Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 36b

Systems Group, Department of Computer Science, ETH Zurich

Low-latency OS protocol stack analysis

by

Antoine Kaufmann

Supervised by

Prof. Timothy Roscoe, Pravin Shinde, Adrian Schüpbach

September 2011–January 2012

Informatik
Computer Science

Contents

Chapter 1

Introduction

Distributed applications have become increasingly popular during the last few years for a number of reasons. But today distributed systems also have one major drawback that limits scalability and sometimes even feasibility of performance critical tasks: communication overhead. A remote procedure call (RPC) is significantly more expensive than a local procedure call. Which means that mostly problems that allow batching of communication i.e. where multiple requests can be sent at once and can be performed without further communication, are implemented as distributed systems.

This has been the case since distributed systems were first developed, even though networking technology has advanced significantly since then. Bandwidth for example increased by more than 3 orders of magnitude over the last three decades. But often those problems that require frequent communication don't transport much data in a message, so that bandwidth is not the limiting factor, e.g. transporting 64 bytes over a 10Gbps link takes around 50ns so this results in 100ns per round trip, while a RPC round trip time of 100us (RTT) for an RPC of that size is realistic today. So the problem is high latency [20] which has been reduced little more than one order of magnitude over the last three decades.

While part of the problem has to be addressed by the hardware designers, the software architecture including device drivers, operating systems and applications has to be optimized and potentially redesigned to achieve lower latency, while most current implementations are focused on bandwidth.

In this thesis I will analyse the current Barrelfish network stack, identify sources of latency, and try to eliminate them (if possible). As part of this thesis a driver for the Intel 82599 10Gbps network controller was developed, that is able to benefit from the hardware features provided by the network controller that allow further optimization of latency. The primary goal is to reduce the latency of an RPC using TCP/IP.

Chapter 2

Related Work

Most of today's network stack implementations focus on achieving maximal throughput. Rumble et al. [20] advocate improving latency and provides motivating examples of systems that are limited by latency.

Latency is directly affected by the CPU load, if the CPU is occupied it will take longer to react to events. One popular method to reduce the CPU load caused by networking is offloading. The idea is to offload parts of the network stack onto the network interface controller. Examples of offloaded features are TCP segmentation in the transmitter and receiver-side coalescing of TCP segments, or checksumming (both IP and TCP/UDP). J.C. Mogul [12] looks at offloading and the situations in which it is useful and can actually enhance performance, and at the problems it presents.

The opposite of offloading is onloading [18]. In contrast to offloading the network packet processing is not offloaded onto the network interface controller, but on another general purpose CPU, which does nothing else, called the protocol processing engine or PPE. Since the PPE doesn't have any other tasks, the overhead from scheduling and interrupt processing can mostly be avoided.

Oritz et al. [14] look at methods to reduce overhead and increase throughput in modern multicore systems. While the emphasis lies in comparing offloading to onloading and theoretical analysis which method performs better under which workloads, the paper also summarizes most of the past research on networking overhead which includes most of the literature presented in this section.

2.1 User-level networking

One approach to reduce overhead that has come up repeatedly over the past years is user-level networking. The idea behind user-level networking is to eliminate the overhead of copying the data around through various buffers in the networking stack and to the application by giving the applications direct access to the network interface. This gives the applications quite a lot of flexibility concerning protocol processing and buffer management. But it also introduces some difficulties that have to be overcome, since a network interface is generally accessed by multiple applications that shouldn't interfere with one another for security and fairness reasons. Another challenge is a direct result of DMA, since the network interface controller can write its data directly into memory, the application has to be prevented from accessing memory that it does not have access to. The following literature elaborates on these problems and proposes different approaches.

2.1.1 Osiris

The first article (of those listed in this section) mentioning the idea of allowing applications to access the network interface in order to enhance performance was written by Druschel, Peterson and Davie [5]. It uses the Osiris ATM network adapter to implement and benchmark user-level networking using an alternative firmware for the adapter. Since ATM is based on virtual circuits they provide an obvious criterion to (de)multiplex the interface to the different applications. One mechanism used to reduce overhead is to minimize interrupts (almost no interrupts on the transmit path, and heavily reduced interrupts on the receive path) by introducing a completion queue in addition to the send and receive queues.

2.1.2 U-Net

U-Net [23] is also based on ATM. It introduces the idea to provide both hardware and emulated software endpoints, since the hardware endpoints are scarce and not all applications need the superior performance. The topic of implementing TCP/IP using this architecture is also briefly addressed.

2.1.3 Arsenic

Since the Arsenic [15] gigabit ethernet network interface controller is not based on ATM as the previous approaches, the choice of how to (de)multiplex the packets is less clear. The proposed solution are filters to be installed onto the NIC, that decide which packets go to which virtual interface. Arsenic also supports traffic shaping and scheduling, to ensure fairness.

2.1.4 VIA

In contrast to the other architectures mentioned above, the virtual interface architecture [7] is not specified with one concrete network technology in mind, instead it is used as a basis for several networks like InfiBand and iWARP. VIA is connection oriented and allows the application to choose from different reliability levels and offers features like remote DMA (RDMA).

2.2 Low-level optimizations

While the optimizations presented in the previous paragraphs focus on the need to process every packet in a central instance (either a network server in the user-space or an implementation of the network stack in the kernel), the literature presented in this paragraph looks at optimizations that can be applied regardless of where the packets are being processed.

Huggahalli et al. [8] present a technique called direct cache access and its use in networking are discussed. The basic idea is to give the network interface controller the possibility to transfer incoming data directly to the CPU cache since it will probably be accessed soon when the packet is processed by the network stack. If the written data is accessed while it is in the cache, the latency is reduced in two places, first copying the data to memory by the NIC has been avoided, and second there are no cache misses when the data is accessed. The paper analyzes in which situations DCA can improve performance. Since the hardware used in their benchmarks is similar to our hardware for this thesis, specifically 10Gbps Ethernet attached over PCI express, the presented results can be expected to apply to our hardware as well.

One article that discusses techniques and optimizations to improve latency in the network stack implementation were explored by Mosberger et al. [13]. While some of the techniques described might not apply to current hardware since the article optimizes for the DEC Alpha architecture, others can probably be adapted to today's CPUs.

Chapter 3

Background

3.1 Barrelfish

Barrelfish is based on a multikernel architecture [2], which basically means that there is a microkernel running on each core that does not share any state with the kernels on the other cores, so the machine is regarded as a distributed system. The cores or CPUs don't have to be identical, one motivation for this architecture is to support heterogeneous architectures. The kernel or CPU driver as it is called, is the only code that is executing in kernel mode (or ring 0 in x86 terminology). It is supported by the so called monitor service on each core that is responsible for coordinating the rest of the system, including communications with other cores.

3.1.1 Capabilities

As in many microkernels memory and other system resources as dispatchers (kernel threads in Barrelfish terminology) are managed by the user space and the kernel merely provides the necessary mechanism. Barrelfish uses a capability based approach [22].

3.1.2 Inter-dispatcher communication

The different services and applications running on top of Barrelfish need a way to communicate with each other. This mechanism is called inter-dispatcher communication or IDC [1] and provides message passing, and is provided using multiple back ends that handle different situations such as inter-core communication (UMP) versus core-local communication (LMP). LMP stands for local message passing and is similar to the inter-process communication mechanisms offered by many microkernels such as L4 [10] using a system call. Communication between dispatchers on different core on the other hand is handled mainly in user space, in a shared memory region as used by URPC [4].

Depending on the back end used the sender of a message can also send some capabilities along with a message (not all capabilities can be transferred). This makes it possible to share e.g. memory regions with other protection domains.

