Abstract

While software routers are able to provide great flexibility for relatively low cost, they struggle to fulfill the high packet rates which are required for modern 10 Gbit/s or faster networks.

In this thesis, design concepts for software routers are analyzed and an architecture is proposed, which enables high speed software routing, based on modern commodity hardware.

To achieve high flexibility, the router functionality and module interactions are described by the Lua scripting language.

A prototype is developed for conducting performance measurements. The results show, that the proposed router is able to achieve a packet rate of 14.6 Mp/s on a single CPU core, running at 3.2 GHz. This allows saturating a 10 Gbit/s Ethernet port with minimum sized packets to a level of 98%. It is also shown, that the proposed router achieves linear scaling with CPU frequency, as well as with the number of CPU cores, allowing the software router to serve multiple 10 Gbit/s network ports.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
BIRD	Software, implementing routing protocols
C++	Object oriented programming language
CIDR	Classless Inter-Domain Routing
CPU	Central Processing Unit
C	Programming language
DIR-24-8	Algorithm for implementing routing tables
DPDK	Data Plane Development Kit
FFI	LuaJIT library allowing to call external C functions
GbE	Gigabit Ethernet
G	Giga (Billion), 10^9
Hz	Hertz (Unit of frequency)
ICMP	Internet Control Message Protocol
IPng	Internet Protocol next generation aka Internet Protocol Version 6 (IPv6)
IP	Internet Protocol
KNI	Kernel NIC Interface (a part of DPDK)
LPM	Longest Prefix Matching
LuaJIT	Just-In-Time Compiler for Lua
Lua	Scripting programming language
L	Level (L1 Level 1)

MAC Media Access Control (address)

M Mega (Million), 10^6 (Mp/s Million packets per second)

NAPI New API (an approach in the Linux kernel to avoid interrupts)

NIC Network Interface Card

NUMA Non-Uniform Memory Access

No. Number

OSPF Open Shortest Path First (a routing protocol)

Quagga Software, implementing routing protocols

RCU Read Copy Update

RFC Requests for Comments (technical publications for and about the Internet)

RIP Routing Information Protocol

RxB Receive Batch

Rx Receive

ST-II Internet Stream Protocol (RFC1190)

TCP Transmission Control Protocol

TTL Time to live

TxB Transmit Batch

Tx Transmit

UDP User Datagram Protocol

aka Also known as

b/s Bits per second

bps Bits per second

dst. Destination

freq. Frequency

k Kilo (Thousand), 10^3

p/s Packets per second

pps Packets per second

rnd. Random

Chapter 1

Introduction

Routers implemented in software are already widely used for home networking. The two main benefits of these routers are, that they run on cheap commodity hardware and that their functionality can be changed in a flexible way, by modifying the software. This allows a fast reaction time for changing networking needs and adapt to new technology. Because of these advantages, it becomes increasingly interesting, to investigate, how software routers can also be used in bigger networks and whether they are able to compete with commercially available hardware routers.
As currently 10 Gbit/s Ethernet is used in server networks and 40 Gbit/s Ethernet devices start to appear on the market, the challenges for software routers, to fulfill the increased packet rate requirements are getting even harder. Commonly used software routing implementations, which are often based on the networking stacks of operating systems, can not keep up with the demands of these new standards [27, 32].
In this context, numerous research was performed to increase the performance of software routers [2, 8, 25]. However, no solution was yet published which is able to perform software routing on multiple 10 Gbit/s Ethernet ports in a reasonable way.

1.1 Goal of this thesis

The goal of this thesis is to develop an architecture and a software framework, for building software routers on modern multi-core hardware. Thereby, routing of packets with high performance is the priority, while at the same time supporting a maximum level of flexibility.
Additionally a prototype should be developed, implementing the proposed router architecture. This prototype is then to be analyzed in performance tests, to answer the research question, if software routers are able to route minimum sized packets, saturating multiple 10 Gbit/s network ports, while maintaining their flexibility for adding or modifying functionality.

