Technische Universität München
Department of Informatics
Chair for Network Architectures and Services

Comparison of Memory Mapping Techniques for High-Speed Packet Processing

Master's Thesis in Informatics

Chair for Network Architectures and Services
Department of Informatics
Technische Universität München

by

Sebastian Gallenmüller

Technische Universität München
Department of Informatics
Chair for Network Architectures and Services

Comparison of Memory Mapping Techniques for High-Speed Packet Processing

—

Vergleich von Speichermappingtechniken zur Hochgeschwindigkeitspaketverarbeitung

Master's Thesis in Informatics

Chair for Network Architectures and Services
Department of Informatics
Technische Universität München

by

Sebastian Gallenmüller

Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisors:	Florian Wohlfart, M. Sc.
	Daniel Raumer, M. Sc.
Submission date:	September the 11th, 2014

Abstract:

Network stacks currently implemented in operating systems can no longer cope with the high packet rates offered by 10 GBit Ethernet. Thus, frameworks were developed claiming to offer a faster alternative for this demand. These frameworks enable arbitrary packet processing systems to be built from commodity hardware handling a traffic rate of several 10 GBit interfaces, entering a domain previously only available to custom-built hardware.

Three examples for this kind of framework are netmap, PF_RING ZC, and Intel DPDK. The common structure and techniques shared by these frameworks are presented. In addition, differences in implementation and architecture are closely examined. API and usage of these frameworks are demonstrated with code examples.

Moreover, a model is developed during this work, demonstrating the fundamental relationship between costs and performance of these packet processing frameworks. This model describing an arbitrary complex packet processing application allows building tailor-made systems for a specific packet processing task. As the performance to be expected from available equipment can now be predicted using the model the hardware for the packet processing application can be chosen accordingly. The model is verified with measurements and the behavior of each framework is investigated in different scenarios.

Measurements begin with a basic packet forwarding scenario based on a network consisting of three servers connected via 10 GBit Ethernet with one server acting as a packet forwarder. Forwarding is done using an application running on top of a packet processing framework. This application is gradually modified during this thesis to demonstrate more complex packet processing scenarios. Thus, factors influencing the throughput like CPU load, memory connection, or the number of packets processed simultaneously can be examined more closely.

The insights gained by this thesis can be applied to realize packet processing applications by choosing the framework best fit for the application's requirements. In addition, the performance to be expected from this kind of application is presented together with useful hints to reach the full potential of those frameworks.

Zusammenfassung:

Netzwerkstacks, die aktuell in Betriebssystemen zum Einsatz kommen, sind den Paketraten, wie sie 10 GBit Ethernet bietet, nicht länger gewachsen. Um dieser Anforderung nachzukommen, entstanden Frameworks als eine leistungsfähigere Alternative zu den Netzwerkstacks. Diese Frameworks ermöglichen die Erstellung beliebiger Anwendungen zur Paketverarbeitung, die auf gewöhnlicher Hardware den Netzwerkverkehr mehrerer 10 GBit Schnittstellen verarbeiten können und damit in Bereiche vordringen, die ehemals spezialisierten Hardwaresystemen vorbehalten war.

Mit netmap, PF_RING ZC und Intel DPDK werden drei Vertreter dieser Frameworks näher untersucht. Vorgestellt werden der gemeinsame Aufbau und Techniken, die von diesen Frameworks eingesetzt werden, ebenso wie die Unterschiede in Implementierung und Architektur. Anhand von Programmbeispielen werden API und Benutzung vorgestellt.

Ferner wird in dieser Arbeit ein Modell entwickelt, das den grundlegenden Zusammenhang zwischen den aufgewandten Kosten und der erreichbaren Leistung dieser Paketverarbeitungsframeworks aufzeigt. Dieses Modell, mit dessen Hilfe beliebig anspruchsvolle Paketverarbeitungsvorgänge dargestellt werden können, erlaubt die Herstellung von Systemen, die perfekt auf die Anforderungen der Paketverarbeitung angepasst sind. Da die zu erwartende Leistung verfügbarer Hardware nun eingeschätzt werden kann, kann auch die Hardware für die Paketverarbeitungsprogramme passend ausgewählt werden. Dieses Modell wird mittels Messungen verifiziert und das Verhalten der Frameworks wird in verschiedenen Szenarien untersucht.

Das grundlegende Szenario der Messungen ist eine Paketweiterleitung. Die Tests finden in einem 10 GBit Ethernet Netzwerk statt, das aus drei Servern besteht, wobei ein Server die Pakete weiterleitet. Die Paketweiterleitung wird von einer Anwendung durchgeführt, die auf jeweils eines der Frameworks zur Paketverarbeitung aufsetzt. Diese Anwendung wird im Laufe der Arbeit schrittweise modifiziert, um komplexere Paketverarbeitungsvorgänge darstellen zu können. Dadurch können Faktoren, die den Paketdurchsatz beeinflussen, wie zum Beispiel die CPU Leistung, die Speicheranbindung oder auch die Anzahl gleichzeitig verarbeiteter Pakete näher untersucht werden.

Die Erkenntnisse, die in dieser Arbeit gewonnen werden, können dazu dienen Anwendungen zur Paketverarbeitung zu entwerfen, unter Auswahl des Frameworks, das den Anforderungen der Anwendung am besten gerecht wird. Zusätzlich wird die zu erwartende Leistung einer solchen Anwendung vorgestellt, inklusive einiger Hinweise, um das volle Potential dieser Frameworks auszuschöpfen.

Contents

1. Introduction

Since the introduction of integrated circuits for computers, Moore's Law has provided the basis for advancements in computer hardware by predicting an exponential growth for the number of transistors per integrated circuit. Up until today, this development allows the integration of new features into CPUs like larger caches, additional cores, or memory controllers leading to increased processing capabilities. Advancements offered by Moore's Law are not limited to CPUs but are also used to promote the progress in other domains like digital signal processing. For network signal processing this enables the integration of more complex algorithms for filtering, error correction, or compression resulting in higher available bandwidth.

The advancements in hardware need to be accompanied by changes in software to fully utilize the new hardware features. For instance, software needs to be adopted to benefit from modern multi core architectures. This also applies for advancements in network hardware where currently a shift from 1 GBit Ethernet to 10 GBit Ethernet is happening (at least for server hardware). Network stacks that are implemented in operating systems like the Linux network stack cannot cope with the tens of millions packets per seconds offered by hardware.

Several frameworks specialized in high-speed transfer of packets have been developed claiming to fill this performance gap between hard- and software. These frameworks make use of recently added hardware features of CPUs and network processors currently not supported by regular network stacks and enable packet processing at full line rate on several 10 GBit interfaces based on commodity hardware. However, this focus on raw performance leads to the necessity of adaptations in architecture and programming interfaces compared to a regular network stack. Additional performance was gained by removing features provided by the operating system's network stack, like support for protocols like TCP or UDP.

A common misconception seems to be that these frameworks replace a general-purpose network stack. This led Intel to put an explicit note on the front page of their framework's website stating that DPDK is "not a networking stack" [1]. These frameworks should be seen as a high-performance but stripped down interface for packet IO rather than a substitute for a full-blown network stack.

1.1 Goals

This thesis sums up the current state of high-speed packet processing on commodity hardware. Therefore, a overview is given over related work and available solutions for handling network traffic based on PC hardware, i.e. using the network IO capability of a standard operating system or using a framework, specialized in high-performance packet processing. The three most prominent representatives of these frameworks, namely netmap [2], PF_RING ZC [3], and DPDK [4] are closely examined.

The theoretical background explaining the performance increase of those frameworks is investigated. Therefore, architectural design and acceleration techniques both common to all three frameworks are explained but also differences between them are identified.

A model is designed describing the performance of a packet processing system. Using this model the expectable throughput of each framework with the available processing power offered by hardware can be predicted.

Finally, measurements are designed and executed to verify the model but also to show the implications resulting from varying designs in different scenarios. These measurements start with a very basic forwarding scenario and get more complex when taking different batch sizes or cache effects into account, leading to a simplified software router emulation based on the investigated frameworks.

1.2 Structure

This thesis starts in Chapter 2 with an overview over the current state of the art in packet processing on commodity hardware looking at literature and relevant hardware and software features.

After that Chapter 3 presents the main acceleration techniques of high-speed packet processing frameworks.

The following three chapters present the frameworks netmap (Chapter 4), PF_RING ZC (Chapter 5), and DPDK (Chapter 6). Each of these three chapters shows the architecture and demonstrates the API with code examples of the respective framework.

Chapter 7 explains the test setup used to conduct the measurements of this thesis.

In Chapter 8 a model is designed to describe a generic packet processing application. During this chapter the model is refined to describe a application realized in one of the mentioned frameworks.

An evaluation of the three frameworks is presented in Chapter 9. There the model is verified by various measurements starting with a simple forwarding scenario. These simple forwarders are gradually modified during the course of this chapter to emulate more complex scenarios like routing.

2. State of the Art

This chapter discusses the current state of the art in packet processing on commodity hardware by investigating related work. Moreover, hardware features enabling high-performance packet processing are presented. This chapter ends with a description of software systems for packet transmission currently offered by operating systems.

2.1 Related Work

The traditional way of packet processing uses methods and interfaces offered by operating systems. During the last few years a number of frameworks emerged that allow to shift the packet processing to user space.

Packet Processing using Standard Operating Systems

Kohler et al. [5] developed a modular software router called Click. Click is an example for a packet processing application using the standard interfaces provided by Linux. This router can either run as a user space application but offers better performance if loaded as a Linux kernel module. Click is highly adaptable through its modular architecture that allows for the free combination of modules needed for a specific packet processing task. An architecture which makes Click a software platform with possibilities similar to those of specialized packet processing frameworks.

A publication by Bolla et al. [6] presents an evaluation of software routers running on Linux based on commodity hardware. Typical problems of packet processing in Linux are addressed. Furthermore, performance and latency of software routers based on Linux and Click are evaluated. This paper provides performance figures achievable with conventionally designed applications like this thesis does for high-performance frameworks.

Another notable example for a packet processing application is Open vSwitch introduced by Pfaff et al. [7]. This software switch is able to connect virtual machines allowing a high degree of flexibility due to its programmable rule set. Rules are specified in OpenFlow [8] and allow adopting this switch to different packet processing scenarios, which makes this application a competitor for the specialized frameworks. A comparison of switching performance between Open vSwitch and a configured Linux kernel is included. For performance reasons part of Open vSwitch is realized as a kernel module designed to handle the main load of the packet processing task.

Packet Processing using Specialized Frameworks

Deri, the developer of PF_RING, has published several papers about the predecessors of the framework, which is called PF_RING ZC in its most recent version.

A publication from 2003 by Deri [9] presents a software solution purely designed for fast packet capture. To buffer packets, a data structure called PF_RING is used. Basic techniques to accelerate packet reception are presented. It includes a rudimentary comparison of packet capture rates between the newly designed framework and the Linux kernel.

One year later Deri [10] introduced a packet processing framework called nCap capable of high-speed packet capture and transmission. He explains the basic design decisions allowing the high-performance of nCap, its API, and includes one test showing the performance gain compared to Linux.

Rizzo [2] presents his solution of a packet processing framework, which is called netmap. He analyzes shortcomings of the network stack implemented by BSD and introduces netmap. The acceleration techniques, the API, and the architecture of this framework are presented in detail. In addition, packet generation and forwarding performance is evaluated. A survey compares different forwarding scenarios including solutions based on BSD, Click, Open vSwitch, or netmap alone.

A joint publication by Deri and Rizzo [11], the authors of netmap and PF_RING, presents a comparison between their respective frameworks. This paper compares the different organizational structures and architectures of both frameworks but it lacks comparative performance measurements. However, the PF_RING version presented in this publication is not the ZC version evaluated by this thesis but its predecessor.

Wind River [12] published a white paper giving an overview over the packet processing framework DPDK. The process of packet processing in Linux is explained and compared to DPDK. In addition, basic acceleration techniques and an overview over the most important libraries included in DPDK are presented. Beside DPDK, commercial products based on DPDK provided by Wind River are presented. This paper does not contain any performance figures or comparisons to other frameworks.

The basics of network stacks and different solutions for high-performance packet processing are discussed by García-Dorado et al. [13]. The frameworks investigated are PF_RING DNA, the predecessor of PF_RING ZC, netmap, PacketShader [14], and PFQ [15]. The theoretical background of network stacks and frameworks are explained in detail. Descriptions and measurements presented are limited to the reception of packets; packet sending is neither explained nor measured.

Other Notable Examples for Packet Processing Frameworks

Some frameworks are not part of a closer investigation by this thesis due to different reasons like stopped development, immaturity, or hardware dependencies.

Han et al. [14] have developed PacketShader, a high-performance software router with support for GPU acceleration. The packet engine of this router is publicly available offering similar features as the frameworks investigated by this thesis. Code is updated very rarely and there seems to be no implementation of this engine except for the use in PacketShader. Therefore, this thesis passes on a closer examination.

Another framework is presented by Bonelli et al. [15] called PFQ. The focus was the acceleration of packet monitoring systems. Therefore, PFQ was developed exclusively for packet capturing. However, it is updated and enhanced regularly, for instance, sending functionality was only added recently. As it seems to be more immature than the frameworks investigated for this thesis and due to the apparent lack of applications using PFQ this framework was not examined. Nevertheless, it may become a powerful competitor to the investigated frameworks in the near future.

An example for a vendor specific implementation of a high-performance packet processing framework is OpenOnload by Solarflare [16]. This framework implements a complete network stack with the traditional BSD socket API allowing an easy migration of network applications. The architecture of this framework is explained and the integration of state sharing mechanisms between kernel and different user processes needed for protocol handling. However, this network stack can only be used in combination with NICs provided by Solarflare. The presentation of this framework does not include any performance figures. Lacking the equipment, this framework was not assessed in this thesis.

Another open source packet processing framework is called Snabb Switch [17]. This framework is realized in the Lua Language. The focus of this framework is lowering the entry barrier by offering a simple and clean API together with a language that is easy to understand. Due to the novelty of this framework, documentation is limited and only basic performance figures are available. This is also the reason why that framework was not examined.

2.2 High-Performance Applications

Each of the examined frameworks is either part of applications or applications were adopted to make use of these frameworks.

For instance, a virtual switch called VALE was realized on top of netmap by Rizzo and Lettieri [18]. Support for netmap was also included in the software router Click [19] and in the firewall ipfw [20].

The maker of PF_RING ZC ntop offers different tools with integrated PF_RING support. One of those tools is a network traffic recorder called n2disk [21]. Moreover, a traffic monitoring tools is offered called nProbe [22].

Intel adopted Open vSwitch, a virtual switch, to use DPDK [23]. Wind River, a subsidiary Intel, offers a Linux network stack accelerated by DPDK as part of its Network Acceleration Platform [12].

2.3 Hardware Features enabling High-Speed Packet Processing

High-speed packet transfer is not only done by simply switching a slower network chip for a faster one but needs additional hardware features to make use of modern CPUs' processing power, involving caches, memory access techniques, and several cores.

Caches

Figure 2.1 shows the different memory levels of a CPU with typical values for size and access time. It also shows that access times increase from top to bottom but also the size of the respective memory level. Faster memory is more expensive and therefore slower, less costly memory is available in larger quantities.

Memory accesses of programs have certain properties. Recently used memory locations have a high probability of being used again (temporal locality). Memory locations next to recently accessed data are also likely to be accessed by a program (spacial locality). These basic locality principles are the reason why fast caches, despite offering only a fraction of the size available in main memory, hide access times effectively and efficiently.

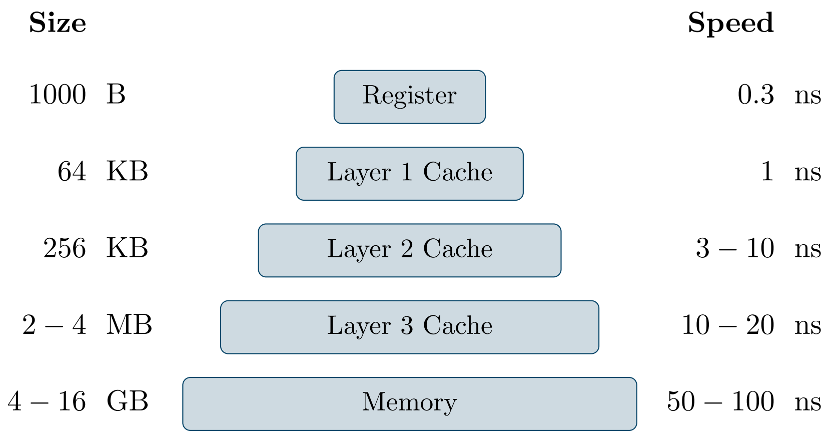

Figure 2.1: Cache hierarchy (cf. Hennessy et al. [24])

To improve access times even more, data can be put into the cache before the data is actually needed. The data to cache can be guessed with high probability due to the locality principles. This can be done in software (software prefetching) or hardware (hardware prefetching). [24]

Transition Lookaside Buffer

To access a virtually addressed memory location, the virtual address has to be converted to a physical one before it can be accessed. The previously presented principles of locality also hold for these accesses. Subsequently, a specialized cache is implemented called Transition Lookaside Buffer (TLB) containing a limited number of virtual to physical address mappings.

To access a date, the virtual address of this date is looked up in the TLB and if the TLB contains an entry for this mapping, the physical address is returned. In this case, a costly address conversion does not need to be performed. Therefore, the TLB effectively reduces memory access time. [24]

Direct Memory Access

Direct Memory Access (DMA) is a feature allowing IO devices to access memory without involving the CPU. Therefore, a CPU has to prepare a buffer in memory and sends a descriptor containing a pointer to the buffer's location to an IO device, for instance, a Network Interface Card (NIC). The NIC can use this buffer for reading or writing data independently freeing the CPU for other tasks.