IDC interfaces are specified in a domain specific language called Flounder (documented in Barrelfish technical notes) and are compiled into stub code that allows them to be used in a clean manner independent of the back end used for a particular message. Flounder uses the remote procedure call abstraction.

3.2 Original network stack

The Barrelfish network stack is based on an exokernel architecture, meaning that most of the network functionality is provided by the lwip [6] library (could also be replaced by another library, or the application could provide its own implementation) running in the address space of every application that uses networking. To coordinate the applications the netd [17] service coordinates access to ports, and performs some other system wide network services. In every Ethernet device driver there is an instance of the ethersrv library running, that interfaces between the system and the driver and distributes packets to the applications using the bulk transfer mechanism [21].

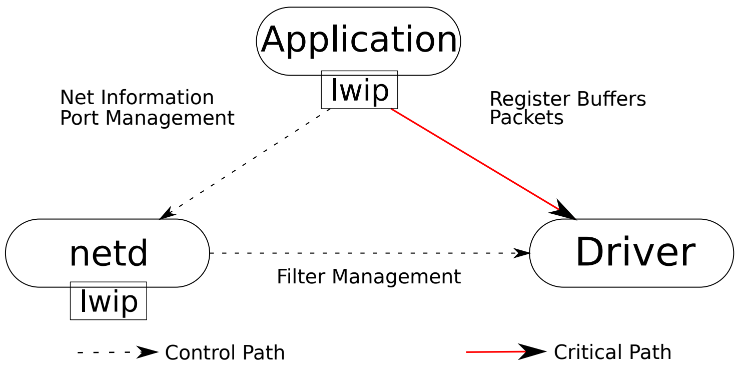


Figure 3.1: Original architecture overview

3.2.1 Network card driver

In the original architecture the device driver for the Ethernet card just passes packets from the ethersrv library to the device and vice versa. During the initialization phase the driver registers callbacks to get the MAC address, send a packet and to remove sent packets from the transmit queue.

For sending buffer chains representing the packets are provided by the library, and can (if the device supports it) directly be inserted into the transmit ring buffer of the device. After the packet is sent the driver notifies the library so it can release the buffer. On the receiving side the driver allocates buffers and notifies the library whenever a packet arrives, after which the buffer can be inserted again into the receive queue.

With modern network hardware that do not require physically contiguous memory, this allows for a straight forward driver implementation, that basically just manages the receive and transmit queues and handles interrupts.

ethersrv library

The ethersrv library takes care of interfacing with the rest of the system. To that end it provides two IDC interfaces, called *ether* and *ether_control*. The *ether* interface is used by every client that transfers data over the network, while *ether_control* is only available to netd mainly for filter management. Listing 3.1 shows the most interesting parts of the Flounder interface definitions, for both of these interfaces.

```
interface ether "Generic Ethernet Driver" {
    call register_buffer(cap buf, cap sp,
                         uint64 slots, uint8 role);
    call get_mac_address();
    call sp_notification_from_app(uint64 type,
                                  uint64 ts);
    response sp_notification_from_driver(uint64 type,
                                         uint64 ts);
};

interface ether_control "Generic Ethernet Control" {
    call register_filter_memory_request(cap mem);
    call register_filter_request(uint64 id,
                                 uint64 len_rx, uint64 len_tx,
                                 uint64 buf_rx, uint64 buf_tx,
                                 uint64 filter_type, ...);
    call deregister_filter_request(uint64 filter_id);
    call pause(uint64 filter_id, uint64 buf_rx, uint64 buf_tx);
    call unpause(uint64 filter_id);
};
```

Listing 3.1: Original ethersrv interfaces

When sending a packet the physical address of the buffers sent by the application can directly be inserted into the transmit queue. For receiving the driver has to register its own buffers, and when a packet arrives the data is copied out of the buffer by the ethersrv library.

To determine to which application or more precisely into which buffer a packet is copied (demultiplexing), netd can register filters [3]. Those filters are represented by byte code that is interpreted by a virtual machine when applying the filter to a packet. Filters are at the moment only applied to incoming packets, outgoing packets are not filtered yet. The driver itself does not have to worry about filtering. netd registers these filters using a shared memory region it registered during initialization and a notification message, both using the *ether_control* interface. The following filter types are currently used:

- TCP/UDP port
- TCP connection (source and destination port and ip)
- ARP packets (forwarded to all clients)

One thing that makes filtering more complicated is IP fragmentation, since that leads to the L4 headers not being in every packet relevant to the flow. The current solution is to handle this special case in the ethersrv library by keeping a list of fragment filters that move the packets to the correct buffer, in some cases the fragments are buffered there until the packet with the full header arrives (out of order reception).

Another architectural quirk is caused by the emulation of some POSIX behaviour for network connections if *fork* is used. To be able to support applications that use fork to spawn workers for incoming TCP connections, a combination of the *pause* methods and TCP connection filters is used to first hold back the relevant packets until the child has initialized its networking stack, and then redirect the incoming packets.

Transfer of packets to and from applications is implemented using the Barrelfish bulk transfer mechanism, which requires a shared buffer per queue shared between the application and the driver registered using the *ether* interface. Also required for the bulk transfer mechanism to work are notification messages over IDC in both directions, they too are in the *ether* interface.

3.2.2 netd

Basically netd has two different duties, it manages access to ports by applications and performs system wide networking services including initialization of the network interface (DHCP) and handling all packets not handled by anyone else. An IDC interface called *netd* (Listing 3.2) provides networking information like IP and MAC address of the interface, and allows other applications to request ports.

```
interface netd "Network Daemon" {
    rpc get_ip_info(out ipv4addr ip,
                    out ipv4addr gw,
                    out ipv4addr mask);
    rpc get_mac_address(out uint64 hwaddr);
    rpc get_port(in port_type type,
                 in bufid id_rx, in bufid id_tx,
                 out errval err, out uint16 port);
    rpc bind_port(in port_type type, in uint16 port, ...);
    rpc close_port(in port_type type, in uint16 port, ...);
};
```

Listing 3.2: Original netd interface

Access control to ports by netd ensures that there are no conflicts between multiple applications that use the same port. There is also a function to allocate any free port (e.g. for TCP client connections). When a port is assigned to an application, netd generates a filter for that port and registers it using the *ether_control* interface.

As mentioned above netd also performs network services that are not specific to a specific application. This includes responding to ARP requests of other machines, and handling ICMP packets. Those services are performed using lwip, which in the case of netd gets all incoming packets not handled by anyone else.

3.2.3 Application

An application that wants access to the network (be it as a client or as a server) needs connections to two services, netd and the network card driver. At the moment the network stack used is the lwip library, but any implementation could be used here, the application could also interface directly with the two services.

After connecting to the card driver two buffers are registered with the driver, one for receive and one for transmit. Those buffers are managed using the bulk transfer mechanism.

Communication with netd is only required for management tasks such as initialization or requesting a new port. Other than that all communication concerning packet transfer happens directly with the card driver.

lwip library

lwip is currently the default network stack used in Barrelfish. It includes a full-featured TCP/IP implementation as well as other protocols, e.g. DHCP. On the client side it provides the well known BSD socket interface for compatibility, and also a native interface that allows for more efficient and sometimes more controlled operation. To send and receive packets it also interfaces with the card driver and for management with netd, as mentioned above.

pbufs, the mechanism lwip uses for internal buffers (similar to BSD's mbufs), reference memory directly in the bulk transfer buffers (see below), so they can be sent/received without copying. But if the traditional socket interface with `send()`/`recv()` is used, the data has to be copied between the specified buffer and the pbuf. The native interface uses multiple call-backs (e.g. for data reception, with a pbuf as a parameter) and allows the application to take control of buffer management. In the current implementation there is a limitation that makes allocation of a new pbuf and copying of the data into it necessary, even if it was in a pbuf before.