1.2 Outline

This thesis is structured as follows:
Chapter 2 will present some basics for software based packet processing and introduce existing approaches for software routers. Thereby, key techniques are outlined, which will be adapted in the proposed router architecture.
After that, the technologies such as hardware features and packet processing software which are used as a foundation for the proposed router are explained in Chapter 3.
In Chapter 4, first the design decisions and the router architecture architecture is described.
Subsequently, details about the implementation of the prototype are explained in Chapter 5.
Performance tests of the prototype are presented in Chapter 6.

Chapter 2

Background

This chapter provides background information to software based packet processing and also presents the most important approaches which are currently used as software routers. We elaborate the strengths and weaknesses of each solution and hence build up the basis for the software router which was developed as part of this thesis.

2.1 Performance metrics and packet sizes

For comparing different router implementations, a way to measure and represent performance is necessary. The two most widely used performance properties for routers are latency and throughput, as seen in [8,21,25,27,39]). With latency, one associates the time which elapses from sending one packet to the router, until the packet is completely sent out again.
For the throughput, the naive metric is, to measure the bitrate, meaning the number of bits per second (bit/s also known as bps or b/s), the router can process. However, Internet routers are processing packets and usually only need to access packet headers. The amount of payload, a packet contains, has little influence to the router performance for most router implementations[1] (also see [8]). Hence, a more useful and versatile metric is, to measure the performance in number of packets per second (p/s aka pps) which can be processed by the router. The performance in bits per second, can then easily be derived, by multiplying with the average packet size, which is observed, or expected in the application environment of the router (under the assumption, that the payload size has no influence on the router performance).
To empirically measure the throughput of a router, test traffic is generated. Usually minimum sized packets are used as test traffic in benchmarks, to allow a wide range of packet rates, including the maximum possible packet rate for the transmission technique.

[1] This is shown in Section 6.3 for the proposed router.

For Ethernet, 64 byte sized packets are used, which is the minimum allowed packet size in one Ethernet frame, totaling to 84 bytes per frame [36].

2.2 Interface to the NIC

In this thesis, we assume that network interface cards (NICs) are connected to the host system via a bus system like PCIe and are able to write and read to host memory via Direct Memory Access (DMA).
Figure 2.1 displays an abstract layout of the core data structures for the receive part of a NIC and its device driver. In the following, the interaction between the NIC and the device driver, as well as the interface to the network stack is described for the receive process. Transmitting packets can be done in a similar way.
A NIC has at least one transmit and one receive queue. Those queues are implemented as a ring of descriptors, which are pointing to memory locations (Pkt. buffers) in the shared host memory region. Only the receive descriptor ring is displayed in Figure 2.1. This ring has to keep track of the descriptors, which point to memory, already containing valid data (dark gray fields in Figure 2.1) and of the descriptors (light gray) which point to empty memory locations (white). Each incoming packet is written into the memory region pointed to by the current empty descriptor.
The host also has a similar ring of descriptors which is tightly synchronized with the NIC's ring via either interrupts or polling. The first purpose of this ring is to provide references to received packets (dark gray fields), which are passed to the network stack. The second purpose is to pass empty descriptors (light gray fields) to the NIC ring for storing packets. The host thereby continuously has to provide descriptors, pointing to empty packet buffers to the ring (replenish) and synchronize with the NIC.
The situation in Figure 2.1 represents an unsynchronized state of the descriptor rings. The host still thinks, there are three descriptors which point to empty packet buffers,

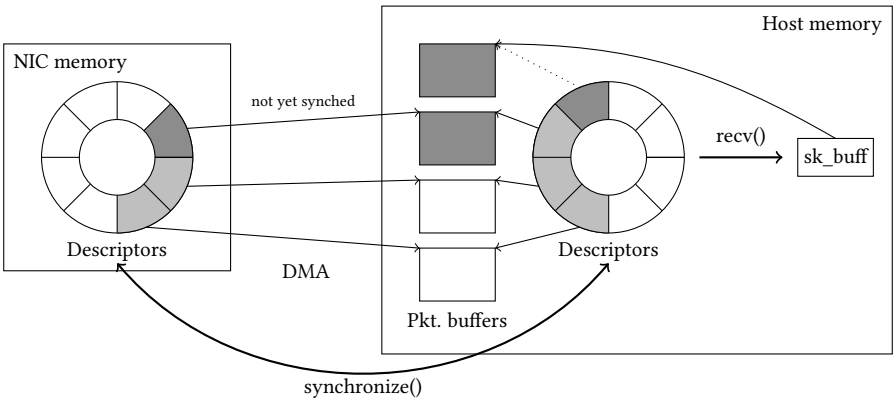


Figure 2.1: Generalized packet receive process.