Huggahalli et al. developed a technique called Direct Cache Access (DCA) allowing a NIC to put incoming packets directly into the cache of a CPU. This speeds up packet processing, as the latency introduced by a copy from memory to CPU for processing is avoided. [25]

Non-Unified Memory Access

Systems with a Non-Unified Memory Access (NUMA) are characterized by differences in memory access latencies caused by the physical memory connection. These differences are the result of the memory links in systems with more than one CPU. On those systems, every CPU is directly attached to a part of the memory via its own memory controller. However, part of the memory is only available by accessing the memory through a different CPU. To guarantee fast memory access, memory allocation in directly attached RAM should always be preferred. [26]

Receive-Side Scaling

To utilize modern multicore processors, NICs have to direct incoming packets to different queues with each queue belonging to a different CPU core. Hash functions ensure that packets belonging to the same traffic flow are distributed to their respective processing core. This technique allows scaling the traffic processing capabilities with the number of cores available on a CPU. One implementation of this technology is called Receive-Side Scaling (RSS). [26]

Offloading Features

Another possibility to save processing time on the CPU is to shift checksum validation and calculation from the CPU to the NIC. For instance, the CRC checksums used by Ethernet frames can be calculated in hardware for outgoing packets or validated for incoming traffic without CPU involvement. Similar features exist for checksums of higher-level protocols like IP and TCP freeing additional resources on the CPU. [26]

2.4 Accelerating the Host Stack

In traditional operating systems, IO like network operations are handled by interrupts. This way the processing time for polling the network device is saved freeing the CPU resources for other tasks. In addition, reaction time to incoming network traffic is kept low as interrupts are triggered immediately to inform the CPU about newly arrived packets. In cases of high network load, the interrupt driven system is slowed down as interrupts have priority over other tasks causing a system to process nothing else but the interrupts. This phenomenon is referred to as interrupt livelock. To fix this problem, the interrupts for a device are disabled during phases of high load and this device is polled instead. [27]

Linux kernel versions prior to 2.4.20 [28] suffered from this livelock problem. The problem was fixed by the introduction of the so-called New API (NAPI). Using NAPI a network device interrupts on the reception of a packet. Subsequently, interrupts are disabled for this device and the device is scheduled for polling. Incoming packets are buffered and the OS regularly polls the device. On a polling call, the device gets a chance to offer a specified number of packets (quota). If the device has more than a number of quota packets to offer, the device is again scheduled for polling. Otherwise, polling is disabled and interrupts are reenabled for this device. [29]

3. High-Speed Packet Processing Frameworks

The following section provides a detailed description of techniques used by high-performance packet processing frameworks.

3.1 Common Acceleration Techniques

Despite their different backgrounds and design philosophies netmap, PF_RING ZC, and DPDK use very similar techniques to speed up packet processing. Some features are newly introduced for packet processing like bypassing the regular network stack, avoidance of packet copying or pre-allocation of memory [2, 12, 10]. Other techniques are already used by the OS like polling for newly arrived packets or processing in batches [29].

Bypass of the Standard Network Stack

A packet processing framework completely bypasses the standard OS functions for packet handling. For the kernel and its network stack, it seems as if there is no network traffic at all. A packet processing framework has less functionality than a general-purpose network stack handling a variety of protocols, for instance, IP and TCP. Subsequently, the resources used for protocol handling are not needed effectively reducing packet processing time.

Avoidance of Packet Copying

To send packets using traditional system interfaces, the packet data must be copied from a user socket to kernel buffers or vice versa for the reception of packets. This introduces additional delay for packet processing avoided by a packet processing framework. There the packet buffers reside in a memory location shared by applications and packet processors, i.e. a network device delivers/fetches the packet data to/from the same memory location (via DMA) where an application reads/writes the packet data.

The time needed for duplicating packets depends on the packets' length. Subsequently, the per-packet processing costs are determined by the length of the packets. Avoiding the copying of packets renders the processing costs independent from the length of the packets at least for the packet processing framework. The collection of packet data in the application and the time needed for the DMA transfer still depends on the length of the packet.

Another advantage is the possibility to easily realize packet forwarding applications. Received packet data does not need to be copied by the application to the packet buffer of the outgoing interface but can simply be marked for sending.

Pre-allocation of Memory

The allocation of memory is done in the kernel of the OS. An allocation may be necessary if a packet buffer has not enough free space for newly received packets. This leads to a system call for memory allocation introducing a delay before the packets can be received. The same thing happens if a packet buffer is too small to hold the data filed for sending.

In packet processing frameworks the memory used for packet buffers is pre-allocated, i.e. memory allocation only happens in the startup phase of an application. Pre-allocation subsequently avoids the unnecessary allocation overhead during packet processing. As the amount of memory cannot be changed during runtime, the size of the allocated memory must be chosen to fit the application's needs.

Polling & Interrupts

NAPI switches to polling mode under high load to avoid blocking the system with interrupt handling. [29]

Applications running on top of high-performance frameworks use polling as these applications are specifically designed for handling high traffic loads. Interrupts are only used to wake up a sleeping application in netmap or PF_RING ZC if the respective blocking API call was used. DPDK completely avoids interrupts.

Multiple queues of NICs are supported by all the investigated frameworks to allow simultaneous polling of different cores for additional speed-up.

Batch Processing

All three frameworks provide functions for processing several packets with a single API call, i.e. the packets are processed in batches. The frameworks use the word burst synonymously. Every API call has fixed costs, for instance, to load the instructions or needed data structures to the cache. If packets are processed in batches, these fixed API costs are distributed to a number of packets. This lowers the average per-packet processing costs compared to an API where only a single packet is processed per call.

3.2 Specialized Acceleration Techniques

Some features are not used by each framework like huge pages or the avoidance of system calls.

Huge Pages

PF_RING ZC and DPDK use bigger memory pages called huge pages. The size of these pages is increased from 4 KB to 2 MB. Larger pages reduce the number of misses in the TLB and the costs for handling TLB misses. Both properties lead to shorter memory access times for data structures residing in memory allocated in huge pages. [12]

Using huge pages reduces the number of usable page entries in the TLB, for instance for a recent Intel processor from 256 entries of 4 KB pages to 32 entries of 2 MB pages. Despite the reduced number of entries, the amount of data addressed by the TLB is still larger for the bigger pages. [26]

Avoidance of System Calls

System calls handle functionality offered by an OS, like calls for initiating packet transfer. If a system call is executed by an application, the OS takes over control from the application. Subsequently, the execution of the application is paused and the context changes from kernel space to user space. There the packet transfer from or to the NIC and additional packet processing happen, e.g. protocol handling. After that, context is switched to user space and the application is put in charge. Pausing the application, context switches, loading of different cache entries, and packet processing causes high costs for these system calls.

PF_RING ZC and DPDK completely avoid these system calls for packet processing and offer user space functions replacing these calls. Therefore, these frameworks have no costs caused by system calls handling packet transfer.

The solution netmap prefers is not removing system calls for packet processing but heavily modifying them. The work done during a system call is reduced to simple checks of the packet buffer like updating the number of free/available packet slots. Costs of system calls are reduced even more by larger batch sizes resulting in fewer system calls and distributing its costs over a larger number of packets.

4. Netmap

This chapter gives an introduction into the technical background of netmap as presented by Rizzo [2]. The architecture and API of netmap are demonstrated using a piece of example code.

4.1 Overview

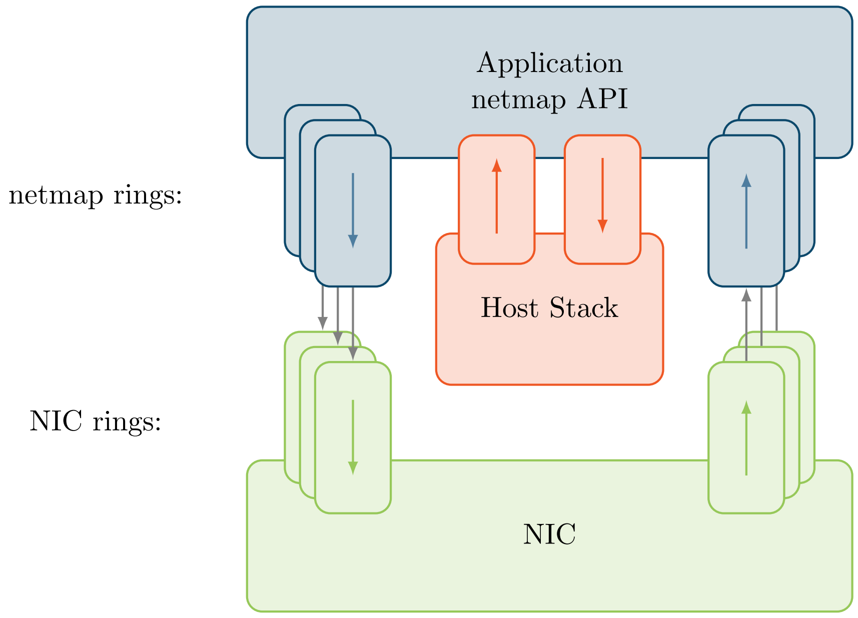

Figure 4.1: Overview over netmap (cf. Rizzo [2])

Figure 4.1 shows a simplified version of a typical application using netmap for its communication. Data is exchanged between application and NIC via data structures called netmap rings and NIC rings. Incoming and outgoing traffic uses separate rings; one or more rings may be available per direction. During the execution of an application using netmap, the host stack is cut off from communication. This state of execution is called netmap mode.

Packets may be delivered from the application to the host stack and vice versa via two special host rings. A possible use case for these rings is the realization of a packet filter. The packet filter is realized as netmap application using the high-speed interfaces to forward traffic from or to the host stack through the host rings. OS and applications do not need to be changed and can still use their traditional interfaces.

4.2 Driver

To use netmap a modified driver must be installed. A number of prepared driver patches for widely used NICs are included in the netmap repository[1]. Different 1 GBit NICs by Nvidia, Realtek, and Intel are supported using the `forcedeth`, `r8169`, or `e1000` drivers. Moreover, `virtio` driver for virtual interfaces and the `ixgbe` for 10 GBit Intel NICs drivers are available. The modifications of the drivers were kept small only needing roughly 100 lines for the driver patch files. Reducing the necessary changes to a driver simplifies the implementation of netmap support for new drivers.

As long as no netmap application is active, the driver behaves like the standard NIC driver. The NIC is managed by the OS and delivers packets to the standard system interfaces, until a netmap application is started and takes over control. Subsequently, behavior is changed to the way described in Section 4.1.

4.3 Architecture

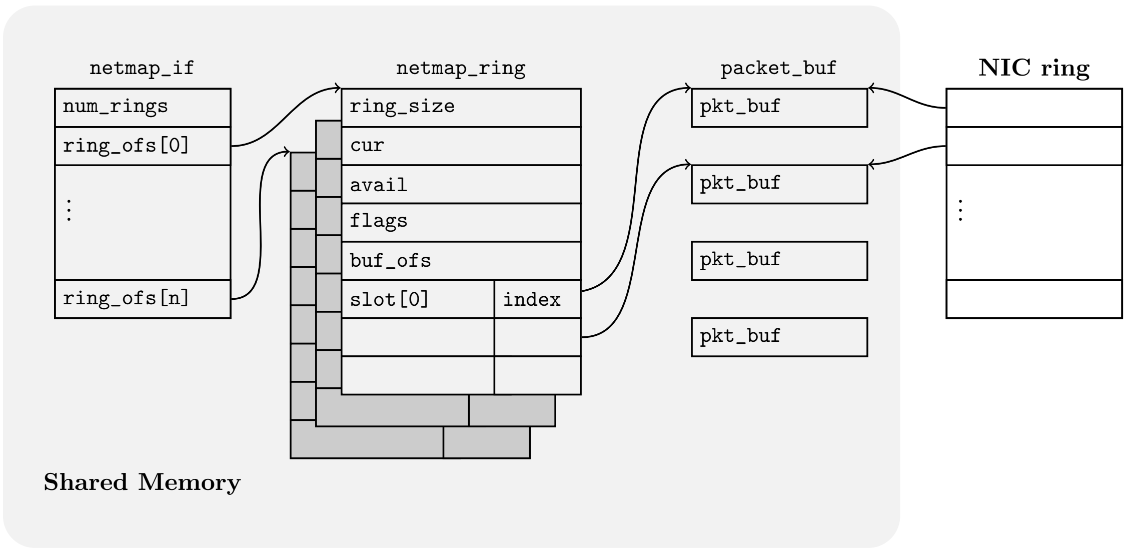

Figure 4.2: Data structures of netmap (cf. Rizzo [2])

The main data structures used by netmap are shown in Figure 4.2.

The data structures `netmap_if`, `netmap_ring` und `packet_buf` reside in a single not swappable section of memory, allocated by the kernel and shared by all processes. As every process works in its own virtual address space references are kept relative. These relative references allow every process to calculate the correct position in memory for a data structure independent from its own address space. All the data structures mentioned are allocated when entering netmap mode to avoid allocation delay during execution as explained in Chapter 3.

The data structure `netmap_if` contains read-only information about the network interface, including the number of `netmap_rings` in `num_rings` and the array `ring_ofs[]` with entries pointing to the `netmap_rings`.

[1]https://code.google.com/p/netmap/

A ring buffer called `netmap_ring` references the hardware independent buffers of a network interface. Such a ring buffers exclusively incoming traffic or exclusively outgoing traffic. Every ring has a maximum size of entries (`ring_size`), a pointer to the current read/write position (`cur`), a number of free/occupied entries (`avail`), and a field (`buf_ofs`) containing the offset between the ring and the beginning of the `paket_buf` data structure. Moreover, the ring contains the array `slot[]` with a number of `ring_size` entries, containing the entries for `packet_buf`.

Every entry of `packet_buf` contains a single network frame. These entries are shared between kernel processes, user processes, and the DMA Engine of the NIC. Every slot contains variables for keeping additional data for each packet, for instance, the length of the network packet.

4.4 API

The netmap API keeps the familiar Linux interfaces for opening a device and sending or receiving packets. The structure of a typical netmap program is given in Listing 4.1. This example forwards packets from interface `eth0` to `eth1`. To reduce the amount of code, the forwarder was simplified, for instance, the error handling or support for several `netmap_rings` was removed.

In line 2 and 3 the forwarding application opens the NICs named `eth0` and `eth1` with the following function call:

> `nm_desc* nm_open(ifname, ...)`
>
> This call is an easy way to open and configure a netmap device. It uses standard system calls that can also be called separately without the use of this convenience function.
>
> An interface name can be specified with the argument `ifname`. The prefix `netmap:` opens the Ethernet device in netmap mode. The omitted arguments can be used to configure the device or the memory mapping.
>
> As a first step the function `nm_open()` creates a file descriptor using the system call `open("/dev/netmap", O_RDWR)`. The next step configures the NIC and generates the data structures (cf. Figure 4.2) with the system call `ioctl(fd, arg1, arg2)` using the previously generated file descriptor. The argument `arg1` configures the usage of the NIC's hardware rings. In standard configuration, the value is set to `NIOCREG` to bind all available hardware rings to this single file descriptor. It is also possible to only bind a selected pair of hardware rings to the file descriptor. This allows a thread or a process to allocate an exclusive pair of rings used for Receive-Side Scaling (cf. Section 2.3). The argument `arg2` is set to the name of the NIC.
>
> Beside other values, the file descriptor and a pointer to `netmap_if` are put into the return argument of type `nm_desc*`.

The forwarder creates pointers to the `netmap_rings` to access the packets in line 5 and 6. As different processes with their own virtual address spaces use the same descriptors `rx_if` and `tx_if`, the pointers to the data structures are only kept as offsets. To calculate pointers from the offsets of their respective descriptors, the macros `NETMAP_RXRING()` and `NETMAP_TXRING()` are used.

After this initialization phase, the forwarder enters an infinite loop, which begins with a call to receive packets.

```c
nm_desc *rx_if, nm_desc *tx_if;
rx_if = nm_open("netmap:eth0", ...);
tx_if = nm_open("netmap:eth1", ...);

netmap_ring *rxring, netmap_ring *txring;
rxring = NETMAP_RXRING(rx_if->nifp, rx_if->first_rx_ring);
txring = NETMAP_TXRING(tx_if->nifp, tx_if->first_tx_ring);

while (1) {

  ioctl(rx_if->fd, NIOCRXSYNC, NULL);

  int pkts = MIN(nm_ring_space(rxring), nm_ring_space(txring));

  if (pkts) {

    int rx = rxring->cur;
    int tx = txring->cur;

    while (pkts) {

      netmap_slot *rs = &rxring->slot[rx];
      netmap_slot *ts = &txring->slot[tx];

      /* copy the packet length. */
      ts->len = rs->len;

      /* switch buffers */
      uint32_t pkt = ts->buf_idx;
      ts->buf_idx = rs->buf_idx;
      rs->buf_idx = pkt;

      /* report the buffer change. */
      ts->flags |= NS_BUF_CHANGED;
      rs->flags |= NS_BUF_CHANGED;

      rx = nm_ring_next(rxring, rx);
      tx = nm_ring_next(txring, tx);

      pkts -= 1;

    }

    rxring->cur = rx;
    txring->cur = tx;

    ioctl(tx_if->fd, NIOCTXSYNC, NULL);

  }

}
```

Listing 4.1: Simplified forwarder implemented in netmap[2]

[2]The complete example code is available at https://code.google.com/p/netmap/source/browse/examples/bridge.c

> `ioctl(fd, NIOCRXSYNC, ...)`
>
> This system call makes newly received packets available in the `netmap_rings`. Therefore, the ring's `avail` value is set to the number of received packets. The available packets are referenced by the array `slot[]` starting at the value `cur`.
>
> The arguments of this function are a file descriptor of the NIC and the value `NIOCRXSYNC` rendering this `ioctl()` call a receiving call.