3.2.4 Bulk transfer mechanism

The transfer of packets to and from the network card driver uses the bulk transfer mechanism, which is a library that implements a solution to the producer-consumer problem that allows zero-copy data transfer between two protection domains (address spaces) in Barrelfish using a shared memory range. In the context of networking two bulk transfer buffers are used, for the receive side the card driver assumes the role of the producer and the application is the consumer, for transmit they switch roles.

A bulk transport buffer is divided into multiple components, the two most important ones are the shared pool where the packet data goes and the consumer queue which is a ring buffer used for communication between producer and consumer. Multiple shared pools in different memory regions can be used in connection with one consumer queue. The basic operation of the bulk transfer mechanism is similar to modern network cards that use descriptor rings to manage their queues. The memory in the shared pool is managed by the producer. The main difference is that the entries in the consumer queue do not point directly to physical memory but into the shared pool, more precisely to a slot in the shared pool. A slot is a piece of memory in the shared pool of fixed size. In the consumer-queue there are two sets of elements (slot-pointers), those that have been produced and await to be consumed and the other set consists of those elements that have been consumed and the consumer is done with. Note that the consumer does not have to return the consumed elements in the order it consumes them, but can keep each slot for an arbitrary time span.

Demultiplexing

As also discussed in the technical note [21] there exists a trade-off between security/privacy and performance, especially if the data comes from hardware that just blindly puts its data into the next provided buffer. To achieve true zero-copy all the clients need to have access to one (or multiple buffers) into which the device copies its data, and it can be decided afterwards to which client the data goes, which means that those clients are not shielded properly and could see data of other clients. Depending on the concrete use-case that can be acceptable or not, if not there is no way around a copy from some internal buffer.

The current ethersrv receive code uses the secure method with separate buffers and copies once the packet is received and filtered. For transmit this is not an issue since we do not have to demultiplex so there copies can be avoided. Also note that the data can only be exchanged using the shared pool(s), so if the source/destination (depending on the direction of the transfer) in the application is not in the shared pool a copy is necessary (discussion about interfaces to lwip in previous section).

3.2.5 Critical path

The critical path for packet reception and transmission is highlighted in figure 3.1. As described in the previous sections it only involves the application sending/receiving the packet and the driver for the network card on the software side. The following lists explain the sequence of events when receiving a packet and sending a reply (only the range of events that contribute to the latency) under the condition that the queues are empty and that there is no other traffic and both the driver and the application are on the same core. We assume that the application uses the native lwip interface (exactly as the Barrelfish application *echoserv* does).

- Driver receives interrupt signalling newly arrived packet.
- Driver finds packet in queue and passes data to ethersrv library.
- ethersrv library applies multiple filters (IP fragment, application packet, ARP packet) and our packet matches as an application packet.
- ethersrv library copies packet into a slot in the bulk transfer buffer determined by filter.
- ethersrv library adds slot to consumer-queue.
- ethersrv library signals application that there is a new packet in consumer-queue.
- Context switch to application
- lwip glue code checks consumer-queue for new packets.
- lwip glue code finds new packet and passes it to lwip.
- lwip processes packet and passes the modified pbuf to the application.

Packet transmission includes basically the same steps except for the demultiplexing part in different parts. After the processing by lwip the packet data is not accessed by the CPU anymore

- Application instructs lwip to send the reply using a new pbuf (copy of data necessary).
- lwip generates new packet and passes it to glue code.
- lwip glue code flushes packet contents from cache to main memory.
- lwip glue code calculates slot for pbuf and inserts it into the consumer-queue for transmit.
- lwip glue code signals driver that there is a new packet in consumer-queue.
- Context switch to driver
- ethersrv library checks consumer-queue for new packets.
- ethersrv library finds new packet, records some internal information and passes it to the driver.
- Driver adds descriptor for packet to transmit queue (and records some internal information to pass on to the ethersrv library when transmit is done).

3.3 Network controller

This section will provide an overview over the network controller used, Intel 82599EB [9], and the features provided focusing on those that are potentially useful to reduce latency. As already described in the introduction this controller supports 10 Gbps Ethernet and on the other side PCI express 2.0, one PCI function per device (changes when virtualization is used, see below). The interface used by the driver consists of a memory region containing a set of memory mapped registers, where a number of other memory regions can be configured by the driver e.g. for descriptor rings.

3.3.1 Multiple queues

One of the features that distinguishes this controller from common cheap 1 Gbps controllers is the support for multiple hardware queues, 128 receive and 128 transmit queues. Those queues are managed using descriptor rings in host memory as they can be found in most modern network cards (e.g. Intel e1000 cards). A queue ring consists of a configurable number of descriptors of 16 bytes each, where the concrete format depends on the queue type (receive/transmit) and the configuration (advanced/legacy descriptors). The advanced descriptor format is required for multiple features (such as several offload features). To maintain the queue once it is configured, there are two registers for queue head and tail pointer, or more correctly head and tail index, where the head index is written by the device and the tail index by the driver.

In a receive queue the host just has to fill in a buffer address where the controller puts the data, the buffer size is configured globally for all descriptors in a queue (has to be a multiple of 1024). When a packet arrives the controller copies the data into a sequence of buffers (as many as needed) and adds some additional information to the descriptor, such as the length of the data copied to the buffer referenced and flags such as "end of packet" that specifies if a descriptor is the last descriptor for a particular packet, and "descriptor done" that indicates that the controller is done copying data to this particular buffer. Also after the card is done with a descriptor it increases the index value in the queue head register.

A transmit queue operates in a similar manner as a receive queue. The main difference is that in this case the host has to provide more information, instead of just the address also a packet length has to be provided and some offload features can be enabled on a per-packet basis (see below) and the controller basically just sets the "descriptor done" flag after the packet is sent, and increases the queue head register. There is also a feature called TX head pointer write back that disables write back of TX descriptors entirely, instead a separately configured location is updated with the new queue head index. This is useful in combination with direct cache access (see below) to reduce cache thrashing caused by hardware and software writing the same cache line modifying different descriptors.

Descriptor buffering

The controller has some on-die buffer space for packets and descriptors, the exact amount of buffer space depends on the configuration as other features use this buffer too (see information about flow director filters below). On the receive side no descriptors are cached, they are fetched on demand after a packet is received and written back immediately after the packet is copied to memory. For transmit queues a policy for descriptor fetching and write back can be configured using three different values PTHRESH, HTHRESH and WTHRESH. Descriptors are always fetched if the queue is empty and the software writes the tail pointer, if the queue was not empty and some buffers are cached a descriptor pre-fetch operation is started if there are less than PTHRESH descriptors buffered and more than HTHRESH ready descriptors in host memory. Write back occurs if at least WTHRESH descriptors have accumulated or if a configured timer expires. Note that there is a trade-off between low latency and PCI express bus overhead. To keep latency as low as possible it is desirable to fetch as many buffers in advance as possible as soon as they are available (PTHRESH maximal, HTHRESH minimal), and write them back immediately after the controller is done with them (WTHRESH minimal). But frequent write backs also generate load for the PCI express bus, which can be reduced by combining as many descriptors as possible into one fetch/write back.

