] that is quiet static. Making this more dynamic that queues could be allocated on demand, would open up more possibilities to further investigate benefits of different ways to assign hardware queues, but also make the driver more practical. Furthermore, there are a lot of hardware features that could be implemented and made use of. An other direction that may be interesting is to introduce the possibility for a more configurable network stack based on needs of an application.

Pushing the limits of Mackerel and its compiler pointed out some of the shortcomings regarding modern high performance cards. Having tables as big as 140'000 leads to long compile times even when changing the definition of a single register. Additionally support for 128 bits would be desirable and also make sense regarding newer hardware.

# Bibliography


[1] A. Baumann. Inter-dispatcher communication in barrelfish. technical note 011, 2011.

[2] A. Baumann, P. Barham, P. E. Agand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpback, and A. Singhanda. The multikernel: a new os architecture for scalable multicore systems. In *SOSP*, volume 9, pages 20–44, 2009.

[3] A. Dunkels. Minimal tcp/ip implementation with proxy support. Master's thesis, Swedish Institute of Computer Science, 2001.

[4] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. pages 251–266, 1995.

[5] G. R. Ganger, D. R. Engler, M. Kaashoek, H. M. briceno, R. Hunt, and T. Pinckney. Fast and flexible applicaiton-level networking on exokernel systems. pages 49–83, 2002.

[6] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, pages 64–76, 1991.

[7] Intel. Intel 82599 10 gbe controller datasheet.

[8] A. Kaufmann. Low-latency os protocol stack analysis. Bachelor's thesis, ETH Zurich, 2012.

[9] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, 1995.

[10] J. Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996.

[11] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, page 5. USENIX Association, 2003.

[12] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *In proceedings of the eleventh ACM symposium on Operating Systems Principles*, pages 39–51, 1987.

[13] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *IN PROCEEDINGS OF IEEE INFOCOM*, pages 67–76, 2001.

[14] K. Razavi. Barrelfish networking architecture. Distributed systems lab, 2010.

[15] D. Riddoch and S. Pope. Openonload, a user-level network stack. Google Tech Talk, 2008. http://www.openonload.org/openonload-google-talk.pdf.

[16] T. Roscoe. Mackerel user guide. technical note 002, 2013.

[17] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We need to talk about nics. In *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*, pages 1–8. USENIX Association, 2013.

[18] A. Singhania and I. Kuz. Capability management in barrelfish. technical note 013, 2011.

[19] Solarflare. Openonload http://www.openonload.org/. visited 2013-08-20.

[20] Solarflare. SFN5122F Dual-Port 10GbE SFP+ Onload Server Adapter. http://solarflare.com/Content/userfiles/documents/Solarflare_Onload_SFN5122F_10GbE_Adapter_Brief.pdf. visited 2013-08-27.

[21] Tcpdump. http://www.tcpdump.org/. visited 2013-08-26.

[22] Y. Weinsberg, T. Anker, D. Dolev, and S. Kirkpatrick. On a nic's operating system, schedulers and high-performance networking applications. In *In HPCC-06*, 2006.

[23] Wireshark. http://www.wireshark.org/. visited 2013-08-26.