Before packets can be processed, the minimum between the packets received in `rxring` and the available space in `txring` is calculated. As soon as free space is available in the `txring` and packets are ready for processing in the `rxring`, the inner loop starting at line 20 iterates over every single slot. The currently active entry of the array `slot[]` is marked by the value of `cur`. The `cur` entry of the receiving ring is put into `rs` marking the current position to read packets from. To send packets, the first free entry of `slot[]` in the sending ring is used referenced by `ts`. The length of the packet to forward is copied to the sending slot in line 26. Afterward, the pointers to the packet buffers are switched between the receiving and the sending ring. The flag fields of the `rs` and `ts` slot entries are set to `NS_BUF_CHANGED` marking the change of ownership of these packet buffers. As soon as the inner loop ends, the packets processed are filed for sending with the following call:

> `ioctl(fd, NIOCTXSYNC, ...)`
>
> This system call makes packets available for the NIC to send. Therefore, the packets must be available in the packet buffers referenced by the slots of a `netmap_ring`. On completion of `ioctl()` the `avail` value is updated to indicate the number of free packet buffers referenced by `slot[]` for sending. The value `cur` marks the position of the first free packet buffer in `slot[]`.
>
> The arguments of this function are a file descriptor of the sending NIC and the value `NIOCTXSYNC` rendering this `ioctl()` call a sending call.

Blocking Send/Receive

The previously presented send/receive calls based on `ioctl()` are non-blocking, i.e. a call returns even if no work was done and `avail` value was not updated. In addition to these, the API offers two additional calls for sending and receiving packets with a blocking behavior:

> `poll()/select():`
>
> These two system calls process the `netmap_rings` the same way as the `ioctl()` calls would. The only difference is that these blocking calls do not return before the `avail` value has changed, which happens if new slots for transmit are available or new packets have arrived.

Differences to Standard System Calls

The traditional system calls for packet transmission and reception do more work than the respective calls of the netmap framework. For instance, memory needs to be allocated and packets have to be copied. These operations are not necessary in netmap. Therefore, the work to be done during system calls is kept very small making the system calls less expensive. The steps performed during these modified system calls involve:

- Validation of `cur/avail` values and the contents of the respective slots (lengths, buffer indices).
- Synchronization of content between hardware rings and `netmap_rings` and informing NICs about newly available buffers or packets ready to transmit.
- Updating the `avail` value of a `netmap_ring`.

These checks increase the robustness of netmap against wrong data originating from the user application. They offer a basic protection from crashes during system calls.

5. PF_RING

In this chapter, an overview over PF_RING is presented. After an introduction into the different versions of PF_RING, the youngest and most powerful version called PF_RING ZC is explained in more detail. Architecture and API are described based on publications of Deri [10, 11] and the user manual [30].

5.1 Overview

A typical application using PF_RING is depicted in Figure 5.1. The application uses a library to access the packet buffers called PF_RING. Separate rings are available for sending and receiving packets. The NIC driver uses the same rings for accepting packets or offering newly arrived packets.

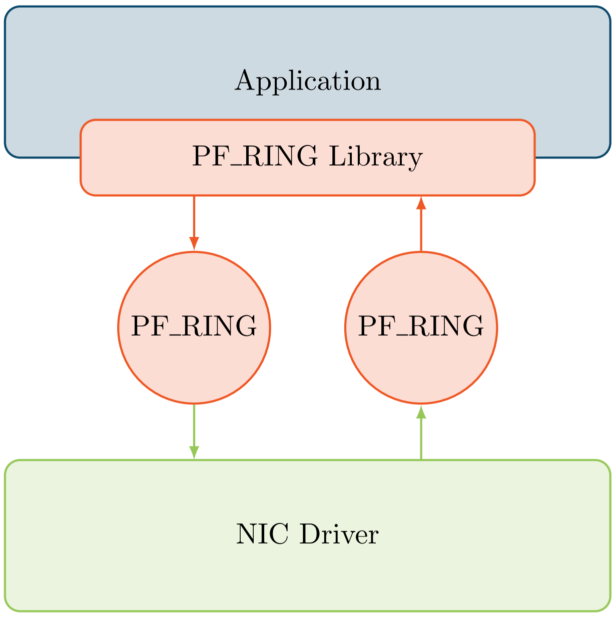

Figure 5.1: Overview over PF_RING (cf. PF_RING user manual [30])

5.2 Driver

Over nearly one decade, various versions of PF_RING were developed with only slightly different names, but great differences in performance and architecture. While the basic packet buffer of PF_RING was kept throughout different versions, the feature set and the performance of driver and library differ from version to version.

Discontinued Drivers

The first option is to use PF_RING in combination with standard Linux NIC drivers. This offers great flexibility as every driver supported by Linux can be used, but the performance gain using PF_RING is low.

Using an optimized driver version called PF_RING aware driver accelerates the transmission of packets from NIC to ring for increased performance. However, drivers have to be adapted.

Another discontinued diver version was called TNAPI. Specifically built for monitoring purposes, it supports only packet reception and distribution of packets to several processing threads.

The highest performance is gained by using a specialized driver transmitting packets directly into and from user space without the need to copy packets. This version of PF_RING was called DNA and had a library called Libzero offering convenient functions for packet processing.

PF_RING ZC

The most recent version of PF_RING is called PF_RING ZC.

The drivers delivered with the ZC version combines the features of the PF_RING aware drivers and the DNA drivers. As long as the DNA feature is not used this driver behaves like a standard Linux driver. With the `transparent_mode` option, aware drivers can copy a packet into the PF_RING buffer and deliver an additional copy to the standard Linux interfaces. This configuration offers a lower performance than using the DNA feature of the ZC driver. If an application uses this feature, packets bypass the kernel and standard interfaces for increased performance.

Modified drivers for Intel cards (`e1000e`, `igb`, and `ixgbe`) are included in PF_RING ZC.

For this thesis only the ZC version of PF_RING is investigated, as it is the only high-performance version currently under development.

5.3 Architecture

The source code of the PF_RING ZC library is not publicly available. Therefore, only a very coarse overview is given over the architecture.

PF_RING ZC offers basic building blocks for an easy implementation of applications. One important feature is that the ZC library transmits packet between queues, threads, and processes without the need to copy the packet.

Another important structure is the `pfring_zc_cluster`. This cluster is identified by an ID and used for grouping threads or processes to share packets amongst them.

To transmit packets between packet processing entities, e.g. NICs, processes, or threads `pfring_zc_queue` is used. Queues can be combined to form a `pfring_zc_multi_queue`.

For the distribution of packets to different entities, `pfring_zc_worker` is used. Two different kinds of workers are offered a balancer and a fanout. The balancer can distribute packets coming from one of several incoming queues to one of several outgoing queues. The distribution function can be programmed. Fanout uses multiqueues as an outgoing queue to distribute an incoming packet to every member connected to the multiqueue.

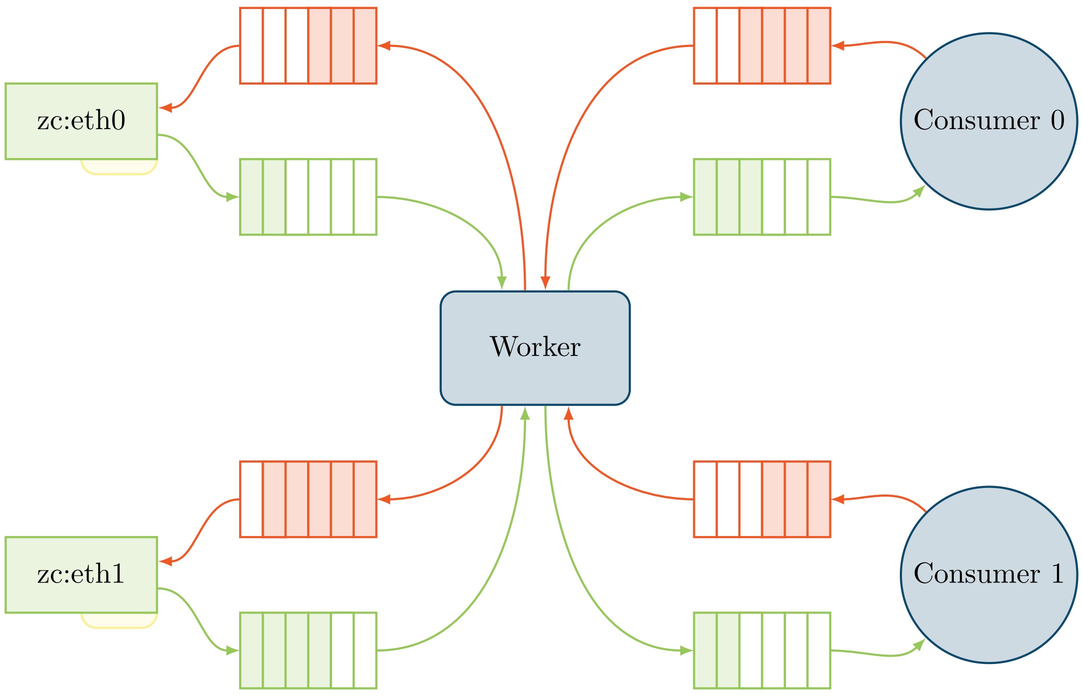

Figure 5.2: PF_RING ZC cluster (cf. PF_RING user manual [30])

The `pfring_zc_buffer_pool` is a memory pool, which is used, for instance, by the workers.

Figure 5.2 shows an application using a PF_RING ZC cluster. It consists of two NICs, a worker for packet distribution, and two packet consumers. The NICs, worker, and consumers use separate `pfring_zc_queues` for submitting and accepting packets. The worker can be realized either as balancer to distribute a packet to one of the consumers or as fanout to deliver every packet to both consumers using a `pfring_zc_multi_queue`. The consumers can be realized as threads or as separate processes to do arbitrary packet processing. In the displayed example, packets are delivered back to the worker after being processed by the consumers. Other solutions are possible including the packets being put into a queue connected to another consumer or the packets being processed without any output.

5.4 API

The API of PF_RING ZC is different from the standard Linux interface. With the transition to the ZC version, the API was reworked to provide a clean and consistent interface. One feature of the API is convenient multicore support as shown in Figure 5.2.

The structure of a simple forwarder using PF_RING ZC is given in the Listing 5.1. The packets are forwarded from `eth0` to `eth1` in a dedicated thread. Because of the simplicity of the forwarding, only one thread without any workers for balancing suffices. To reduce the amount of code, the forwarder was simplified, for instance, the error handling was removed. Errors are indicated by negative return values for functions returning integers or `NULL` for functions returning structs like `pfring_zc_queue`.

The first step in the PF_RING ZC sample forwarder is the creation of the cluster.

`pring_zc_create_cluster(id, len, meta_len, num_buff, numa, huge_mnt):`

This call creates a new cluster returning the cluster handle on success.

```c
#define BURST_SZ 32

pfring_zc_cluster *zc;
pfring_zc_queue *inzq, *outzq;
pfring_zc_pkt_buff *buff[BURST_SZ];

int start_forwarder() {

  zc = pfring_zc_create_cluster(
      1,     /* cluster id */
      1536,  /* buffer length */
      0,     /* buffer metadata length */
      65537, /* num of buffers */
      ...);

  /* prepare buffers and queues */
  for (int i = 0; i < BURST_SZ; ++i)
    buff[i] = pfring_zc_get_packet_handle(zc);
  inzq = pfring_zc_open_device(zc, "zc:eth0", rx_only, 0);
  outzq = pfring_zc_open_device(zc, "zc:eth1", tx_only, 0);

  /* start forwarder */
  pthread_create(..., forwarder_thread, ...);

}

void* forwarder_thread() {

  int tx_queue_not_empty = 0;

  while(1) {

    /* receive packets */
    int num_pkts = pfring_zc_recv_pkt_burst(inzq, buff, BURST_SZ, 0);

    if(num_pkts > 0) {

      /* send packets */
      pfring_zc_send_pkt_burst(outzq, buff, num_pkts, 0);
      tx_queue_not_empty = 1;

    } else {

      if (tx_queue_not_empty) {

        pfring_zc_sync_queue(i->outzq, tx_only);
        tx_queue_not_empty = 0;

      }

    }

  }

}
```

Listing 5.1: Simplified forwarder implemented in PF_RING ZC[1]

[1] The complete example code is available at https://svn.ntop.org/svn/ntop/trunk/PF_RING/userland/examples_zc/zbounce.c?p=7718

> The cluster is identified by a unique `id`. Buffer size is determined by `len`. A buffer must be large enough to hold a maximum sized packet. For optimal usage of caching, a multiple of 64 should be chosen. The `meta_len` value determines the size of the buffers containing metadata. Argument `num_buff` gives the number of buffers in total, which, for example, are used by queues. To allocate the buffers in a memory space directly connected to the used CPU socket, the NUMA node id can be entered in `numa`. The mount point used by huge pages can be set manually or detected automatically if `huge_mnt` is set to `NULL`.

The next step is to create the buffer, which has to contain at least a number of `BURST_SZ` entries. After that, the NICs are prepared by opening a read-only queue on interface `eth0` and a write-only queue on interface `eth1`. The prefix `zc:` is used to signal the driver to switch to ZC mode.

> `pfring_zc_open_device(cluster_handle, if_name, q_mode, flags):`
>
> This function call opens a network device returning a queue to this interface on success.
>
> A device is opened by using the `cluster_handle`. The interface is identified by its `if_name` and the direction of the queue (input or output) is determined by the `q_mode`. Additionally, `flags` can be specified.

After initialization is finished, the actual forwarding thread is started, which enters an infinite loop trying to read the queue for newly arrived packets. The number of received packets is returned and these packets are available in `buff`.

> `pfring_zc_recv_pkt_burst(queue, pkt_handles, max_batch, block):`
>
> To receive a batch of packets, this function is used.
>
> For the reception of packets, the `queue` must be specified along with an array of buffers (`pkt_handles`). This buffer must contain at least `max_batch` entries, as this is the maximum number of packets, which can be received during a call. It is also possible to receive fewer packets. The return parameter reports the number of newly received packets. If the `block` flag is set, this call is a blocking call, i.e. it sleeps until at least one packet is received. Without blocking, no interrupts are used but for waking up the caller from a blocking call, an interrupt is triggered.

On a successful reception of packets, the buffer is forwarded to the outgoing queue.

> `pfring_zc_send_pkt_burst(queue, pkt_handles, max_batch, flush):`
>
> For sending packets in batches, this function is called.
>
> The first three arguments have the same purpose as the parameters of the receiving call with the exception that an outgoing queue must be used. If the flag `flush` is set, the packets residing in the queue are flushed to the recipient immediately.

After filing packets for sending, the ingoing queue is read again. If no new packets have arrived and the outgoing queue is marked as not empty, the queue is flushed by executing the function `pfring_zc_sync_queue` on the outgoing queue.

6. Data Plane Development Kit

This chapter presents Intel's Data Plane Development Kit (DPDK). Information about DPDK its driver, its architecture, and its API are taken from the extensive documentation offered by Intel [31, 32].

6.1 Overview

The main goal of DPDK is not only to accelerate the packet IO mechanisms but also to offer a collection of libraries and components for comprehensive packet processing. A DPDK based application can select the libraries needed for its processing task. The most basic libraries are presented in Figure 6.1 with the most important library called *Environment Abstraction Layer* (`rte_eal`) on top. The EAL initializes and manages the different packet processing threads. The other libraries displayed use functionality offered by the EAL.

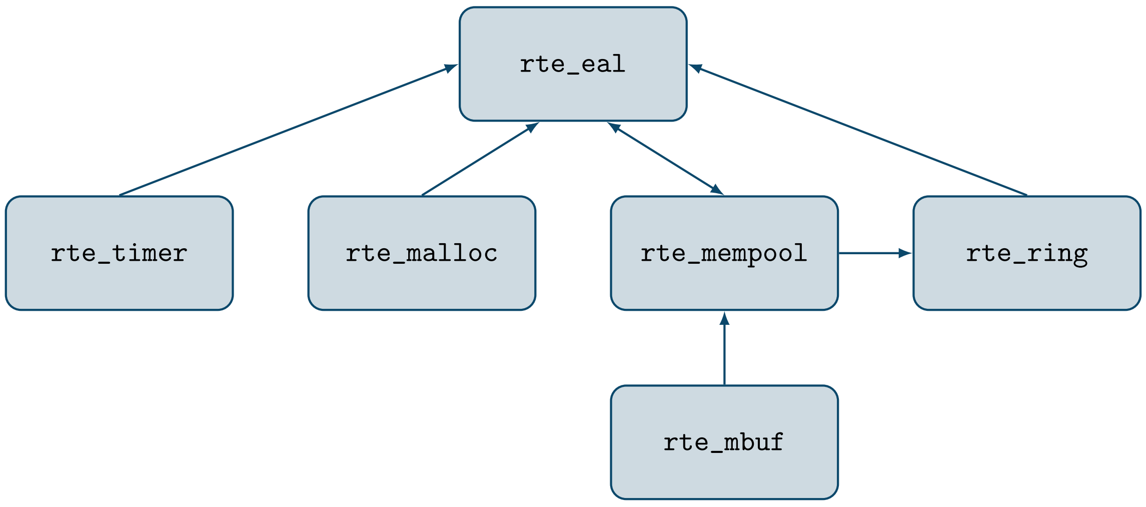

Figure 6.1: Basic libraries offered by DPDK (cf. DPDK programmer's guide [31])

6.2 Driver

Drivers for Intel's 1 GBit and 10 GBit NICs are included. In addition, a number of virtualized drivers were adapted to DPDK. The driver is called Poll Mode Driver (PMD) as packets only can be polled without the use of any interrupts. To use the devices with this driver, the NICs have to be bound to the driver. The Linux kernel cannot use these interfaces as long as they are bound to the PMD. To bind/release NICs to/from the PMD, a script is available called `pci_unbind.py`. [33]

This driver is based on the `igb_uio` driver. UIO drivers are specially designed drivers aiming to do most of the processing in user space. An UIO driver still has a part of its code realized as kernel module but its tasks are reduced. For instance, the kernel module only initializes devices or acts as interface for the PCI communication. [34]

6.3 Architecture

The basic architecture is shown in Figure 6.1. The main library called *Environment Abstraction Layer* (`rte_eal`) offers a generic interface for applications hiding implementation details of hardware and OS from the application or other libraries. Moreover, the EAL initializes application and resources, e.g. memory, PCI components, and timers.