Queue assignment

To assign packets incoming packets to different queues (see figure 3.2), several different filter types are available. The following list provides an overview in the same order as they are applied (first filter that matches is used, after that no other filters are checked):

1. L2 Ethertype filters

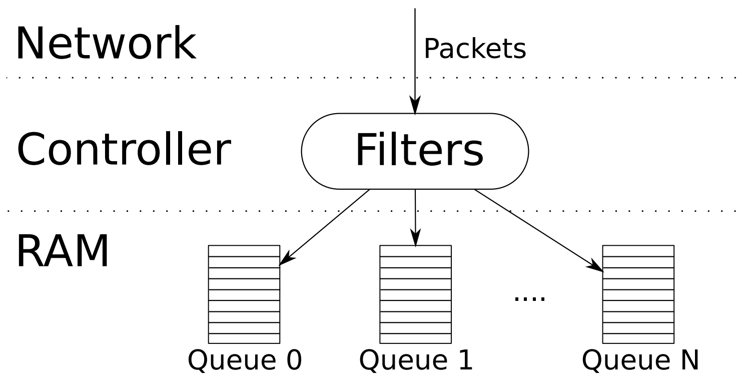


Figure 3.2: Hardware demultiplexing

2. L3/L4 5-tuple filters
3. TCP SYN filter
4. Flow director filter (configurable number)
5. Receive side scaling (RSS)

Note that the matching order of the SYN filter and 5-tuple filters can be configured to be the other way around.

L2 Ethertype The 8 L2 Ethertype filters assign a queue id based on layer 2 information such as the type field in the Ethernet frame. Could for example be used to filter ARP packets into one queue.

L3/L4 5-tuple More use for most applications can be gained from the 128 5-tuple filters. These filters allow packets to be filtered based on L3 information such as source and destination IPv4 address, L4 protocol (TCP, UDP or SCTP), and the L4 protocol fields for source and destination port. For each filter a mask can be configured that specifies which of those fields should be matched, so a filter can be set up that matches all packets from any source IP or port to a specific destination port. There is also a 3 bit priority field that allows the filters to be ordered in case multiple filters match.

TCP SYN This filter matches all TCP packets that have the SYN flag set, i.e. new TCP connections.

Flow director The flow director mechanism allows by far the most filters to be registered. It can be configured in two modes, perfect match and hashing, the first one allows up to 8K filters to be registered while the hash mode allows up to 32K filters. But those filters are stored in the on-chip buffer that is also used for packets and descriptors, so if more flow director filters are enabled less space is available for packet and descriptor buffering. They allow matching based on the same properties as the 5-tuple filters, plus some additional ones such as IPv6 addresses and a flexible 2 byte field whose offset can be configured. In contrast to the 5-tuple filters the flow director filters only allow the mask flags (and the offset for the flexible 2 byte field too for that matter) to be configured globally for all filters.

The two modes differ in how the filters are configured and how they are matched. For both modes they are stored in a hash table using bucket hashing. In hash mode only a hash and an action that is executed in case of a match (includes a queue id) have to be registered for a filter, and also the filters are only compared based on the hash, while in perfect match mode first a hash is used to find a bucket and then the fields are compared directly hence the values for those fields have to be specified when adding a filter.

Receive side scaling Receive side scaling or RSS is not used to direct specific packets to specific queues, but rather for load balancing between multiple queues that are used by multiple cores. It works by first calculating a hash over multiple predefined protocol fields in the packet, after that this hash is used to look up a queue id for that hash in a small lookup table (128) entries that contain the queue id, but supports only 16 queues.

Virtualization

Especially for use in virtual machines the controller can be divided in multiple isolated virtual devices, which appear to the host as separate PCI functions (called virtual functions). Aside from providing independent interfaces for the virtual machines, the division into multiple PCI functions allows the IOMMU (in Intel terminology VT-d) of modern CPUs/chipsets to be used, to make sure the virtual machine only has access to physical memory it actually owns. The registers available in the virtual functions are mainly equipped for queue management and provide a mailbox mechanism to communicate with the driver for the physical function, for management tasks such as filter registrations.

When virtualization is enabled the 128 queues are divided into a fixed number of equally sized pools, which can be configured to be either 16, 32 or 64. Incoming packets are routed to a pool based only on L2 characteristics such as MAC address or VLAN tags, the higher layer filters described above can only be used to select into which queue a packet is stored inside a pool. Packets can be assigned to multiple pools at the same time.

3.3.2 Offloading

The controller also provides a number of features that offload some protocol processing tasks such as calculating checksums for TCP or UDP packets both for checking them on receive and for filling them in when sending packets. TCP large segment offload (TSO) allows the network stack to add TCP packets to the transmit ring that are too big to be sent in a single Ethernet frame (up to 256 KB), so the network controller has to send them as multiple packets (TCP segments), which includes keeping track of sequence numbers of each packet, as well as calculating checksums on multiple layers. The counterpart on the receive side is receive side coalescing which allows the network controller to merge multiple packets (TCP segments) belonging to the same TCP connection into one or multiple buffers. Another offloading feature provided is header split, where the packet header and payload can be put into different buffers when receiving a packet, here the following packet types are supported Ethernet, IPv4 and IPv6, TCP, UDP and even NFS.

3.3.3 Interrupts

Multiple interrupt configurations are supported, on one hand the controller can use a single legacy PCI interrupt or a single MSI, on the other hand if MSI-X is used, up to 64 different interrupt vectors can be configured e.g. for different queues. Interrupt throttling reduces the rate at which interrupts are generated, which allows multiple packets to be processed per interrupt and thus reduces CPU load since every interrupt causes CPU overhead. To that end a time interval can be configured that specifies the time minimal time interval between two interrupts (only applies to receive and transmit interrupts).

Since not all connections or packet types are equally sensitive to latency, a mechanism called low latency interrupts (LLI) is provided, that allows specific packets to trigger interrupts immediately. Some of the filters above (such as Ethertype filters or 5-tuple filters) can enable LLIs for specific packets. Low latency interrupts can also be controlled by a moderation mechanism. This mechanism uses credits that are decreased for each LLI, if the credit counter reached zero the interrupt is delayed. An interval has to be configured that controls how often the counter is increased to allow additional low latency interrupts for that queue.

Note that interrupt moderation is also necessary for receive side coalescing. If every incoming TCP segment triggers an interrupt the controller has no opportunity to merge multiple segments.

3.3.4 Direct cache access

In traditional settings packet data that is generated by a device has to be written directly to main memory and the CPU has to read it from there without using caches. Direct cache access [8] allows a device to write data directly into the CPU cache. This reduces the memory latency when the packet contents or descriptors are first accessed by the driver or network stack.

Since the transmit queue is mostly very short in low to medium loads, cache thrashing can occur if both the head and tail pointers point to descriptors in the same cache line, which in turn causes multiple writes to the same cache line by the CPU and the network controller. In the receive queue there is usually a large number of buffers, so there that problem does not occur. As mentioned above TX head pointer write back can be used to resolve this problem.

Chapter 4

Approach

This chapter provides an overview over the original Barrelfish network stack, analyses (some of) its limitations and discusses possible improvements. The e10k driver used for the profiling of the existing code, only uses basic hardware features required to send and receive packets, using just one hardware queue for both directions.

4.1 Sources of latency

To get some indication where the time is spent when sending and receiving packets, this section presents profiling results of the critical path.

4.1.1 Profiling results

For the results presented in this section the following setup was used on the sbrinz1 machine : essential Barrelfish services as well as netd, e10k driver and *net_latency* running on core 0. net_latency was configured to generate 32 round trip runs (send packet and wait for reply) with 64 byte TCP packets. The receiving side was on gottardo, running the same setup except echoserver is running instead of net_latency. To get the desired results, the critical path was annotated with trace events roughly corresponding to the items listed in the previous section.