whereas in reality the NIC already filled one of them. After synchronization the NIC will have no filled descriptors and the host ring will contain two descriptors which point to filled data.
When the network stack requests to receive a packet, a management data structure is created, which describes the packet and also contains various metadata for the packet. This metadata includes results of offloading processes, pointers to higher layer protocol headers, the interface from which this packet was received and timestamps. Eventually this data structure also contains references to one or more (in case of fragmentation) memory locations with valid packet data, taken from the descriptor ring. In Linux this data structure is called `sk_buff` (also see [34]).

2.3 The Linux network stack

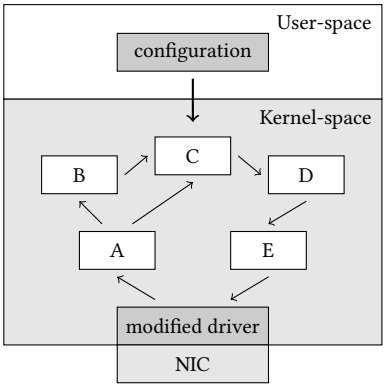
The Linux operating system provides software-based support for layer 3 packet forwarding as an integral component of its network stack. With Linux supporting a wide variety of hardware, cost effective routers can be built using commodity or even embedded hardware.
The Linux network stack is actively developed and optimized for performance. The most notably change being the introduction of the New Application Programming Interface (NAPI). Its focus was to improve performance for high packet rates by reducing interrupt overhead [1]. This is done by not synchronizing (see Figure 2.1) with the NIC using interrupts, but by actively polling in phases of high load.
In addition to the basic forwarding functionality, which is provided by the Linux kernel, many applications like BIRD[2] or Quagga[3] were developed which run on top of Linux to provide advanced routing functionality and routing table management by implementing routing protocols like RIP, OSPF or BGP. These applications do not only utilize the Linux routing functionality, but also higher levels of the network stack like TCP or UDP sockets.
However, as the Linux network stack is designed to be a general purpose network stack, its implementation is not optimized for packet forwarding. The implementation was done with the general use case in mind, causing lots of calculation overhead, which is not needed for pure packet forwarding.
For example in the Linux kernel, packets are represented by the `sk_buff` structure, which is allocated and populated on packet reception (see Figure 2.1). This structure contains meta information about the packet, which is mostly useful for processing it in higher network layers (see [34] for more details about the `sk_buff` structure and its uses).
Another example of unnecessary overhead are locks which have to be acquired, to en-

[2] `http://bird.network.cz/`
[3] `http://www.nongnu.org/quagga/`

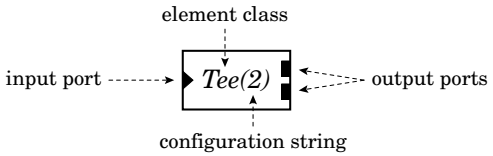
sure operation in the general case, when multiple tasks are accessing the packet queues. This causes significant overhead, even if the queues are never locked, and is not needed for pure packet forwarding [14].
Measurements on the Linux kernel 3.7 running on an Intel Xeon E3-1230 v2 CPU at 3.3 GHz were performed at TUM which showed, that for IP forwarding a maximum packet rate of only 1.67 Mpps for one CPU core is possible [32]. While this is sufficient for routers in smaller networks using 100 Mbit/s or even 1 Gbit/s network links, Linux routers are currently not able to saturate 10 Gbit/s networks ports with minimum sized packets, even when using multiple CPU cores.