A *Timer Manager* (`rte_timer`) allows to call functions asynchronously, once or periodical. The precise reference time is provided by the EAL.

A component called *Memory Manager* (`rte_malloc`) allocates memory in huge pages to use the memory more efficiently.

The *Network Packet Buffer Management* (`rte_mbuf`) can create or destroy packet buffers. These buffers may be adopted to hold other data. All buffers are created at startup of an application.

The *Ring Manager* (`rte_ring`) offers lock free ring buffers of fixed size. These rings support several producers and several consumers. They can be used as a communication facility between different threads.

The *Memory Pool Manager* (`rte_mempool`) is used to store objects (usually the previously described mbufs) in rings. Additionally a per-core object buffer is provided by this library. To optimize memory access, objects are distributed equally over the available memory channels.

DPDK also provides libraries for higher-level functionality for example, a library called `librte_hash` can be used to provide hash based forwarding of packets. In addition, a longest prefix matching algorithm is included in DPDK (`librte_lpm`). The library `librte_lpm` provides useful methods for handling protocols like IP, TCP, UDP, and SCTP. For instance, header structure information is given or the protocol numbers used by IP.

6.4 API

The API of DPDK has defined its own interface for packet processing. This is by far the most powerful API of the investigated frameworks. Therefore, even a simplified forwarder without error handling is split in two phases the initialization and the forwarding phase shown in Listing 6.1 and Listing 6.2. In the terminology of DPDK, threads are called lcore due to running on a dedicated core. The main thread or master lcore executes the first phase and the forwarding is done on a regular lcore.

The first step of the initialization phase is the initialization of the EAL by entering the command line arguments to the initialization call.

```c
struct rte_mempool* pktbuf_pool = NULL;
int rx_port_id = 1;
int tx_port_id = 0;

int main(int argc, char **argv) {

  rte_eal_init(argc, argv);

  pktbuf_pool =
    rte_mempool_create("mbuf_pool", NB_MBUF,
                       MBUF_SIZE, 32,
                       sizeof(struct rte_pktmbuf_pool_private),
                       rte_pktmbuf_pool_init, NULL,
                       rte_pktmbuf_init, NULL,
                       rte_socket_id(), 0);

  rte_pmd_init_all();
  rte_eal_pci_probe();

  prepare_dev(rx_port_id);
  prepare_dev(tx_port_id);

  rte_eal_remote_launch(run, NULL, 1);
  rte_eal_wait_lcore(1);

  return 0;
}

static void prepare_dev(int id) {

  int sock_id = rte_eth_dev_socket_id(id);

  rte_eth_dev_configure(id, 1, 1, &port_conf);
  rte_eth_rx_queue_setup(id, 0, 128, sock_id, &rx_conf, pktbuf_pool);
  rte_eth_tx_queue_setup(id, 0, 512, sock_id, &tx_conf);
  rte_eth_promiscuous_enable(id);

}
```

Listing 6.1: Simplified forwarder implemented in DPDK (Initialization)[1]

[1]The complete example code is available at http://www.dpdk.org/browse/dpdk/tree/examples/l2fwd/main.c?h=1.6.0

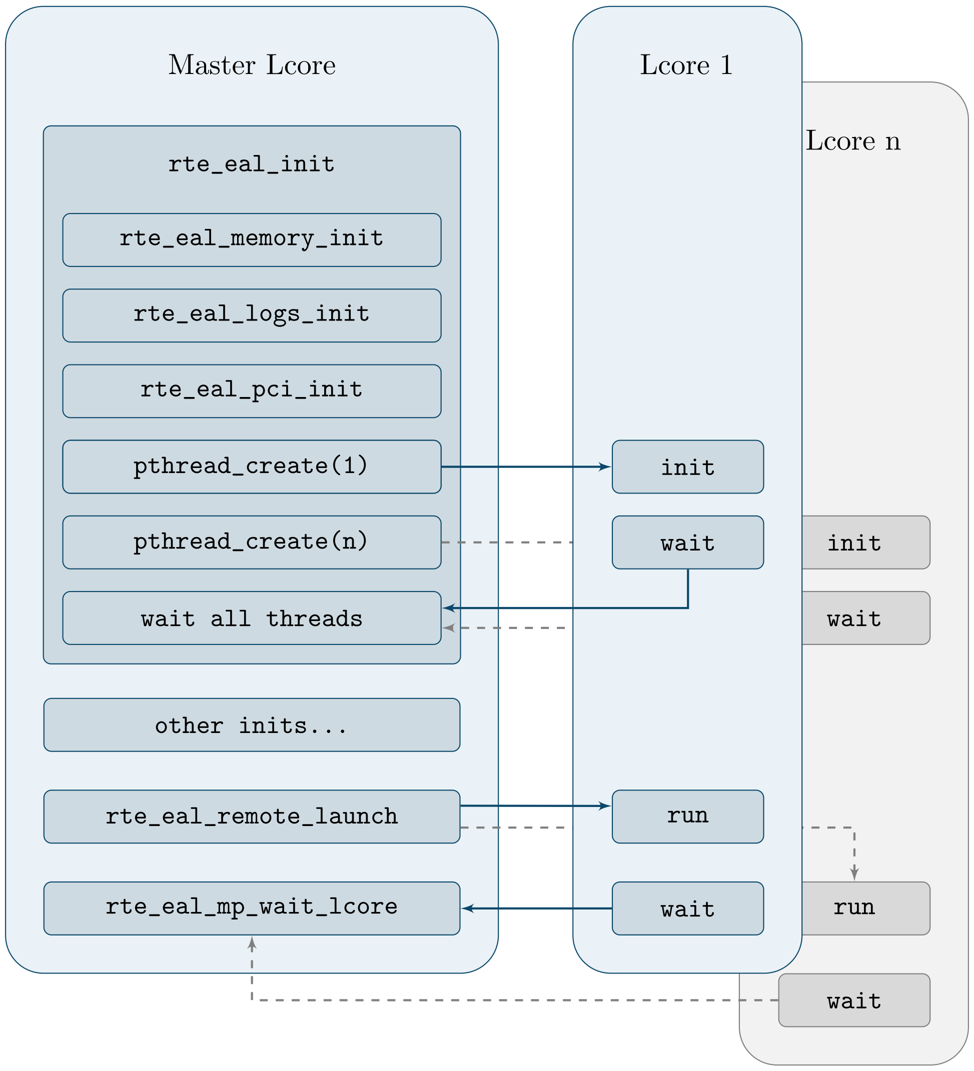

Figure 6.2: `rte_eal_init()` (cf. DPDK programmer's guide [31])

`rte_eal_init(argc, argv):`

> This call prepares the application by initializing different components and creating the threads used by the application. It returns if all threads created are in a waiting state. Figure 6.2 visualizes this call.
>
> Mandatory arguments needed for this init call are `-c` and `-n`. The `c`-parameter must be followed by a bitmask specifying the cores used. For instance 0x5 enables the 0th core and the 2nd core, therefore two threads are created. The indices of the cores are determined by the OS. The `n`-parameter followed by a number specifies the number of memory channels used per processor socket.

After initialization of the EAL, the memory pool is allocated.

`rte_mempool_create(name, n, el_size, cache_size, priv_size, mp_init, mp_arg, obj_init, obj_arg, socket_id, flags):`

> This function creates a memory pool, which, for instance, is used to allocate packet buffers.
>
> The pool has a `name` and contains `n` elements of size `el_size`. If the argument `cache_size` is set to a non-zero number, a per-lcore buffer is allocated. Accesses of a lcore to this respective cache are faster, than accesses to non-cached elements. The `priv_size` is used to store private data after a pool. Argument `mp_init` references a function accepting `mp_arg` as its argument. This function is called at initialization of a pool before the object initialization. It can be used to initialize the private data store. The argument `obj_init` initializes every object at pool initialization. This function takes `obj_arg`, a pointer to the pool, an object pointer, and the object number as arguments. To allocate the pool in a NUMA aware manner, the `socket_id` of the used CPU can be specified. The `flags` can make the pool a single producer or single consumer queue. In addition, usage of memory channels and cache alignment can be configured.

Afterward, the driver is initialized (`rte_pmd_init_all()`) and devices are discovered on the PCI bus and initialized (`rte_eal_pci_probe()`). Subsequently, the ports of these discovered devices can be started with the `prepare_dev()` function. This is called once for the receiving port and once for the transmitting port identified by their respective id.

The `prepare_dev()` call configures the device for the usage with one receive queue and one transmit queue. To reduce the lines of code, the configuration structs `port_conf`, `rx_conf`, and `tx_conf` are omitted.

`rte_eth_dev_configure(port_id, rx_q, tx_q, conf):`

> This function is a mandatory call to configure an Ethernet device, which must be executed before any other function on the device can be called.
>
> The port to configure is specified in `port_id`. For this port, the number of read/write queues to allocate is determined by the values of `rx_q` and `tx_q`. Parameter `conf` is a struct containing various configuration data for features like offloading or options for RSS.

After configuring the device, the queues are configured.

`rte_eth_rx_queue_setup(port_id, q_id, desc, sock_id, rx_conf, mb_pool):`

This function configures the receive queue for a port of an Ethernet device.

The `port_id` identifies the Ethernet port and `q_id` the hardware queue on this port. For this port a number of `desc` receive descriptors is allocated. The parameter `sock_id` specifies the NUMA socket. Different thresholds are given in the struct `rx_conf`. Finally, the memory pool, where the packet buffers are allocated, is determined by `mb_pool`.

`rte_eth_tx_queue_setup(port_id, q_id, desc, sock_id, tx_conf):`

This function configures the transmit queue of a port belonging to an Ethernet device.

The used arguments are similar in functionality to their respective arguments for the receive queue setup with the exception of `tx_conf`, which contains different thresholds.

After initializing the queues, the port is set to promiscuous mode for accepting all incoming packets. On completion of `prepare_dev()` the forwarding thread is started by launching the function `run` on core with id 1.

`rte_eal_remote_launch(func, arg, lcore_id):`

This function can only be called from the master lcore and launches a function `func` with the argument `arg` on a lcore with the ID `lcore_id`.

The launch call returns immediately so the execution of the main thread is blocked by waiting on the completion of lcore 1 with `rte_eal_wait_lcore()`.

The forwarding thread is shown in Listing 6.2. It has an infinite loop starting with a receive call, which receives a batch of up to `MAX_PKT_BATCH` packets.

`rte_eth_rx_burst(rx_port_id, rx_queue_id, buffer, batch):`

This function retrieves a batch of packets on a queue of a port specified by `rx_queue_id` and `rx_port_id`. The received packets are available in the array `buffer`. Per call a number of up to `batch` packets can be received. The actual number of received packets is returned.

On a successful reception of packets, the received packets are put into the array `m_table`. If this table holds enough packets, i.e. the `MAX_PKT_BATCH` is reached, a transfer of this table is filed before a new receive call is executed.

`rte_eth_tx_burst(tx_port_id, tx_queue_id, buffer, batch):`

This function sends packets on queue `tx_queue_id` of port `tx_port_id`. The array `buffer` contains the packets to send and `batch` specifies the maximum number of packets to file for send in this call. A number of the packets actually stored in the descriptors of the transmit ring is returned. This number may be lower than `batch`.

If the transmission of packets failed, the packet buffers remaining in `m_table` are freed manually with `rte_pktmbuf_free()`.

```c
#define MAX_PKT_BATCH 32

void run() {

  struct rte_mbuf *pkts_burst[MAX_PKT_BURST];
  struct rte_mbuf *m;
  struct rte_mbuf *m_table[MAX_PKT_BURST];
  int len = 0;

  while (1) {

    int nb_rx = rte_eth_rx_burst(rx_port_id, 0, pkts_burst, MAX_PKT_BATCH);

    for (int j = 0; j < nb_rx; j++) {

      m = pkts_burst[j];
      m_table[len++] = m;

      /* enough pkts to be sent */
      if (len == MAX_PKT_BATCH) {

        int ret = rte_eth_tx_burst(tx_port_id, 0, m_table, MAX_PKT_BATCH);

        if (ret < MAX_PKT_BATCH) {

          do {
            rte_pktmbuf_free(m_table[ret]);
          } while (++ret < MAX_PKT_BATCH);

        }

        len = 0;

      }

    }

  }

}
```

Listing 6.2: Simplified forwarder implemented in DPDK (Forwarding)

7. Setup

This chapter presents the environment with the software and hardware used for this thesis. Measurements were conducted using a specialized test network. This testbed is called the MEMPHIS testbed depicted in Figure 7.1.

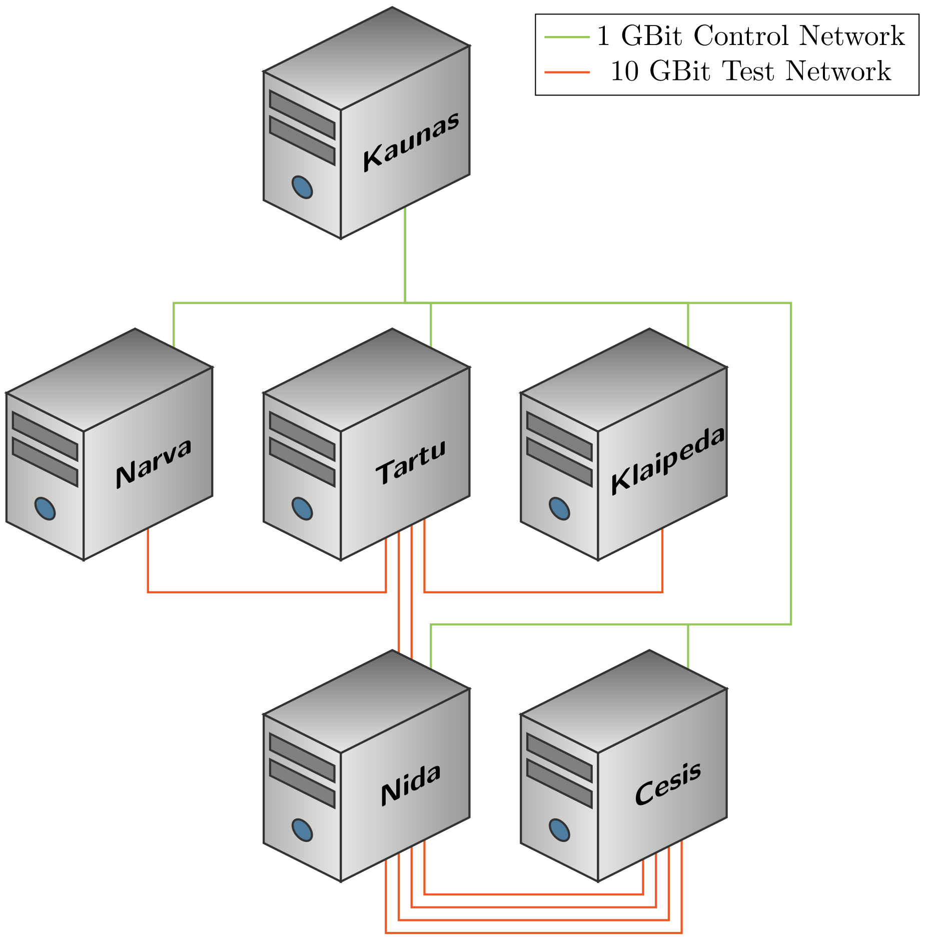

Figure 7.1: Testbed topology

7.1 Memphis Testbed

The Memphis testbed consists of two separate networks, a management network and a test network. Every server is connected to the management network, which itself is controlled by a management server named *Kaunas*. This server loads the software for the experiments on the test servers, which subsequently execute the measurements on the test network. The connections of the test servers shown in Figure 7.1 are direct connections between the servers without the use of hubs or switches to provide measurements unaltered by possible influences of these interconnecting devices.

The test servers mainly used for this thesis were *Narva*, *Tartu*, and *Klaipeda* in a forwarding scenario with *Narva* as a traffic generator, *Klaipeda* acting as traffic sink, and *Tartu* configured for packet forwarding. *Cesis* and *Nida* are the newest servers of the testbed (available since June 2014) using the latest hardware. These two servers were only used for specially selected measurements to show differences between the older NICs used in *Narva*, *Tartu*, and *Klaipeda* and the newer NICs used in *Cesis* or *Nida*.