The total round trip time measured was 75.6μs Since we are interested in where the time is spent in the network stack, only those two parts were analyzed in the trace. The total time for transmit was 9.38μs, for receive it was 14.85μs on average, more detailed results can be found in tables 4.1 and 4.2.

38%	3.63μs	Remove packet from consumer queue
28%	2.52μs	Context switch to driver for notification
14%	1.35μs	Protocol processing in lwip
12%	1.12μs	Flush packet content from cache
8%	0.78μs	Enqueue packet in consumer queue

Table 4.1: Time spent in different parts of transmit code

44%	6.46μs	Enqueue packet in consumer queue
19%	2.77μs	Copy packet into application buffer
14%	2.13μs	Apply filters to select destination buffer
10%	1.51μs	lwip protocol processing
9%	1.36μs	Context switch to application for notification
4%	0.61μs	Remove packet from consumer queue

Table 4.2: Time spent in different parts of receive code

These results were obtained using the Barrelfish trace library. This library records the time-stamps of different tracing events that can be generated using a simple function call, and allows this trace to be dumped later. The output consists of a tuple for each event consisting of core id time-stamp and an event id. A simple test measuring the latency of the network stack with tracing enabled and disabled showed that the overall latency is not noticeably influenced by the tracing calls. So the results obtained using tracing can be regarded as accurate enough for results in the order of magnitude expected here.

4.1.2 Analysis

Comparing both tables the most obvious result is that a lot of time is spent on queue management for the bulk transfer mechanism, for both directions handling the queue to add and remove packet together takes a major part of the overall time. This was a rather surprising result, since it is basically just a FIFO queue with one producer and one consumer and without large amounts of data being copied. A closer look at the code for queue management showed that there were a lot of *mfence* instructions (memory barriers). In the code that enqueues a package from lwip for the transfer to the driver more than 30 such instructions were executed (most of them useless if both ends are on the same core). Also the code does a lot of redundant loads from shared registers in the queue to local copies, and there is also a number of sanity checks that should not be necessary once the code works properly.

The context switch and notification part is mostly explained by IDC overhead and the TLB misses because of the TLB flush that occurs on an address space switch.

Receive path

On the receive path filtering and packet copying is the next major part of the latency. Not surprising those add up to about the difference between receive and transmit since they are the only major difference. On the receive side part of the problem turned out to be due to the packet buffers in which the packets are received being mapped non-cacheable, which means that every memory access on those buffers results in a cache miss, and those filters access several locations inside a packet. If this memory range is marked as cacheable, only the first access should result in a cache miss, since the header should fit into one cache line.

After the filters are done and a target buffer has been determined the packet gets copied, which resulted in additional cache misses (Barrelfish memcpy uses 8 byte units, i.e. 7 unnecessary cache misses on the source side), and also TLB misses assuming the driver is freshly scheduled. If both the card driver and the application are on the same core, and assuming the application is scheduled soon after receiving the packet it should at least find the received packet in its buffers to still be in the cache. Last but not least copying also uses up additional CPU time albeit not very much for small packets.

4.2 Reducing latency

The sources of latency discovered and analysed in the previous section provide a reasonable starting point to look at possible optimizations. This section will provide multiple suggestions on how latency can be reduced, not all of them will necessarily be implemented later on.

4.2.1 Bulk transfer

Probably the most obvious part that needs optimizations is the code that interacts with the bulk transfer mechanism and the implementation of the mechanism itself. Code optimization by hand is one way to go here that should have some potential, just disabling the memory barriers for test purposes (safe in our case) cuts the time spent in the queue management code approximately in half, depending on the machine. Removing some of the unnecessary checks (or at least make them optional for debugging), as well as reducing the number of register reloads to local copies should lower the latency noticeably.

IDC performance

The latency introduced by the bulk transfer mechanism also directly depends on IDC latency (at least if there is not much traffic, so the queue is empty long enough) since notifications sent are based on IDC. Depending on the setup this also includes context switches. If driver and application are on different cores so UMP can be used, all context switches can be avoided in the best case.

4.2.2 Demultiplexing

As already mentioned above software demultiplexing has some non-desirable effects, and it would be nice if the network controller could just copy the packet into the right buffer, so that it does not have to be copied afterward. As described in [21] this can be achieved with the bulk transfer mechanism, if multiple applications use the same shared pool for their received packets. But using that approach we lose all separation between different networking applications in regard to privacy and data integrity.

One way to do this in a more safe manner is using multiple hardware queues as they are offered by the 82599 controller, and use one queue per application and only add its buffers to the receive ring. This implies also that the controller needs to know the criteria for demultiplexing. Regrettably the filters in the network controller used here are much less flexible than the virtual machine mechanism used in bfdmux [3] that is currently used, so they can't be mapped directly.

Direct queue access

Another possibility to reduce latency even more would be to give the application (or the lwip instance in it) direct access to the receive and transmit ring, and let it insert its buffer descriptors directly [5] [23] [15] [7]. In that case no communication with the driver would be necessary on the critical path, and the bulk transfer mechanism would not even be used. Figure 4.1 provides a schematic overview.

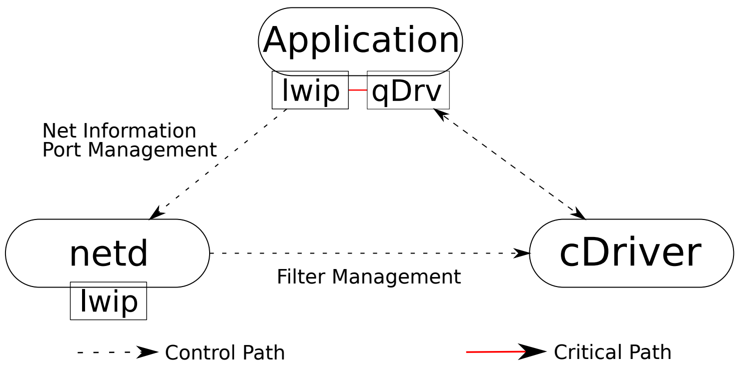


Figure 4.1: Architecture overview with direct queue access

Unfortunately this introduces multiple problems. It compromises memory protection since physical addresses have to be specified in the receive and transmit rings, so if a hostile (or buggy) application has access to that queue, it can read or write any memory location, regardless of it is mapped into its virtual address space or not. Another problem is that in order to maintain the queue the tail register of the queue has to be written, but those registers are too close together in the memory mapped region to be separated properly using paging.

Both problems would be solved if the virtualization feature is used, since it provides a separate PCI function with a separate memory region for its registers, which would also enable the use of the IOMMU, to control access to physical memory. But this introduces another problem since packets can only be assigned to a queue based on L2 criteria, and not the whole range of filters.

A compromise that would allow at least to reduce the latency on the receive side would be to map the queues read-only, and requiring buffers to be registered using a driver. With that approach the application could at least detect when a packet arrived, and directly process it, without context switches, same for transmits that are finished. This would allow the application to batch the registration of receive buffers and amortise the cost for the context switch.

4.2.3 Cache optimizations

Because of the cache coherency guarantees provided by the platform, the buffers used in the receive and transmit rings don't have to be mapped non-cacheable, just as the memory regions that contain the rings don't have to be either. Also the packet contents don't need to be flushed before a transmit, since the cache coherency mechanism will take care of that. If the network controller can read the data directly from the cache, the time savings are twofold, no time for an explicit flush and no RAM access by the hardware.