2.4 The Click Modular Router

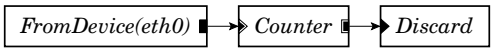
The Click Modular Router is a flexible packet processing framework, which runs on Linux [27]. As shown in Figure 2.2a it uses modified device drivers and all packet processing modules run completely inside the Linux kernel as a kernel thread, replacing its network stack. This avoids switching to user-space and improves performance. A configuration interface exists, which allows controlling the router at runtime from user-space.
Packet processing functionality is split into small functional modules, called *elements*, which perform simple tasks on individual packets. Figure 2.2b shows the interface of a typical element: An element has input ports, on which packets can arrive and output ports, which will be connected to other elements. The element then performs its operation on the received packet and based on this, sends it to one of its output ports.
The complete functionality of the Click system is hence described by a directed graph with the elements as nodes. A simple example configuration of Click is shown in Figure 2.2c. Here packets are received from a network interface *eth0*, counted and then dropped.
The user is able to describe the graph, using a special Click configuration language. Router functionality can then be updated during runtime, by passing the new configuration to the kernel thread.
Click is shipped with a variety of already implemented modules, but provides the possibility to implement own elements as C++ classes.
With Click, a unique approach of modularity was introduced. However, developed in the year 1999, it was not designed with current highly parallelized processor architectures and modern NICs in mind. According to [27] the performance of a Click IP router, forwarding 64 byte packets reached 0.35 Mp/s on an Intel Pentium III processor running at 0.7 GHz. Assuming linear scaling with CPU frequency, we would reach a packet rate of about 1.65 Mp/s on a CPU running at 3.3 GHz. This is roughly the forwarding rate of a pure modern Linux router (see Section 2.3) and is consistent with the fact, that the main performance benefit of the original Click was based on modified device drivers to



(a) Task distribution within Linux.



(b) Element interface (Source: [27]).



(c) Simple graph example (Source: [27]).

Figure 2.2: Click router structure.

support polling, which became standard in modern Linux systems with the introduction of the New Application Programming Interface (NAPI).
Another problem with the Click design is, that it runs inside the Linux kernel and also requires some modifications to the kernel, to improve scheduling [27]. This leads to problems when porting Click to other Linux kernels and extending the functionality by writing new elements, as debugging in a kernel environment is hard and unstable code makes the whole Linux system unstable. To partly overcome this problem, Click contains an user-level driver, which utilizes Berkeley packet filters and allows running Click in user-space. However, this method is intended for debugging only as it heavily influences performance.
In the past numerous research was done to improve the performance and stability of the Click Modular Router [6, 8, 25, 26, 33, 39]. Some of them are described in the following sections.

2.5 Batching with Click

Two projects [21, 25] explored the possibility to amortize costs for handling packets by performing tasks on batches consisting of multiple packets [25]. Thereby, two key problems of the Click framework were isolated:

1. The way, Click handles reception of packet causes much similar calculation, which is done for every packet. As Click uses Linux data structures [21], for every received packet, a packet buffer has to be allocated and an associated `sk_buff` data structure has to be filled with metadata about the packet (see Figure 2.1).

2. After receiving a packet, all necessary processing steps (traversing the element graph) are performed and eventually, the packet is sent out again, or dropped.

This approach yields bad branch prediction and cache performance, as firstly each processing step requires other program code to be run and secondly, different data will be required in different processing steps, which might not yet be in the cache. Also initialization and passing of packets to other Click elements has to be done for each packet separately.