Hardware

	Narva/Klaipeda	*Tartu*
Mainboard	Supermicro X9SCM-F	Supermicro X9SCM-F
Memory	DDR3, 16 GB	DDR3, 16 GB
CPU	Intel Xeon E3-1230 (3.2 GHz)	Intel Xeon E3-1230 V2 (3.3 GHz)
NIC	Intel X520-SR1	Intel X520-SR2 *(to Narva/Klaipeda)*
		Intel X540-T2 *(to Nida)*

Table 7.1: Hardware configuration *Narva*, *Klaipeda*, and *Tartu*

	Cesis	*Nida*
Mainboard	Supermicro X9SRH-7TF	Supermicro X9DRH-iTF
Memory	DDR3, 16 GB	DDR3, 16 GB
CPU	Intel Xeon E5-2640V2 (2.0 GHz)	2x Intel Xeon E5-2640V2 (2.0 GHz)
NIC	Intel X540 *(dual port, on board)*	Intel X540 *(dual port, on board)*
	Intel X540-T2 *(dedicated)*	2x Intel X540-T2 *(dedicated)*

Table 7.2: Hardware configuration *Cesis* and *Nida*

Certain features influence the processing capabilities of CPUs depending on the current system state. Turbo-Boost and SpeedStep adapt the CPU's clock frequency to the current system load, which subsequently influences measurement results. Therefore, these features were disabled to provide constant processing performance for consistent, repeatable measurements. With Hyper-Threading, each physical core is split into two virtual ones, allowing for the scheduling of two processes to the same physical core. This leads to different results from a scheduling of two processes to different physical cores, counteracting the reproducibility of measurement results. Therefore, Hyper-Threading is also disabled. [26]

Software

Measurements are based on a Linux distribution named Grml Live Linux 2013.02 using a Linux kernel of version 3.7 provided by the MEMPHIS testbed. Being a live system the operating system is freshly loaded on every reboot ensuring a consistent foundation for every experiment.

7.2 High-Performance Frameworks

The versions of the packet processing frameworks are given in Table 7.3. Drivers are included in the frameworks, netmap patches the standard Linux driver ixgbe, PF_RING ZC provides a modified ixgbe driver (v.3.18.7), and DPDK uses an adapted version of igb_uio.

Framework	Version
netmap [1]	*published on 23rd March 2014*
PF_RING ZC [2]	6.0.2
DPDK [3]	1.6.0

Table 7.3: Versions of the investigated packet processing frameworks

[1] https://code.google.com/p/netmap/
[2] https://svn.ntop.org/svn/ntop/trunk/PF_RING/
[3] http://www.dpdk.org/download

8. Modelling Packet Processing Applications

To provide a better understanding how packet processing applications work, a model is conducted to describe such an application. This chapter starts with the investigation of potential bottlenecks to get the main influencing factors of packet processing performance. These factors allow to set up a model for packet processing applications in general. Afterward, this model is adapted to the packet processing frameworks. Describing this kind of application allows to determine the resources needed by the framework itself.

Beside deepening the knowledge about packet processing applications the model enables several possibilities, e.g. to compare different frameworks, to predict the performance of an application using a specific framework, or to provide a tool assessing the applicability of hardware systems for a particular packet processing task.

8.1 Identification of Potential Bottlenecks

As a first step for the design of a model describing the performance of a generic packet processing system, factors influencing performance have to be identified.

The NIC is designed to process 10 GBit/s transfer rates. Therefore, the subsequent interconnects and processing elements need to be investigated for potential bottlenecks.

Modern CPUs have an integrated PCI express (PCIe) controller to connect external devices [26]. PCIe is a point-to-point bus system, i.e. the full bandwidth is available for every device and does not need to be shared with other devices. The NIC uses an 8x PCIe 2.0 interface offering a usable link bandwidth of 32 GBit/s in each direction. The available bandwidth suffices for handling full line rate traffic. There is even enough bandwidth capacity left to handle the overhead of the protocol running on the PCIe bus. [35]

Incoming packets have to be transferred to RAM. Therefore, the memory bandwidth also needs to be considered. A typical configuration for currently available DDR3 memory provides a bandwidth of 21.2 Gigabyte/s (dual channel mode with effective clock speed of 1333 MHz, also used by test system presented in Chapter 7) [24]. The available bandwidth suffices for several 10 GBit NICs and therefore does not need to be considered for the available test setup.

With the interconnects providing sufficient bandwidth, the CPU remains the only factor due to the limited processing capabilities beside the obvious 10 GBit/s limit of the NIC.

8.2 Designing a Generic Performance Prediction Model

To understand the influence of the CPU in a generic packet processing scenario, a model was conducted, which can be described by the following formula:

$$F_{CPU} \geq T \cdot C_{packet}$$

The available resources are represented by the frequency of the CPU (F_{CPU}). The total costs of the packet processing task are given by $T \cdot C_{packet}$, i.e. the number of processed packets represented by throughput T and the costs C_{packet} each packet causes on the CPU. F_{CPU} is fixed depending on the used hardware. Therefore, the available resources are bound by this limit, i.e. they have to be smaller or equal than F_{CPU}. In addition, T is limited by the Ethernet limit of 10 GBit/s.

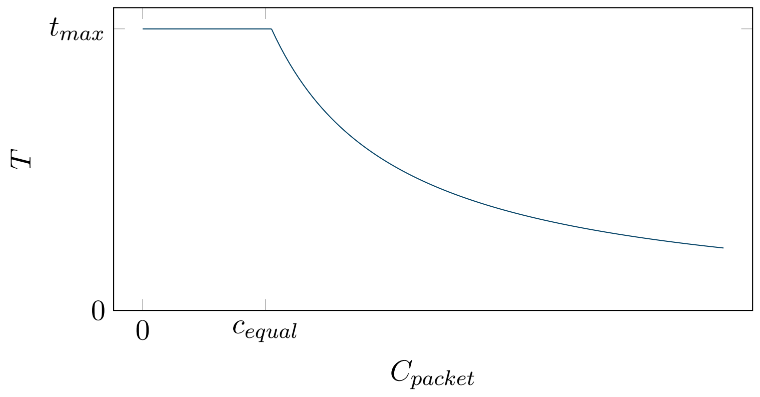

Figure 8.1: Simple model

The Figure 8.1 derived from this model shows the effects of the two limiting factors, i.e. CPU frequency limit and Ethernet limit. Throughput starts at the limit of t_{max} packets per second. The value t_{max} depends on the size of the packets for instance 14.88 million packets per second can be achieved when using 64 Byte sized packets. As long as t_{max} is reached, i.e. the costs C_{packet} are cheap enough to be fully handled by the available CPU, the traffic is bound by the limit of the NIC. At the point c_{equal} the throughput begins to decline. Beyond this point, processing time of the CPU does not suffice the traffic capabilities of the NIC, i.e. the traffic becomes CPU bound and the throughput subsequently sinks.

8.3 Derivation of a High-Performance Prediction Model

Due to the architecture of the frameworks, which all poll the NIC in a busy waiting manner, an application uses all the available CPU cycles all the time. If the limit of the NIC is reached but $T \cdot C_{packet}$ is lower than the available CPU cycles, the cycles are spent waiting for new packets in the busy wait loop. To include these costs, a new value is introduced C^*_{packet} leading to the new formula:

$$F_{CPU} = T \cdot C^*_{packet}$$

The costs per packet C^*_{packet} can originate from different sources:

1. C_{IO}: These costs are used by the framework for sending and receiving a packet. The framework determines the amount of these costs. In addition, these costs are constant per packet due to the design of the frameworks, e.g. avoiding buffer allocation.

2. C_{task}: The application running on top of the framework determines those costs, which depend on the complexity of the processing task.

3. C_{busy}: These costs are introduced by the busy wait on sending or receiving packets. If throughput is lower than t_{max}, i.e. the per-packet costs are higher than c_{equal}, C_{busy} becomes 0. The cycles spent on C_{busy} are effectively wasted as no actual processing is done.

The formula for the refined model is:

$$F_{CPU} = T \cdot (C_{IO} + C_{task} + C_{busy})$$

Figure 8.2 shows the behavior of the throughput while increasing C_{task} for the refined model. Furthermore, the relative part of the three components of C^*_{packet} is shown with highlighted areas. These areas depict the accumulated per-packet costs of their respective component x called $C^{\%}_{x}$.

C_{IO} was kept to a randomly chosen, but fixed value throughout the plot. The relative importance of $C^{\%}_{IO}$ sinks because of two reasons. The first reason is the sinking throughput with fewer packets needing a lower amount of processing power. The second reason is that while C_{task} increases the relative portion of cycles needed for IO gets smaller.

For low values of C_{task} many cycles are not used by C_{IO}, which increases busy waiting that leads to a high value for C_{busy}. $C^{\%}_{busy}$ decreases linearly while $C^{\%}_{task}$ grows accordingly until c_{equal} is reached. This point subsequently marks the cost value, where no cycles are wasted on busy waiting.

C_{task} increases steadily, which leads to a growing relative portion of $C^{\%}_{task}$.

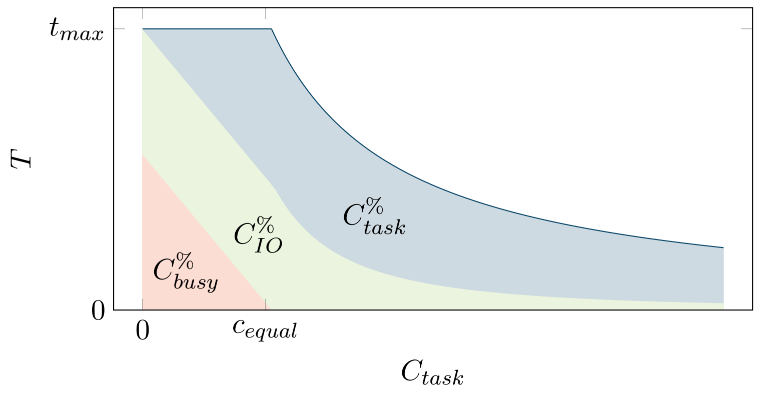

Figure 8.2: Refined model with relative components of C^*_{packet}

In Chapter 9, measurements are used to generate data for this model. The first step is to determine the framework dependent value C_{IO}, which allows comparing the efficiency of packet IO between frameworks.

9. Packet Processing Frameworks - Quantitative Evaluation

Section 2.1 introduced various publications on the three frameworks netmap, PF_RING ZC, and DPDK. Though all these publications cover only a single framework or offer merely incomplete comparisons lacking tests. This chapter tries to provide new information by conducting comparative measurements of all three frameworks under identical circumstances. Therefore, various measurements are presented using artificial but realistic scenarios to investigate the performance of these frameworks. Furthermore, the model presented in Chapter 8 is verified using these measurements.

9.1 Prerequisites for Measurements

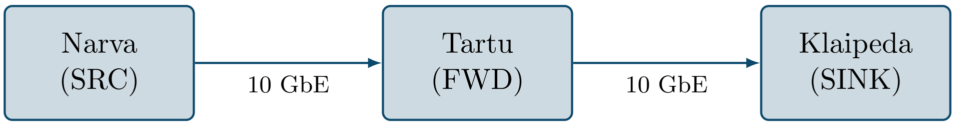

Figure 9.1: Forwarding setup

A basic scenario for packet processing is L2 forwarding, which also was performed for the present survey. Forwarding was chosen as it is integrated as an example project by each framework respectively ensuring optimal performance. Moreover, forwarding is a basic building block for more complex applications such as routing, where the forwarding to different interfaces is determined by a routing table, or packet filtering, where forwarding or dropping of packets depends on the filtering options.

The measurements were performed on the Memphis testbed as shown in Figure 9.1. Ethernet frames were generated on Narva, forwarded by Tartu, and received by Klaipeda. To determine the throughput a packet counter script is running on Klaipeda. For the calculation of C_{IO} the CPU load is measured on Tartu. The packet generator running on Narva is a PF_RING based tool called `pfsend`, which is able to generate traffic at full line rate.

The sampling time for one iteration of a series of measurements was 30 seconds. In fact the RFC 2544 [36] describing methods for measuring network equipment suggests a test duration of 60 seconds. However, tests showed that the frameworks generate a stable throughput rendering longer measurement times unnecessary. In addition, the 30 seconds limit allows to double the possible iterations using the same amount of time. The following experiments present average values taken over the 30 second measurement period. Confidence intervals are unnecessary as results are highly reproducible. An observation also made by Rizzo in the initial presentation of netmap [2].

Adaption of Functionality

Only the forwarder included in DPDK accessed and modified the Ethernet frame header. The alternative solution was implemented by the other two forwarders, i.e. forwarding without any access or modification of the Ethernet header. Aiming for a comparable and realistic scenario the processing of the Ethernet frame was also implemented into the other forwarders.

PF_RING ZC and netmap offer blocking calls for packet reception in contrast to DPDK, which does not offer this feature. To provide a common basis for comparative tests, only non-blocking calls were investigated.

The API of PF_RING ZC offers - beside the batch calls for packet IO - single packet receive and send functions. As the included forwarder example uses these single packet variants, the code was changed to process batches to ensure comparable results.

Packet Size

Due to avoidance of buffer allocation and packet duplication (cf. Section 3.1) CPU load for forwarding mainly depends on the number of packets processed, making the processing costs of a minimal sized packet nearly as high as costs for a maximum sized packet. To aim for the most demanding scenario, the packet size used in the forwarding traffic consisted of 64 B packets, the minimal frame size for Ethernet. For the same reasons netmap [2] was introduced with measurements also done using only the smallest possible packets.

Testing exclusively on the minimal packet size ignores suggestions of RFC 2544 [36]. This request advises to test for different frame sizes on Ethernet (64 B, 128 B, 256 B, 512 B, 768 B, 1024 B, 1280 B, 1518 B). To minimize test duration, only 64 B, 128 B, 512 B, and 1518 B frames were tested, in order to exclude possible anomalies. However, the outcome of all measurements could be predicted from only looking at the results of the 64 B experiment. Therefore, only the results of these measurements are presented.

9.2 Determination of Consumed CPU Resources

Usually standard Linux tools like `top` allow logging the CPU load. However, due to the busy wait loop polling the network interface the forwarders always generate full load on the CPU cores used by the framework. This observation can also be explained using the model. The value measured by `top` is a combination of C_{IO}, C_{busy} and C_{task} without the possibility to determine their relative portion. Therefore, this value is useless for the calculation of C_{IO}.

The load on the CPU can also be measured using a Linux profiling tool like `perf`. However, this application needs processing time for its own execution, which influences the measurement. PF_RING ZC and DPDK also suggest to pin threads to certain cores to minimize system overhead and provide high-performance. Running a profiling tool on the same core would counter these design principles.

Rizzo [2] evaluated the performance of netmap by lowering the CPU frequency while measuring the throughput. This allows finding the minimal clock frequency needed for transmitting at full line rate. This point is highlighted as c_{equal} in Figure 8.2. There C_{busy} is 0. In addition, the forwarders are kept very simple that C_{task} can be approximated with a value of 0. Subsequently, the transmission efficiency of C_{IO} can be calculated from these measured results.

9.3 Hardware Dependent Measurement of Expended Processing Time

To measure forwarding efficiency, Rizzo's method was chosen. Therefore, the frequency of the CPU used by Tartu was adjusted using a tool called `cpufrequtils`. This tool allows running the CPU at fixed, predefined operating frequencies that are provided by the CPU. The frequency range offered by Tartu's CPU starts at 1.6 GHz and ends 3.3 GHz, with intermediate steps of 100 or 200 MHz.

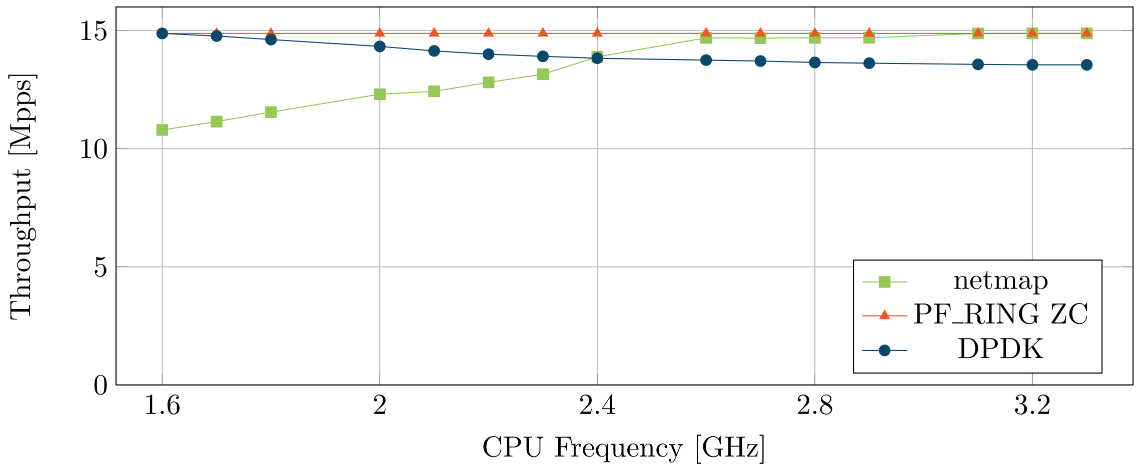

Figure 9.2: Throughput at different CPU frequencies

Figure 9.2 shows the throughput achieved by all three frameworks for forwarding 64 B packets. The expected behavior for all three frameworks was to forward fewer frames for slower clock frequencies. However, three different results were measured:

- **netmap:** behaves as anticipated. It starts at roughly 11 Mpps at 1.6 GHz and increases throughput linearly until line speed is reached for clock speeds higher than 3 GHz.
- **PF_RING ZC:** transmits at full line rate for every clock speed offered by the CPU.
- **DPDK:** reaches line speed at 1.6 GHz but decreases its throughput performance for faster clock speeds.