Direct cache access

When caching of a memory region is enabled, and if the right hardware is used, direct cache access can be used to optimize memory accesses further. As described in the background chapter, DCA allows the network controller to write its data directly into the CPU cache. This is especially useful for latency sensitive applications that use small packets and access the packets soon after they arrive [8]. With many large packets DCA can lead to cache thrashing, and if the packets are not accessed shortly after they arrived, they might get evicted from the cache again. Using DCA cache misses due to packet payload or descriptor accesses can be eliminated entirely for many cases.

4.2.4 Interrupts

Interrupts are not really useful for low latency communication either, since polling does not require a transition to kernel space and back. But they can be used if CPU time or power consumption is too valuable to be wasted on polling if for a longer time no packets arrive and a higher latency for the first packet after that can be tolerated.

4.2.5 Other hardware features

There is a whole bunch of features mentioned in the background chapter and even more that are not, that are not really useful for reducing latency or might even be contra productive since they optimize for bandwidth such as e.g. interrupt throttling, or receive side coalescing. Also some of the offload features such as TCP large segment offload are not intended for use with small packets (plus TSO adds some initialization overhead when sending packets). Others such as Header splitting simply don't fit into the Barrelfish network stack architecture because of the exokernel based approach.

But there are some other features worth looking at. That includes the descriptor fetching mechanism described in 3.3.1 that can be configured for different performance characteristics. But that mainly matters when there is a higher load of packets being sent and received.

4.2.6 Multi core optimizations

When multiple cores are available with complex cache hierarchies as they are common in today's servers, the placement of different system components on different cores becomes non-trivial. On one hand parallelism is helpful, but on the other hand communication performance between different components also varies (such as slower access to devices or memory in different NUMA nodes). To optimize latency some benchmarking has to be done to reveal how the latency behaves when components are moved around.

For example if the two ends of a bulk transfer consumer queue are on one core notifications are sent using LMP requiring context switches, while if they are on different cores, those context switches can be avoided in some cases. But in the worst case communication is more expensive.

Chapter 5

Implementation

After the analysis of the original Barrelfish network stack, this chapter will present the modifications that were made to reduce latency, and the resulting design.

5.1 Modified network stack

The following section will describe the modified network stack. Figure 5.1 provides an overview over the new architecture. From an architectural point of view the most important change is that the e10k driver has been split up into one card driver and multiple queue drivers each servicing one hardware queue.

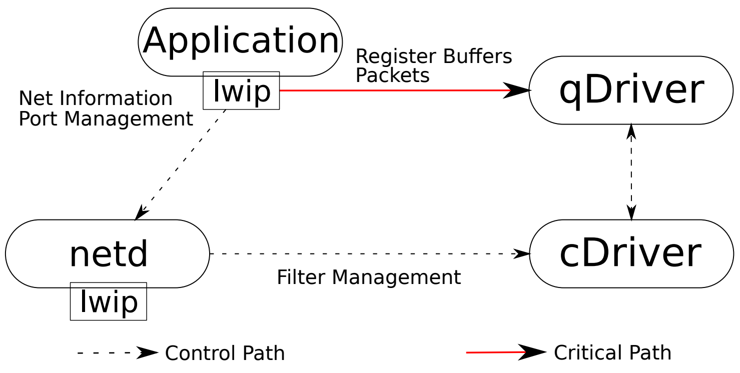


Figure 5.1: Modified architecture overview

5.1.1 Driver

To make the runtime configuration as flexible as possible, the driver now consists of multiple parts that run as independent processes. The card driver takes care managing the hardware itself, basically everything that is not replicated with the queues, and spawns a queue driver for each queue that is enabled. Filters can be registered by the ethersrv library to assign packets to the different queues.

Queue drivers

The queue drivers register their memory regions that contain the descriptor rings for receive and transmit with the card driver, which configures the device registers accordingly. Also filters for the different queues are programmed by the card driver, this eliminates synchronisation problems, since only one thread accesses these registers, and needs to keep track

An alternative would be to use one driver with multiple threads, but since not much communication is necessary between the components this would introduce more problems than it solves. No synchronization primitives have to be introduced, and independent processes also reduce overhead for managing the address spaces of the drivers if they are distributed across multiple cores (e.g. no TLB shoot downs). Explicit communication between a queue driver and the card driver is possible using a private IDC interface. This interface is mainly used for initialization (registering queue memory ranges etc.).

Interrupts

Interrupts are not used in the queue driver for multiple reasons. The main problem is that Barrelfish currently does not support MSI-X which would be necessary to allow the network controller to use multiple interrupts. Without multiple interrupts, either one driver has to forward interrupts to the others, or they have to be broadcasted to all of them, both not optimal for low latency.

Filters

To control which packets go where, the card driver manages the filters that can be programmed into the network controller. As already discussed in the background chapter, those filters are not as flexible as the ones provided by the bfdmux library, so the binary filter format is not sensible as a basis to configure the hardware. Instead the driver provides multiple types of filters, that correspond to their uses in netd:

- TCP/IPv4 connection with source and destination IPs and ports
- TCP server port
- UDP server port

In the current version those filters are mapped down to the hardware using the L3/L4 5-tuple filters. These filters are straight forward to program and more flexible than the other choices. Another possible choice would have been to use the flow director filters since they are available in a larger number, but those are more complicated to program and use up buffer space on the network controller.

5.1.2 ethersrv library

The ethersrv library has been split up more clearly into two parts, one for management functions such as registering filters, and the other one for data transfer. Those two parts more or less correspond to the two interfaces provided by the ethersrv library, *ether* and *ether_control*. In the case of e10k the ether interface is provided by each queue driver, while the control interface is provided by the card driver, or more correctly by the ethersrv instances running in those drivers. The ether interface has not been modified significantly, while the ether_control interface looks quite different. It provides new methods for queues management and the methods for filter registration have been modified.

As already mentioned above, the bytecode interface for the filters is not practical for configuring hardware filters, so a number of methods has been added to register different types of filters for a queue. Listing 5.1 provides an overview of the most important aspects of the modified ether_control interface.

```
interface ether_control "Generic Ethernet Control" {
    call get_mac_address();

    call request_queue(queueid id, uint8 core);

    call filter_register_ipv4_tcp_port(queueid id,
                                       l4port  port);
    call filter_register_ipv4_udp_port(queueid id,
                                       l4port  port);
    call filter_register_ipv4_tcp_conn(queueid  id,
                                       ipv4addr local_ip,
                                       l4port   local_port,
                                       ipv4addr remote_ip,
                                       l4port   remote_port);

    call deregister_filter(filterid id);
};
```

Listing 5.1: Modified ethersrv interface

The drivers have to register a couple of callback functions with the ethersrv library. Queue drivers have to provide functions to send a packet, register receive buffers and clean up sent packages. While the card driver has to provide functions to initialize new queues and register different filter types (roughly correspond to the functions in the Flounder interface).

5.1.3 netd

netd has been adapted to the new ethersrv interfaces, and now performs the additional task of assigning queues to applications. To that end the Flounder interface to netd has been modified to include queue id parameters for the methods modifying filters, and a method to allocate a new queue for an application.

5.1.4 Critical path

In regard to the critical path not much changes, at least as long as only one application is observed. Since the applications no longer communicate all with the same driver, but with multiple queue driver instances, those queue drivers can work independent from one another on different cores if desired. This allows for better isolation, but could also improve performance since each application can have a queue driver running "near it" in the NUMA sense.

Some optimizations for the bulk transfer interfaces on both the application side (in lwip library) and on the driver side (in the ethersrv library) have been implemented by Pravin Shinde to reduce the time spent in the code there, since the profiling revealed that a significant amount of time is spent there.

Those optimizations include the removal of unnecessary mfence instructions, the removal of unnecessary sanity checks, and some overly conservative code that updated local copies of the queue pointers more often than necessary. Sending of packets on the wire has also been modified to allow packets to be sent more efficiently, i.e. without initializing temporary data structures that are only used to communicate with the driver, instead the necessary information is extracted directly from the bulk transfer buffer.

5.2 Problems during implementation

This section presents the most important problems faced during the implementation phase and attempted solutions if they were addressed.

5.2.1 Initialization bug in driver

From the first version of the driver one problem has always occurred in connection with the initialization of the card. The network controller only starts transmitting and receiving data after a delay of multiple seconds, even though the status registers indicate that the link is up and ready. Failure to send those packets is not detected by the driver since they are consumed successfully. Sometimes the initialization does not succeed at all, and the driver hangs forever without detecting the problem. The cause of this problem is not known to me, but somehow in the current version it seems to work quite reliably after about 10 seconds. Once the first packet is sent or received, everything works.

5.2.2 ARP filters

ARP resolution caused another problem I found quite late in process of implementing the design described above. In the original implementation incoming ARP packets were simply forwarded to all applications in the driver code. This is however not easily achieved in with the current design, since no instance knows all buffers, and especially if some of the additional features mentioned above are implemented such as mapping the hardware receive queue directly into the application space read-only as an optimization, it gets even harder to do. So at the moment a workaround is used, instead of doing ARP queries directly the applications send a request to netd which then takes care of resolving the IP address to a MAC address, this result is then cached in the lwip ARP cache of the application.

5.3 Limitations

5.3.1 Polling

As already mentioned above the driver does currently not use interrupts since Barrelfish does not support MSI-X yet. In a latency critical application that is probably the right thing, especially if the application is not very CPU intensive or if the queue driver can run on a dedicated core. But in other settings interrupts (maybe even with interrupt throttling) might be desirable.

5.3.2 Scarceness of hardware resources

The current implementation of the network stack only supports hardware filtering. Which is not appropriate for all applications since there is a limited number of hardware queues and filters available, and non-performance critical applications can just as well be served using software filtering, to leave the hardware resources available to the devices that really need it. The other reason for reintroducing software filtering is to support network controllers that don't support sophisticated hardware filtering (such as for example the e1000 cards), they are not supported in the current implementation. This has no influence on the latency results for applications using hardware filtering.

A combination of software and hardware filters might also become necessary if additional filter types are required that are not supported by hardware. In the case of clients that receive packets from both software filtered and hardware queues the simplest decision is probably to use multiple bulk transfer buffers, one for communication with the hardware queue and another one for configuration of the software queue.

One optimization over simply using one default queue for software filtering would be to distribute the packets into multiple queues using receive side scaling, which can also be used with some other network cards such as some e1000 cards. This would most likely improve performance in heavy workloads.

5.3.3 Pause feature for filters

The pause feature for filters that is used in the original Barrelfish network stack is used to emulate some POSIX behaviour in connection with fork as mentioned before. This feature is currently not implemented, and is also not so straight forward to implement, especially the buffering part until the child is ready, redirecting the TCP connection after that to the correct queue should not be a problem.

One possibility could be to initialize the queue for the child process, including the bulk transfer buffer populated with receive buffers and registering a filter for the TCP connection in the parent process and then passing it on to the child process. This would remove the need for explicit buffering of the packets in the ethersrv library.

5.3.4 Multi core bug

There is also a bug that prevents the implementation from working when the application and the driver are on separate cores. In that case the bulk transfer interfaces just blocks. Due to time limitations that bug could not be eliminated in time for this thesis.

5.4 Additional features and optimizations not implemented

Some features that might help to reduce latency were not implemented, either because of technical problems or just because not enough time was available.

5.4.1 Direct cache access

The purpose of direct cache access has been discussed above. While it would probably help to reduce the latency, especially on the receive side, direct cache access could not be implemented within the given time frame, because of lacking support from Barrelfish (or maybe just the BIOS). The network controller signals that DCA is not supported, and since Linux has a driver for it I suppose some parts of it have to be initialized by the operating system. I was not able to find documentation on how to implement it.

5.4.2 Read-only map of hardware queue in application

As already discussed, the Intel 82599 controller is not really adequately equipped to allow hardware queues to be mapped directly into applications to reduce communication latency, since this would allow the application to add arbitrary physical addresses to the descriptor rings and thus circumvent memory protection. But a restricted mapping can help to reduce overhead, that is if the receive queue is mapped directly into application space allowing only read access to it. This way the application can detect that a new packet has arrived just by looking at the memory region. To add new receive buffers there is still a trusted party (driver) required, but those requests can be batched, so as to amortise the communication overhead. A similar optimization can be used with the transmit queue to detect transmitted packet buffers, although here the gain is probably less noticeable since this event is not on the critical path and transmit requests can usually not be batched in a low latency setting.

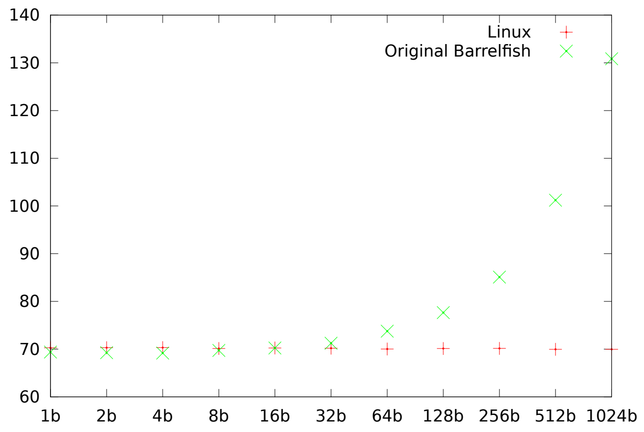
Chapter 6

Evaluation

The following section presents a lot of benchmarking and profiling data to illustrate how the network stack performs before and after optimization, and also to show how different optimizations affect the latency.

6.1 Comparison with Linux

In comparison to Linux the original Barrelfish network stack does not look too bad for small packets, but for large packets the latency increases really bad, but as discussed above this is to be expected since the receive buffers are mapped non cacheable. In Linux there are almost no variations at all on this machine for different packet sizes.



6.2 Optimizations

In the following section the implemented optimizations and their impact on latency will be measured and analyzed. The measurements were taken using the same procedure as described above, using 64 byte TCP packets we started out with a latency of **76.6μs**, echoserver running on gottardo and net_latency running on sbrinz1.

6.2.1 Effect of CPU cache on latency

The results presented bellow contain the same phases for packet transport displayed in tables 4.1 and 4.2. Those are the results after the descriptor rings and receive buffers in the driver have been marked as cacheable and flushing the packet contents has been reduced to a mfence instruction, the round trip time is reduced by 11.4μs to **65.2μs**. First the transmit phases that lasts a total of 7.4μs (2μs less than before):

42%	3.13μs	Remove packet from consumer queue
31%	2.32μs	Context switch to driver for notification
14%	1.07μs	Protocol processing in lwip
11%	0.78μs	Enqueue packet in consumer queue
2%	0.12μs	Flush packet content from cache

Now the receive part takes 11.2μs, here 3.6μs were saved:

58%	6.49μs	Enqueue packet in consumer queue
14%	1.51μs	lwip protocol processing
12%	1.36μs	Context switch to application for notification
8%	0.86μs	Apply filters to select destination buffer
5%	0.58μs	Remove packet from consumer queue
3%	0.37μs	Copy packet into application buffer

Those results are about what was expected, receive benefited in the copy and filter operations that are really expensive without caching, while on the transmit side only the flushing of packet contents takes less time, the rest stays more or less the same.