The explanation for the behavior of PF_RING ZC is probably the higher efficiency compared to netmap. Therefore, the minimal clock speed could still be too high to lower the throughput.

The behavior of DPDK was further investigated. It was speculated that the decline in throughput was caused by a high number of polls. Subsequently, a new counter was introduced to the forwarder counting the receive API calls per second, which is shown in Figure 9.3.

The forwarder included in DPDK works with a batch size of 32, i.e. the NIC is polled in a busy wait manner until at least 32 packets are available (cf. Listing 6.2). From that information, the minimal number of polls per second to reach line rate can be calculated:

$$\frac{\text{Maximum Packet Rate}}{\text{Batch Size}} = \frac{14.88\text{ Mpps}}{32} \approx 0.47\text{ MPolls/s}$$

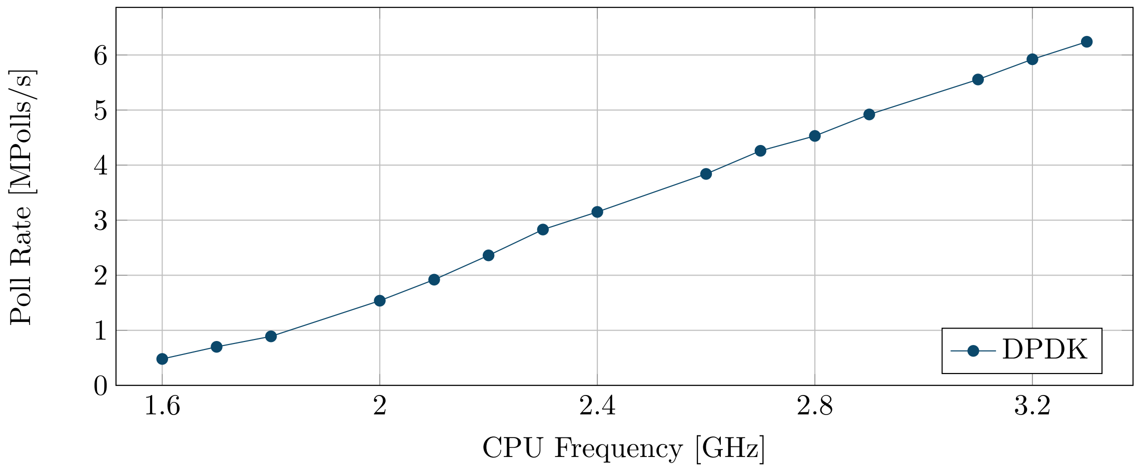

Figure 9.3: Pollrate of DPDK forwarder at different CPU frequencies

If the forwarder meets this number of receive calls (evenly distributed over one second), every call returns with 32 newly arrived packets. For less calls the line rate is no longer reached but for a higher number of calls the busy loop immediately polls the NIC again. This high number of polls during a short period of time leads to the decreasing throughput, which is shown in Figure 9.3. At a clock speed of 1.6 GHz the calculated minimal poll rate is roughly met and DPDK reaches line rate as shown in Figure 9.2. For higher clock speeds the number of polls increase, which subsequently lowers the throughput.

To avoid this poll rate problem, the forwarder was modified limiting its poll calls to the required minimum. Therefore, the minimal wait time between two consecutive receive calls was calculated:

$$\frac{\text{Time Period}}{\text{Calls}} = \frac{1\text{ s}}{0.47\text{ MCalls}} \approx 2.128\frac{\mu\text{s}}{\text{Call}}$$

As `sleep()` and `usleep()` proved to be too inaccurate for this minimal wait time a more precise timer was needed. Therefore, a precise timestamping method based on the RDTSC [37] cycle counter of the CPU was developed. Now the minimal wait time needs to be converted to CPU cycles:

$$\frac{\text{Time Period}}{\text{Calls}} = \frac{3300\text{ MCycles}}{0.47\text{ MCalls}} \approx 7000\frac{\text{Cycles}}{\text{Call}}$$

To adhere to this minimal wait time, every receive call is timestamped with the CPU's cycle counter. By adding 7000 cycles to this value the earliest point for the next receive call is calculated. This minimum delay is enforced by a busy wait loop before every receive call that loops until the RDTSC reaches the previously calculated value. With that improvement DPDK was - like PF_RING ZC - able to reach full line rate for every given clock speed.

Further investigations showed that the poll rate problem could not be reproduced for the PF_RING ZC framework. Nevertheless, it was observed for netmap in a different scenario. The packet generator included in netmap tries to send packets in a busy loop. In this loop the send call is executed even if no space in the send buffer is available also leading to a decreased throughput for higher CPU frequencies. There the problem could also be fixed with an enforced minimum delay between polls.

The poll rate problem was further investigated as Cesis and Nida became available to the testbed. The experiments were repeated on these new servers running the same software with the exact same settings, but NICs were changed to Intel X540 cards. On those cards, unnecessary polls did not lower the throughput. As the only difference between these two test setups was the changed hardware, the poll rate problem seems to be a hardware problem as it only occurred on Tartu using Intel X520 cards.

9.4 Measuring Expended Processing Time under High Load

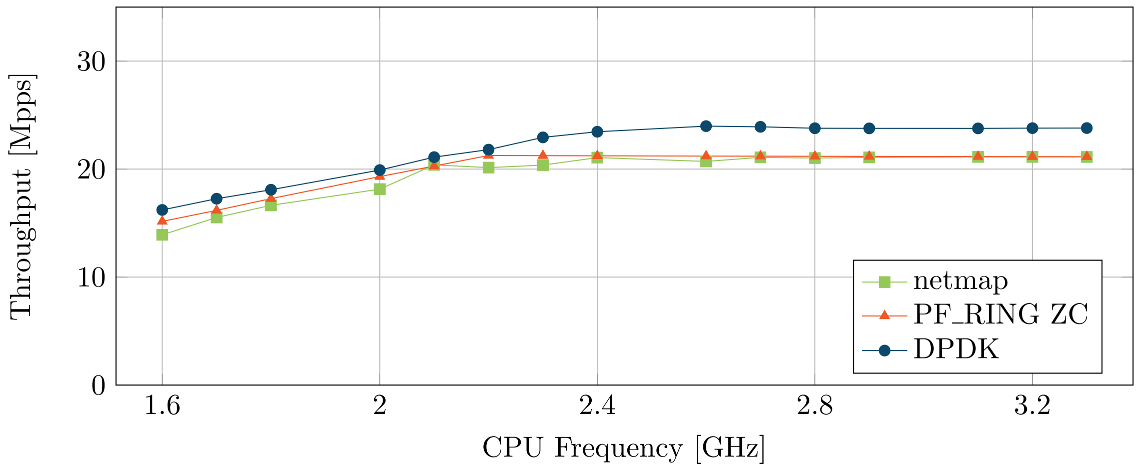

Figure 9.4: Bidirectional forwarding

Measurements in Section 9.3 showed that there was no influence of clock frequency on the throughput for PF_RING ZC and DPDK. It was suspected that the CPU load was too low for the given clock frequencies. As these frequency operation points are fixed and cannot be changed without changing the hardware, the CPU load was changed. This was achieved by switching from unidirectional forwarding to bidirectional forwarding. For this experiment, Klaipeda and Narva both run a packet generator and the measurement script. The forwarder on Tartu forwards in both directions and must process twice as many packets roughly doubling the load on the CPU.

To increase the CPU load, the forwarding applications were limited to the use of one core. DPDK and netmap already support bidirectional forwarding on one core, the forwarder of PF_RING ZC had to be adapted to meet this requirement. The generated packets had a length of 64 B. The results of this measurement are shown in Figure 9.4. It depicts the accumulated throughput of both directions for all supported CPU operation frequencies.

One observation is the fact that netmap has a higher throughput in Figure 9.4 than in Figure 9.2 for the same clock frequencies. For instance at 1.6 GHz the unidirectional forwarder has a throughput of 10.8 Mpps but for bidirectional forwarding a throughput of 13.9 Mpps was measured. The costs for a netmap API call are high as system calls are used for packet reception and transmission. Bidirectional and unidirectional forwarder have the same number of API calls but the received/transmitted packets per call are higher in the bidirectional case. Therefore, the API overhead is distributed to a larger number of packets reducing the average per-packet costs resulting in a higher throughput. For the other two frameworks this observation could not be made as unidirectional forwarding was limited by the NIC and not the CPU.

However, the major observation to be made is the fact that the theoretical limit of bidirectional forwarding is never reached. This limit would be 29.8 Mpps for 64 B packets but it is never met by the forwarders. Despite the availability of CPU resources all frameworks peak long before the clock frequency reaches its maximum. PF_RING ZC and netmap both peak at a throughput of 21.1 Mpps and DPDK at 23.8 Mpps. Therefore, the CPU cannot be the limiting factor in this measurement. Even the activation of a poll rate limiter did not change the outcome of this experiment.

A possible explanation for the limitation of bidirectional forwarding could be an overloaded PCIe bus. However, this assumption could not be proven on Tartu, as the CPU did not support the measurement of the PCIe throughput. The more recent CPUs used by Cesis and Nida allowed the measurement of PCIe utilization. The link between NIC and CPU has the same properties using a PCIe version 2 with eight lanes. Therefore, the bidirectional forwarding experiment was repeated with DPDK on these servers. There the theoretical limit of the NIC was reached. Although two cores had to be used due to the lower CPU frequency of the CPU used on Cesis and Nida. Furthermore, a tool called Performance Counter Monitor[1] provided by Intel was used to measure PCIe utilization during the experiment. The PCIe link was only utilized to 63% / 50% from NIC to CPU or vice versa. Therefore, the PCIe bus utilization is no explanation for the low throughput performance of the NICs. It seems to be another hardware limitation of the X520 cards used by the older servers, which no longer holds for X540 cards used by the newer servers.

9.5 Removing Hardware Dependency from CPU Load Measurements

Bidirectional forwarding was able to lower the throughput for all three frameworks, but showed hardware limitations of the NIC, which make it difficult to clearly decide if a decline happens because of the NIC limit or insufficient CPU resources. Therefore, this method cannot be used to measure the precise CPU load during forwarding. However, the previously presented measurements showed also that unidirectional forwarding was not sufficient to show a decline in throughput for PF_RING ZC and DPDK due to the frequency restrictions of the CPU.

An optimal solution combines a unidirectional forwarding scenario with a higher CPU load. To surpass the hardware limits opposed by the CPU, a new method was developed to lower the processing capabilities without hardware dependency. Therefore, a piece of software was developed, which takes a predefined number of cycles to execute.

Generating and Measuring CPU Load

```
inline void wait_cycles(uint32_t wait_value) {
   asm volatile("mov %0, %%ecx"
                "label: dec %%ecx"
                "cmp $0, %%ecx"
                "jnz labelb":: "r" (wait_value));
}
```

Listing 9.1: Busy wait loop in assembler

The code for this CPU load generator is shown in Listing 9.1 and was put into a header file to easily integrate the software into the forwarders. This function was declared as inline to avoid overhead due to the function call. Volatile assembler instructions were used to prevent the compiler from optimizing this code that could influence the number of cycles needed for execution. The `asm()` function of `gcc` allows to embed assembler instructions into regular C code.

[1]http://www.intel.com/software/pcm

Listing 9.1 implements a wait loop with the number of loop iterations as input argument (`wait_value`). Line 2 loads the first argument to register `ecx`. Subsequently, the content of `ecx` gets decremented, and compared to 0. As a last step, a jump to the label called `label` is performed if the result of the compare operation was false. If `cmp` evaluates to the value true the loop is finished. The compiler takes care of restoring the registers to the original values before the execution of this call as the registers are probably used by other methods.

To measure the execution of a piece of code, a method for precise benchmarking on CPUs using x86 or x64 is described by Paoloni [37]. For this measurement a hardware counter counting every clock on a CPU, called `RDTSC` and `RDTSCP` are used.

```
int start = asm("RDTSC");
/* code to benchmark */
int end = asm("RDTSC");
```

Listing 9.2: Naive benchmarking on x64

A naive implementation of benchmarking is shown in pseudo code in Listing 9.2. There the values of `RDTSC` are read before and after `code to benchmark` to calculate the time difference by subtracting `start` from `end`.

However, modern CPUs with out of order execution can reorder the instructions for optimization purposes. If the code to benchmark contains a memory access, the second counter evaluation could be executed while waiting for memory rendering this method useless for benchmarking. For precise benchmarking, the sequence of execution is critical and must be enforced. This can be done by calling a serialization instruction before reading the counter, forcing the CPU to finish all instructions before the serialization instruction.

```
int start = asm("CPUID
                 RDTSC");
/* code to benchmark */
int end = asm("RDTSCP
               CPUID");
```

Listing 9.3: Precise benchmarking on x64

The benchmarking solution presented in pseudo code in Listing 9.3 performs the serialization instruction `CPUID` before the first counter evaluation. The second time the `RDTSCP` variant of the counter evaluation is used, which ensures that the code to benchmark is completed before reading the value of the counter. The second `CPUID` call ensures completion of counter reading before executing the following instructions.

Another factor influencing the measured time difference is the cache. The first execution of a piece of code contains the time needed for copying the code into the cache. Therefore, the first measurement is omitted in the following measurements.

Figure 9.5 shows the `wait_cycles()` function for `wait_values` between 1 and 1000. The experiment was performed on Tartu with a fixed CPU frequency of 3.3 GHz. Every single point in this graph represents the average of 100000 benchmark calls. Using the data a linear function was derived to calculate real cycles from `wait_value`:

$$\text{Cycles} = \texttt{wait\_value} + 77$$

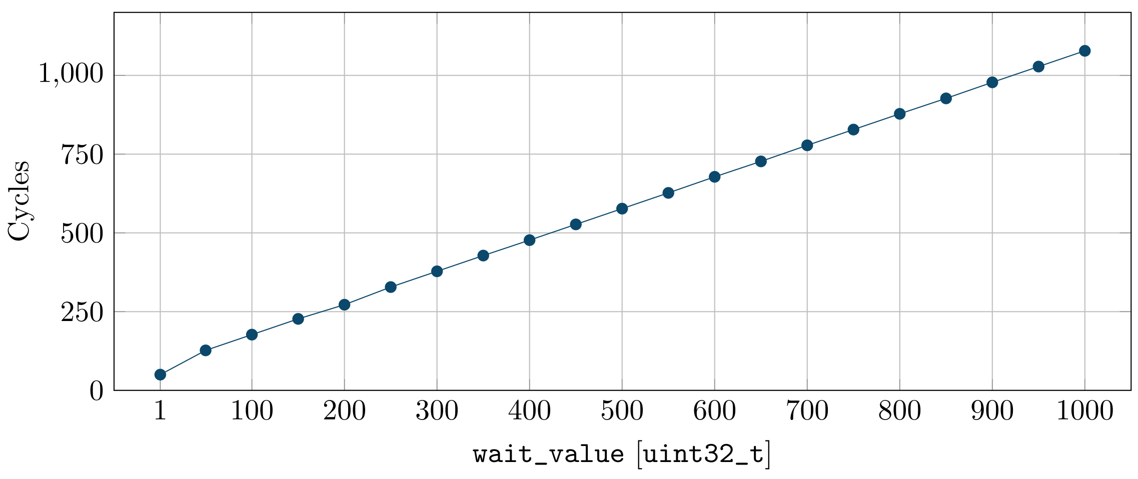

Figure 9.5: Measurement of `wait_cycles`

Enhancing Forwarder to a Generic Packet Processing Application

The `wait_cycles()` function can be used to emulate a CPU load with a predefined number of cycles. Therefore, this function was implemented into the three forwarders. A new command line argument in the forwarders specifies a fixed number of cycles, which is spent for each packet using `wait_cycles()`. These cycles spent on waiting can simulate an arbitrary complex processing task with a fixed runtime. Therefore, this forwarder can now be considered as a generic emulator for a packet processing application using a fixed number of cycles per packet.

Moreover, this packet processing emulator can be described by the model in Figure 8.2. The `wait_cycles()` function simulates the runtime of C_{task}. C_{task} is increased until all cycles are spent either on C_{task} or on C_{IO} and no more cycles are spent on waiting for packets (C_{busy}). Using such a point in the measurement allows to calculate the only remaining unknown variable C_{IO} as C_{busy} is 0 for such a throughput, F_{CPU} is given by the CPU with 3.3 GHz, C_{task} is a predefined value, and T is measured.

Figure 9.6 shows the experiment with the generic packet processing emulators of all three frameworks. The DPDK forwarder used a limited poll rate to reach line rate on low CPU load.

The upper graph of Figure 9.6 appears to behave as it was predicted by the model (cf. Figure 8.2). PF_RING ZC and DPDK offer roughly the same and netmap offers lower performance. This difference in performance can be explained by the architecture of netmap. In netmap system calls cause additional overhead effectively reducing CPU cycles available for packet processing. DPDK has slightly lower performance for C_{task} values from 50 to 100 than PF_RING ZC. That difference is caused by the poll rate limiter, which was tuned very carefully to meet the optimal poll rate for C_{task} being 0. Only at this point line rate is achieved. To meet the optimal poll rate that close for higher values of C_{task} would have needed additional manual tuning. For values of 150 or higher for C_{task} DPDK has slightly higher performance than PF_RING ZC.

The lower graph of Figure 9.6 shows the per-packet costs $C_{IO} + C_{busy}$. By calculating C_{IO} both components can be displayed separately. The curves can also be divided in two groups the very smooth curves of DPDK and PF_RING ZC and the rough curve of netmap. This also can be explained with the system call architecture of these frameworks. System calls and user space functions work on the same cache. Therefore, the user space function of netmap might evict data from cache needed by the system call and vice versa.