6.2.2 Effect of bulk transfer optimizations

After the cache problem was fixed, the next thing we took a closer look at was the bulk transfer interface code. After some profiling and a lot of experimenting the overall round trip time reduced to **46.4μs**, an improvement of no less than 18.8μs. Transmit reduces to 5.5μs, an additional 1.9μs saved:

48%	2.64μs	Context switch to driver for notification
20%	1.12μs	Remove packet from consumer queue
18%	0.96μs	Protocol processing in lwip
12%	0.66μs	Enqueue packet in consumer queue
2%	0.11μs	Flush packet content from cache

On the receive side too we're able to shave off another 4.3μs, resulting in 6.9μs for the software receive side:

29%	1.99μs	Enqueue packet in consumer queue
22%	1.55μs	lwip protocol processing
20%	1.36μs	Context switch to application for notification
15%	1.00μs	Apply filters to select destination buffer
9%	0.61μs	Remove packet from consumer queue
5%	0.37μs	Copy packet into application buffer

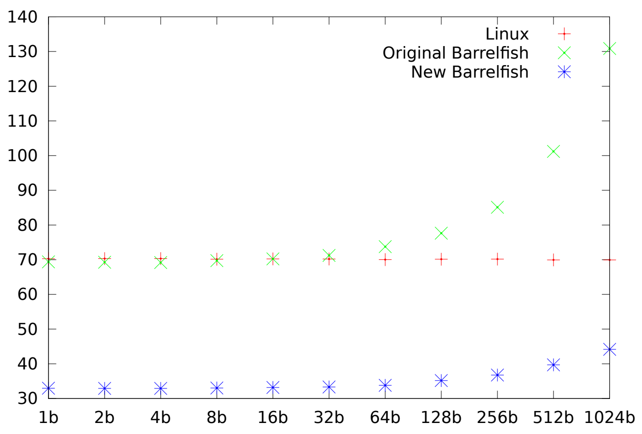
6.2.3 Effect of multiple hardware queues

The last optimization caused the most work, redesigning the network stack and using multiple hardware queues. This also caused some internal data structures in the ethersrv library to become simpler, which could explain some of the non obvious performance gains.

Here the combination of the bug requiring that all network processes run on the same core and the fact that polling is used caused some noise in the benchmark (multiple queue drivers running on same core, both waiting until their time slice gets taken away), therefore not the average results are used, but the median of the results, to get more meaningful results. Anyway using the new network stack architecture we arrive at a final round trip time of **36.6μs**, another 9.8μs saved. Not unexpected transmit did not vary at all and stayed at 5.5μs, while on the receive side 1.5μs were saved, that corresponds more or less to filtering and copying, resulting in 5.4μs.

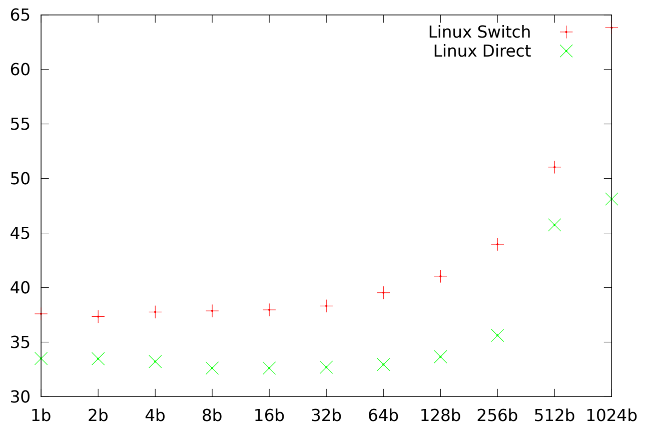
6.2.4 Summary

The following graph provides a before after comparison, displaying both the original and optimized Barrelfish results, as well as the Linux result as a basis for comparison:



6.3 Influence of Ethernet switch

As pointed out by Rumble et al. switches can introduce quite some latency. To measure the latency introduced by the switch the machines sbrinz1 and ziger1 were connected directly, and then the same TCP benchmark used for the Linux results above was used to get numbers for multiple packet sizes. As it turns out the latency introduced is indeed significant, for 64 bytes of TCP payload an additional latency of 6.6μs is introduced:



Since this latency is introduced independent of the endpoints, this leaves us with an effective machine to machine latency of **30.0μs**.

Chapter 7

Conclusion

As it turns out the latency can be reduced a lot by looking at the software side of network packet processing. Using some optimizations, architectural redesign and assistance from the network controller, the total round trip time for a 64 byte TCP packet could be reduced to less than 50% from the first simple implementation. Also some of the optimizations were independent from the 82599 network controller, meaning that other drivers also benefit. The architectural redesign also allows other drivers supporting multiple hardware queues to be plugged in with relative ease. We also saw some limitations of the 82599 controller that make it not ideal for an exokernel environment.

7.1 Future work

Some work will need to be invested to remove the limitations discussed in the implementation chapter, to make the whole architecture useful in practice. Also some further optimizations that were not implemented were described. In particular the optimization for mapping the receive queue read only into application memory should save quite some time, a rough estimate using the results presented in the previous chapter suggests that about 4μs are spent on bulk transfer, IDC and context for receive, if that kind of time is saved on both sides, this could reduce the round trip time by as much as 8μs. Reducing IDC overhead could also help lowering network latency, but probably not as much as other approaches.

Another area where some research would help is the interface between network controller and software, numerous user level networking approaches [5] [23] [15] should provide a reasonable starting point. But in connection with network engines on the CPU die onloading the demultiplexing and more features onto a general purpose core could also be interesting.

Bibliography

[1] A. Baumann. Inter-dispatcher communication in barrelfish. Technical Report 011, 2010.

[2] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP*, volume 9, pages 29–44, 2009.

[3] A. Baumeler and R. Voigt. bdfmux – barrelfish demultiplexer. Distributed systems lab, 2009.

[4] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(2):175–198, 1991.

[5] P. Druschel, L.L. Peterson, and B.S. Davie. Experiences with a high-speed network adaptor: A software perspective. *ACM SIGCOMM Computer Communication Review*, 24(4):2–13, 1994.

[6] A. Dunkels. Minimal tcp/ip implementation with proxy support. Sics research report, 2001.

[7] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *Micro, IEEE*, 18(2):66–76, 1998.

[8] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 50–59. IEEE Computer Society, 2005.

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

[10] J. Liedtke. On-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, 1995.

[11] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope. Getting 10 gb/s from xen: Safe and fast device access from unprivileged domains. In *Euro-Par 2007 Workshops: Parallel Processing*, pages 224–233. Springer, 2008.

[12] 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*, pages 5–5. USENIX Association, 2003.

[13] D. Mosberger, L.L. Peterson, P.G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. *ACM SIGCOMM Computer Communication Review*, 26(4):73–84, 1996.

[14] A. Ortiz, J. Ortega, A.F. Díaz, and A. Prieto. Network interfaces for programmable nics and multicore platforms. *Computer Networks*, 54(3):357–376, 2010.

[15] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 67–76. IEEE, 2001.

[16] K.K. Ram, J.R. Santos, Y. Turner, A.L. Cox, and S. Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 61–70. ACM, 2009.

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

[18] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *Computer*, 37(11):48–58, 2004.

[19] L. Rizzo. netmap: fast and safe access to network adapters for user programs. 2011.

[20] S.M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J.K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 11–11. USENIX Association, 2011.

[21] P. Shinde. Barrelfish bulk transfer. Technical Report 014, 2011.

[22] A. Singhani and Kuz I. Inter-dispatcher communication in barrelfish. Technical Report 013, 2011.

[23] T. Von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 40–53. ACM, 1995.