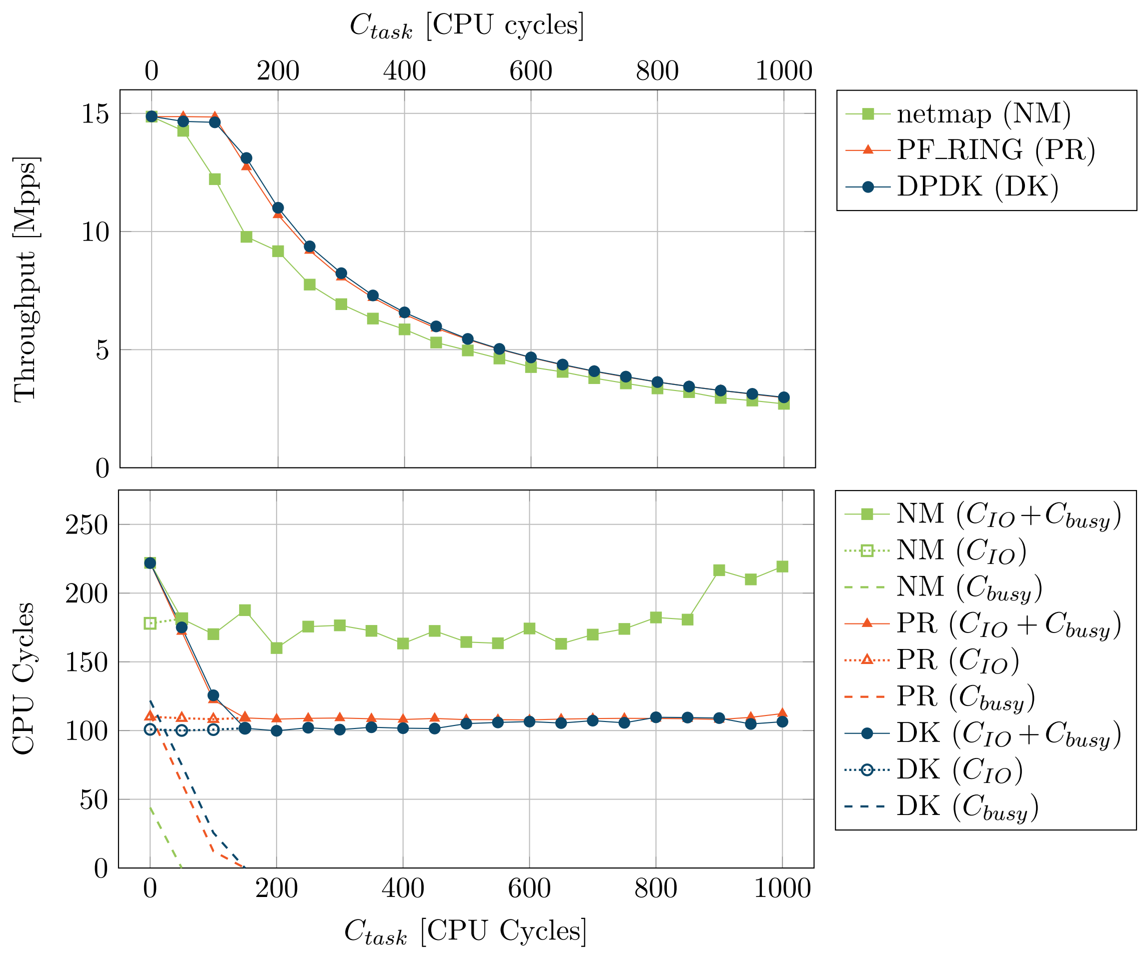

Figure 9.6: Generic packet processing application with fixed per-packet costs

This leads to a higher variance in data access times depending on the current location of the data to be accessed. DPDK and PF_RING ZC generate no system calls for packet processing. Therefore, no competition happens for the space in the caches leading to more homogenous access times resulting in a smoother graph.

The previously presented measurement was done using 64 B packets. It was repeated with larger packet sizes of 128 B, 512 B, and 1518 B. Nevertheless, the curves for these measurements are omitted, as they do not contain any unexpected results. Using larger packets the throughput was able to reach line rate for even higher values of C_{task}. However, the value of C_{IO} remained constant for these measurements. This shows that CPU load generated by these frameworks depends on the number of packets to be processed rather than the size of these packets.

Framework	Average C_{IO} [CPU Cycles]
netmap	178
PF_RING ZC	110
DPDK	105

Table 9.1: Efficiency of packet reception and transmission

The efficiency of every framework is given (C_{IO}) in Table 9.1. Using these values of C_{IO} in the model allows a prediction of packet transfer rate (T) on a CPU with known frequency F_{CPU} for a packet processing application with the known complexity C_{task} per packet.

The value of C_{busy} represents additional cycles, which should be avoided as these cycles are wasted.

9.6 Influence of Burst Size on Throughput

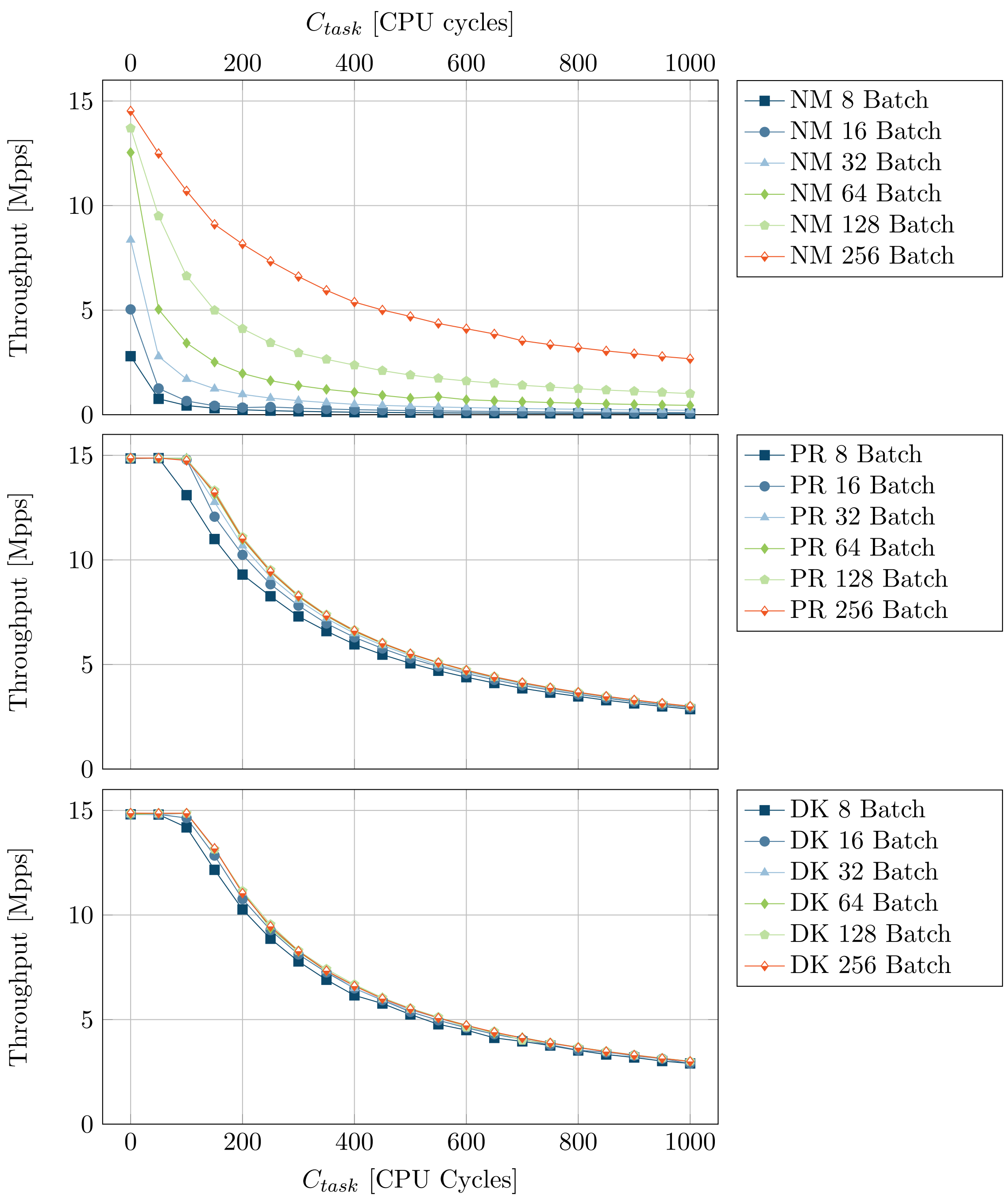

Figure 9.7: Throughput using different batch sizes

For the previous measurements, the batch size of the forwarders was kept to the default settings included in the example code of each framework. PF_RING ZC and netmap only have a maximum batch size. Therefore, per send/receive call a number of packets between one and the maximum batch size is processed. DPDK forwarding has a fixed batch size and a timeout. Received packets are collected and kept in a buffer until the batch size is reached before the packets residing in the buffer are sent. A timeout ensures the sending of the packets on low traffic rates, i.e. if the buffer is only filled very slowly. This is done to avoid unnecessary high latencies for packets.

To compare the influence of batch sizes, the behavior of PF_RING and netmap was changed to the algorithm used by the DPDK forwarder. A function to accept the batch size as command line argument was integrated into all the forwarders. This argument takes the fixed batch size as an input value.

The results of this measurement are shown in Figure 9.7 presenting results in a separate graph for every framework. Once again, PF_RING ZC and DPDK behave rather similarly. Starting with a batch size of 8 throughput increases for larger batch sizes. PF_RING ZC profits up to a batch size of 64. Larger batch sizes have no influence on the throughput of this framework. DPDK is more efficient. At a batch size of 32 throughput no longer profits from increasing the batch size.

Throughput of netmap is very dependent on the used batch size. Increasing the batch size improved the throughput for every measured value until the throughput came close to the values measured in Section 9.5. This behavior is a result of netmap using more expensive system calls for sending and receiving compared to the simple function calls in user space used by the other frameworks. If the high API call costs of netmap are distributed to a smaller batch size, the average per-packet costs are increased lowering the throughput.

These results show that netmap should only be used with batch sizes of at least 256 to offer performance close to what the other frameworks can achieve.

9.7 Influence of Memory Latency

Typical packet processing applications often depend on a data structure as a rule set describing the operations to be executed for a certain packet. In a routing scenario, this rule set is a lookup table, which is applied to generate a forwarding decision. The time needed for an access to this lookup table is critical for the final speed of the packet processing task. Therefore, the forwarding emulator was adapted to perform a data dependent operation for every packet processed. For this purpose a special data structure was developed. This data structure is similar to the data structure used for cache measurements described by Drepper [38].

Design of the Test Data Structure

The main goal of this data structure is to generate a predictable amount of memory accesses to introduce latency while waiting for data. Processing overhead of this data structure should kept low as only the latency introduced by the memory access should be measured. Moreover, the size of this data structure should be variable to test for the effects of different cache levels.

As a basic data structure, a singly linked list was chosen. The size of the list can be influenced by two parameters the size of a single list element and the number of elements in this list.

```c
#define PADDING 7

struct element {
    struct element* next;
    uint64_t pad[PADDING];
}
```

Listing 9.4: List element

Listing 9.4 shows an element of this list. The element contains a pointer to the next list element called `next` and an array `pad`, which can be adapted to vary the size of each element. For this measurement the element size was set to 64 B (8 B for `next` and 56 B in total for `pad`) to match the cache line size of 64 B [26]. This size guarantees one access per element as only a single element is kept in one cache line. In addition, the elements are aligned so no element is distributed to several cache lines what would result in several cache accesses per element.

To generate an infinite number of memory accesses with a limited amount of elements in the list, head and tail of this list were connected to build a circular queue.

A normal queue can be connected linearly, i.e. the element referenced by `next` has its start address directly after the currently selected element. Prefetching mechanisms (cf. Section 2.3) would cache the next elements. Therefore, this access pattern would produce a high number of cache hits. As this access pattern is not realistic for the data structures, which are used in network filters or routers, the links of the queue were randomized to counteract the prefetching mechanisms. The algorithm chosen to generate these connections produces no subcycles, i.e. every element of the list is seen exactly once on a traversal of the whole queue. In addition, the seed, which determines the list connections, was kept the same over all measurements to ensure comparability of results between different test runs.

A technique supported by Linux is the usage of larger memory pages. Regular sized pages have a size of 4 KB and the larger ones called huge pages have a size of 2 MB. The larger pages lead to a more efficient use of the TLB (cf. Section 2.3) as less memory address calculations have to be performed. To test the influence of page size, the queue can either be allocated using normal sized pages or the larger ones.

Results of a test run generated from 1 million iterations in the queue are shown in Figure 9.8 with separate curves for 4 KB and 2 MB pages. The larger pages seem to have no influence as long as the data structure fits completely into cache. However, if the data structure has to be accessed in RAM the larger pages have a positive influence on access times.

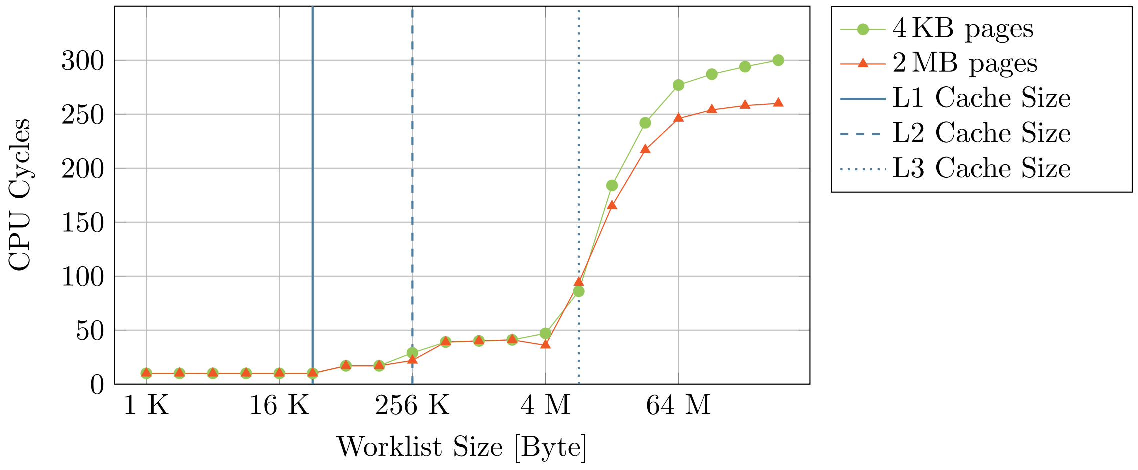

Figure 9.8: Measurement of cache test queue with 4 KB and 2 MB pages

Measurement of Memory Access Latency on Packet Processing Performance

The previously described data structure was implemented into the forwarder, which now can generate a memory access delay beside the fixed CPU load per packet by performing one iteration in the previously presented queue. An additional command line argument specifies the use of regular sized or huge pages. The size of the memory access list can also be chosen via a command line argument. For this experiment, the forwarders use their native algorithm for choosing the batch size.

Figure 9.9 shows the throughput of the adopted forwarders using regular sized memory pages. The size of the queue was doubled after each iteration starting with a size of 1 KB up to a size of 512 MB. Additionally, a fixed CPU load of 100 cycles per packet had to be introduced to lower the throughput. Without this offset the influence of the cache delay would be invisible due to C_{busy}. The throughput of the netmap curve is lower as the 100-cycle offset has a stronger influence on this framework (cf. Figure 9.6).

All three frameworks show a drop in performance before the queue reaches the size of a cache level. This can be explained by different data structures competing for the space in the cache, i.e. the data structures needed by the application without the additional queue.

The throughput of PF_RING ZC for queue sizes fitting into the third level cache is higher than the throughput of DPDK, which offers better performance in every other tested configuration. This shows that DPDK depends to a greater extent on the L3 cache for its operation as PF_RING ZC.

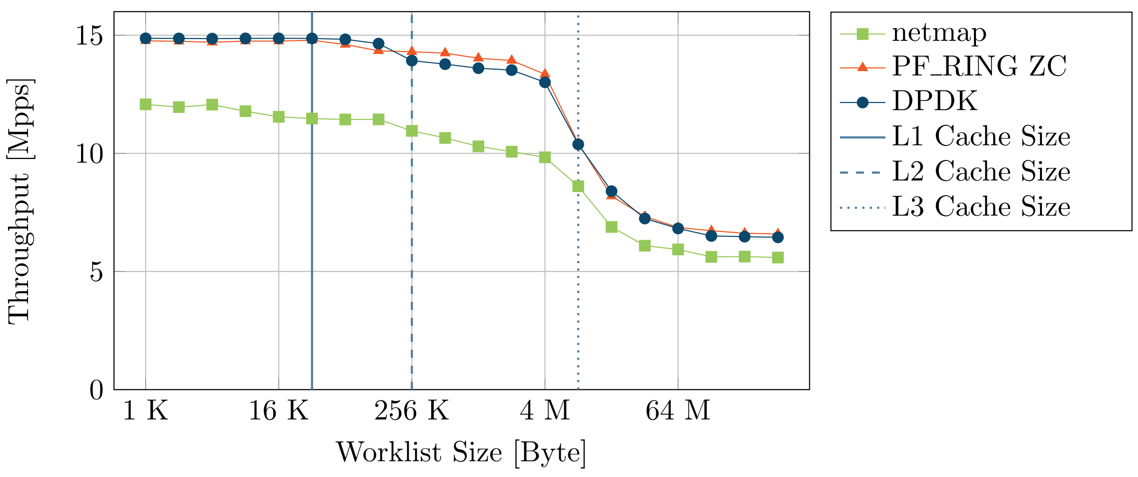

Figure 9.9: Forwarding with one cache access per packet (4 KB pages)

The same experiment was repeated with the queue allocated in huge pages. Results are displayed in Figure 9.10. As long as the test data structure fits into cache, the throughput is nearly the same as for the previous experiment, but for non-cached data accesses, the throughput is roughly 0.5 Mpps higher.

This experiment shows that performance heavily relies on data access times. In this test scenario with an offset of 100 CPU cycles, the throughput drops by almost 6 Mpps when RAM accesses cannot be cached any longer. This drop can be reduced to only 5.5 Mpps by using huge pages. However, even in this case a packet processing application can profit from cached data structures. Subsequently, the caching behavior should be considered when designing a packet processing application. This starts by choosing a CPU with a cache size meeting the application's requirement. In addition, reducing the size of data structures to fit into cache or reordering data to use prefetching should be considered.

Measurement of Cache Hits/Misses

The Intel Ivy Bridge CPUs provide counters for the measurement of specific hardware events [26]. For the previous measurements, counters for the cache hits and the cache misses on every cache level were monitored using these hardware registers. To read them

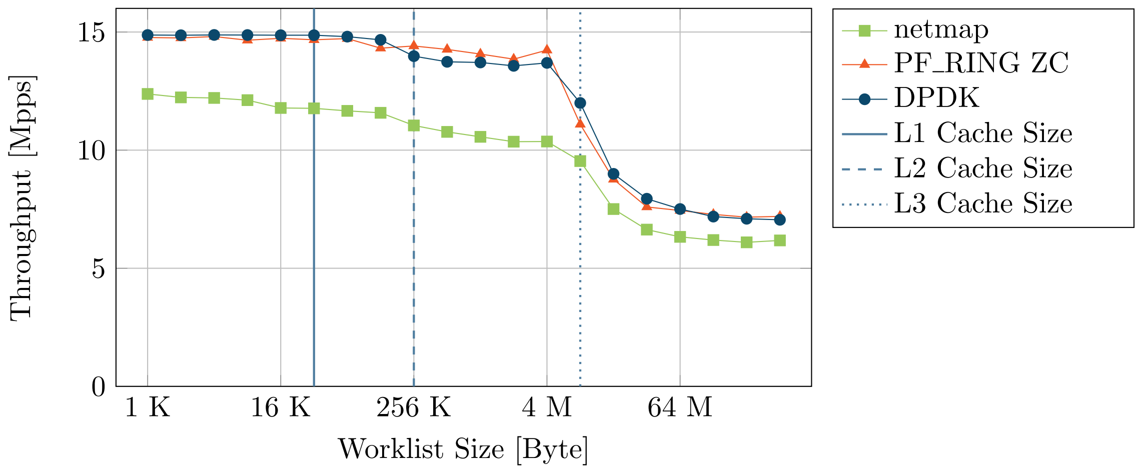

Figure 9.10: Forwarding with one cache access per packet (2 MB pages)

the profiling tool `perf` was applied. Each iteration of the test lasted 2 minutes with the hardware counters read every 10 seconds. As only 4 hardware counters are available, only 4 events can be monitored simultaneously. Therefore, the L1 and L2 events were monitored 6 times. Afterward, the L3 events were measured 6 times. The profiling tool has no measurable influence on forwarding performance it only reads hardware registers and sleeps otherwise so the overhead and possibly introduced cache accesses are kept to a minimum.

The results of this measurement are presented in Figure 9.11 for each framework separately. Only the results for 2 MB pages are displayed. The outcome for 4 KB pages is nearly identical. The total number of hits (and misses) sinks during the course of this experiment as the throughput declines, i.e. the CPU idles and generates less data accesses.

One observation is the high number of L1 Hits in all three cases. This is the case because every variable used regularly effectively never leaves this cache level, i.e. the cache is hit every time on access. The decline of the L1 hits seems to be very little but due to the use of a log scale the number of hits roughly halves during the experiment for all three frameworks.

Another observation is the usage of the cache by the test data structure. The test data structure initially fits into the L1 cache, i.e. an access to the data structure generates a L1 hit with a high probability. Even if a miss is generated, the date is found in the L2 cache with a high probability. By increasing the size of the test data structure, the number of hits in the L1 cache decreases but the number of hits in L2 hits increases. The maximum of L2 hits is reached even before the L1 size limit of 32 KB as the cache is not exclusively used by the test data structure. This observation holds for the iteration from L2 to L3 but also from L3 to memory.

The numbers in Figure 9.11 show differences in the absolute numbers for the frameworks. Nevertheless, explained behavior is obvious for the curves of DPDK and netmap. PF_RING ZC has a similar behavior with the exception of the curve for the L3 cache. This cache is already heavily used where the other two frameworks are still generating L2 cache hits. A possible explanation is that the L2 cache is already used by a large data structure of the frameworks itself, which competes for the cache. This is indicated by the earlier decline of the L2 hit curve at only 128 KB. However, as the source code is closed for this framework this assumption was not verified.

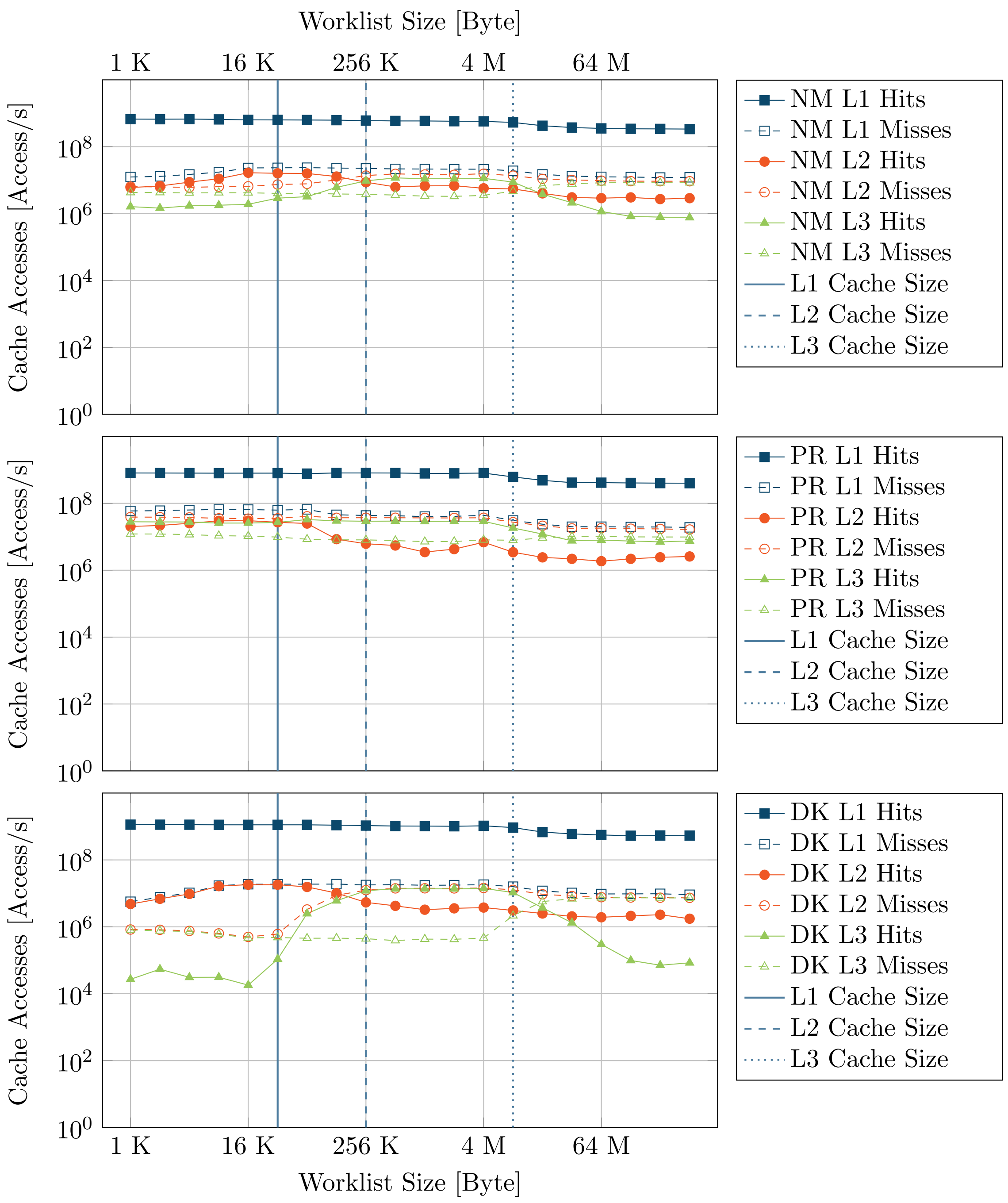

Figure 9.11: Cache hit/miss measurements during forwarding (L1-L3)

10. Conclusion and Outlook

The previous chapters presented a new solution for packet processing breaking with the established way of packet processing by radical reduction of functionality and putting the focus on performance. Three different approaches were introduced: netmap, PF_RING ZC and DPDK.

10.1 Comparison of High-Speed Packet Processing Frameworks

	netmap	PF_RING ZC	DPDK
Throughput performance	+	++	++
System calls for packet send/receive	+	-	-
Support for blocking calls	+	+	-
API ease of learning	+	+	-
Framework function scope	-	+	++
Robustness	++	-	-
Support for huge pages	-	+	+
NUMA awareness	-	+	+

Table 10.1: Summary of features supported by different packet processing frameworks

Table 10.1 provides a compact repetition of the findings for the different frameworks presenting their individual advantages and disadvantages.

The netmap framework has the most conservative design of all three contestants using well-known software interfaces of Linux and BSD. This familiarity of the API reduces the barriers for programmers and simplifies adopting programs to netmap. Using system calls for checkups also increases the robustness of the framework. A price that is paid with a lower performance compared to its rivals.

To aim for simplicity is the goal of PF_RING ZC, which, for instance, is shown by the custom, simple API. Despite its convenience, the software interface is still powerful enough to write multithreaded, high-speed packet processing applications with only a few lines of code. Nevertheless, the API leads to greater effort for the adaption of applications to PF_RING ZC. To use this framework, a commercial license is needed and the source code of the ZC library is not publicly available in contrast to its competitors.

DPDK being a collection of libraries goes beyond pure packet IO. The variety of functionality offers a high degree of freedom for the design of packet processing applications with previously unmet transfer rates. Flexibility and performance are at the expense of simplicity rendering the API the most powerful but also the most complex API of its competitors. Despite the complexity the whole framework is well documented and performance is even higher compared to the already very fast PF_RING ZC.

Comparing these frameworks shows that redesigning packet IO can result in great performance benefits. Moreover, it is demonstrated that the more a frameworks breaks with traditional designs the greater are the performance gains. Even though DPDK offers the best performance of all competitors, considering other factors like robustness, ease of use, and barriers to entry make its contestants viable alternatives for the realization of packet processing applications. For instance, adopting legacy applications to a packet processing framework is more easily realized in netmap because of the similar API. Programmers new to packet processing can realize applications faster in PF_RING ZC than in any other framework due to the low entry barrier of the API. If the highest performance or a additional libraries offered by DPDK are requirements of the packet processing application, DPDK should be the framework of choice.

10.2 Evaluation of Model and Measurements

Another outcome of this thesis is a model description of a generic packet processing application. To use this model the application specific per-packet costs must be available. In combination with a frame specific constant provided in this thesis and the available processing power this model allows to predict the expectable throughput of a packet processing application. Knowing the throughput requirement of an application scenario the model can also be used to assemble the hardware systems needed to meet those performance requirements.

As various measurement methods proved to be incapable of measuring the CPU resources needed by the frameworks a new indirect method is presented. To measure the CPU performance needed by an application, a function is introduced using a defined number of CPU cycles. Knowing this newly introduced additional load allows calculating the CPU cycles needed by the application for a measurable outcome.

To realize this kind of measurement, three simple tools are presented. One of those tools allows measuring the number of CPU cycles needed for a specific function. In addition, the generation of a specified CPU load is demonstrated along with a method to generate reproducible patterns of memory delay into an application.

These three tools were used to measure the outcome of packet forwarders for each framework in different load scenarios. Subsequently, the CPU cycles needed to realize a measured throughput could be calculated.

However, these tools also created additional possibilities. Many applications like routing, switching, and filtering use the forwarding of packets as a basic building block but differ in their respective packet processing task. These tasks can be simulated and measured using the previously developed tools. Therefore, the forwarders of the frameworks were enhanced to simulate arbitrary complex processing tasks on a per-packet basis. This allows realizing an emulator for generic packet processing applications on these once simple forwarders by mimicking a processing task of the designated application.

10.3 Additional Findings

During the measurements, different, unexpected limits of the NICs used were observed. Polling a NIC too often caused a decline in throughput, which could be fixed by introducing a wait time between polls. In addition, bidirectional forwarding was not possible for minimal sized packets at full line rate. Further investigations showed that these observed hardware flaws only hold for the Intel X520 cards but could not be reproduced using the newer Intel X540 cards in an otherwise unchanged software environment.

10.4 Possible Future Work

The measurements presented in this thesis focus on the throughput of the investigated framework but neglect other possibilities for comparison with one of them being latency. A tool for performing delay measurements on the MEMPHIS testbed using the NICs' hardware timestamping feature is currently under development. Further tests could be used to extend the model not only to predict the expectable throughput but also to calculate the possible latency of different packet processing scenarios.

Energy consumption is an important property of current IT systems. Nevertheless, the polling nature of the frameworks generating full load on the CPU effectively disables power saving mechanisms. The frameworks offer possible ways to reduce CPU load either by sleeping during blocking calls or by using specialized interfaces to reduce energy consumption. Therefore, measurements could be performed to determine the influence of these mechanisms on power consumption, performance, or latency. The MEMPHIS testbed was recently equipped with new hardware that allows measuring the power consumption of a server, which could be used to investigate these suggestions.

List of Figures

Bibliography

[1] Intel DPDK: Data Plane Development Kit. http://www.dpdk.org/. Last visited 2014-09-10.

[2] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[3] PF_RING ZC. http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/. Last visited 2014-09-10.

[4] Impressive Packet Processing Performance Enables Greater Workload Consolidation. In *Intel Solution Brief*. Intel Corporation, 2013.

[5] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[6] Raffaele Bolla and Roberto Bruschi. Linux Software Router: Data Plane Optimization and Performance Evaluation. *Journal of Networks*, 2(3):6–17, 2007.

[7] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending Networking into the Virtualization Layer. In *Hotnets*, 2009.

[8] Open Networking Foundation. OpenFlow Switch Specification 1.3.4. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf. Last visited 2014-09-10.

[9] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proceedings of SANE*, volume 2004, pages 85–93. Amsterdam, Netherlands, 2004.

[10] Luca Deri. nCap: Wire-speed Packet Capture and Transmission. In *End-to-End Monitoring Techniques and Services, 2005. Workshop on*, pages 47–55. IEEE, 2005.

[11] Luigi Rizzo, Luca Deri, and Alfredo Cardigliano. 10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals. http://luca.ntop.org/10g.pdf. Last visited 2014-09-10.

[12] High-Performance Multi-Core Networking Software Design Options. Wind River, 2011.

[13] José Luis García-Dorado, Felipe Mata, Javier Ramos, Pedro M Santiago del Río, Victor Moreno, and Javier Aracil. High-Performance Network Traffic Processing Systems Using Commodity Hardware. In *Data Traffic Monitoring and Analysis*, pages 3–27. Springer, 2013.

[14] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[15] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. On Multi–Gigabit Packet Capturing With Multi–Core Commodity Hardware. In *Passive and Active Measurement*, pages 64–73. Springer, 2012.

[16] Introduction to OpenOnload-Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. Solarflare, 2013.

[17] Snabb Switch: Fast open source packet processing. https://github.com/SnabbCo/snabbswitch. Last visited 2014-09-10.

[18] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 61–72. ACM, 2012.

[19] netmap-click netmap-enabled version of the Click modular router. https://code.google.com/p/netmap-click/. Last visited 2014-09-10.

[20] netmap-ipfw userspace version of ipfw and dummynet using netmap for packet I/O. https://code.google.com/p/netmap-ipfw/. Last visited 2014-09-10.

[21] n2disk. http://www.ntop.org/products/n2disk/. Last visited 2014-09-10.

[22] nProbe. http://www.ntop.org/products/nprobe/. Last visited 2014-09-10.

[23] Intel Corporation. Intel DPDK vSwitch. https://01.org/packet-processing. Last visited 2014-09-10.

[24] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[25] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 50–59. IEEE Computer Society, 2005.

[26] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2012.

[27] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[28] Jamal Hadi Salim. When NAPI Comes To Town. In *Linux 2005 Conf*, 2005.

[29] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Proceedings of the 5th annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.

[30] ntop. PF_RING User Guide, Version 6.0.0, 2014.

[31] Intel Corporation. Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide, January 2014.

[32] Intel DPDK: Data Plane Development Kit API. http://dpdk.org/doc/api/. Last visited 2014-09-10.

[33] Intel Corporation. Intel Data Plane Development Kit (Intel DPDK) Getting Started Guide, January 2014.

[34] UIO: user-space drivers. http://lwn.net/Articles/232575/. Last visited 2014-09-10.

[35] PCI-SIG. Express base specification revision 2.0, 2006.

[36] Scott Bradner and Jim McQuaid. RFC 2544. *Benchmarking methodology for network interconnect devices*, 1999.

[37] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. 2010.

[38] Ulrich Drepper. Memory part 2: CPU caches. http://lwn.net/Articles/252125/. Last visited 2014-09-10.