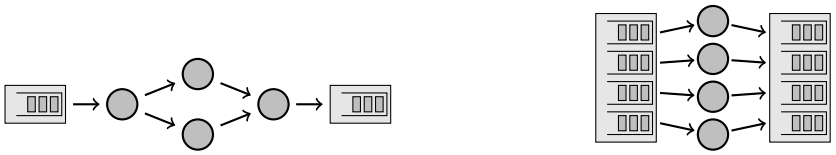
The first problem was addressed, by not using the Linux data structures, which are intended for a general purpose network stack (see Section 2.3). Instead a new packet I/O engine was developed, as part of the PacketShader project [21]. This engine supports lightweight data structures, which only store a minimum set of metadata. However, these data structures still need to be allocated. To overcome the high allocation and deallocation cost, packet buffers and metadata structures are allocated in batches for multiple packets at once. As this approach is amortizing costs for packet input and output, it is called *I/O batching* [25].
The second problem is reduced by performing *Computation Batching*. This means, that each processing step is not applied to single packets, but processes batches of packets in each invocation. Hence, initialization costs are amortized among multiple packets and CPU, as well as cache utilization is optimized.
As claimed in [25], combining these techniques, a performance improvement by a factor of 10 can be achieved, depending on the number of packets in a batch (batch size). In a performance test, the project team used two quad-core Intel Xeon X5550 CPUs running at 2.66 GHz per core and achieved a routing throughput of up to 41.7 Mp/s.

2.6 Route Bricks

The Route Bricks project, as described in [8] adapted the Click Modular Router (see Section 2.4) to modern Server systems, which rely on extensive multi-core and multi CPU architectures. The modifications to the original Click mostly consist of added functionality, to bind Click elements to specific CPU cores.
As part of the project, three typical scenarios in parallel packet processing were identified. For all three, multiple architecture implementations were evaluated and optimized. Figure 2.3 shows two methods for implementing an architecture, in which packet processing work should be split among multiple processing cores. In the examples, packets from two NICs should be processed and forwarded to two other NICs. Both implementations use two CPU cores (gray filled circles). The method, displayed in Figure 2.3a passes all packets through both cores, each doing half of the processing work in a pipelined fashion. However, it was shown, that the second approach (Figure 2.3b), where each core only works on half of all packets, but is doing the complete processing by itself, is faster.
As the previous comparison showed, it is desired, to split a stream of packets to multiple cores, which then can work in parallel. However, this requires a solution on how to

(a) Pipeline processing (1.2Gb/s). (b) Parallel processing (1.7Gb/s).

Figure 2.3: Pipeline processing versus parallel processing [8].



(a) Split by core (1.6Gb/s). (b) Use multiple NIC queues (5.3Gb/s).

Figure 2.4: Split and merge traffic [8].

(a) Merge on core (0.7Gb/s). (b) Merge, using multiple queues (1.7Gb/s).

Figure 2.5: Overlapping paths [8].

do the splitting in case of unbalanced load on the NICs, or in the case, when there are more CPU cores available, than NICs. The approach, shown in Figure 2.4a uses one separate core for distributing the packets to the working units and another core, for merging the streams again. Modern NICs often support multiple queues for receiving and transmitting packets. The NIC thereby can be configured, to automatically handle the distribution and merging of packets to and from these queues (for details see Chapter 3.1.1). Figure 2.4b shows an architecture, where this feature is exploited, by binding one CPU core to each queue of the NIC, enabling a performance increase of over 300% compared to the traditional approach. Hence, the designated cores for distributing and merging packets represent a bottleneck.

One last scenario, analyzed by the Route Bricks project, is the situation, when two streams of packets, which are handled by different cores, have to be merged (in fact, this is similar to the merging part of the previous architecture), to be sent out over one NIC. The usual approach, to bind the NIC to one core, and let this core handle all traffic, which has to be sent to this NIC is displayed in Figure 2.5a. In this scenario, the multi queue feature of modern NICs can again be exploited, as it allows binding multiple cores to the same NIC, which is doing the merging in hardware. Using this feature, again gives a performance boost of over 200% compared to merging on one of the processing cores.

Utilizing all this findings and also implementing computation batching, as described in Section 2.5, the Route Bricks project realized a minimal software router as an example

for their Click adaption. Benchmarks showed, that it is possible to route packets at up to 12 Mp/s on a Server, containing two Intel Nehalem CPUs with 4 cores each, running at 2.8 GHz. This is a significant improvement over the Linux and standard Click forwarding performance (see Section 2.3 and 2.4) which would allow the router to be used in gigabit networks, serving multiple ports. However, saturating a 10 Gbit/s link with minimum sized packets is not yet feasible with Route Bricks on current hardware.

2.7 Click with netmap

Netmap is a framework, which allows user space applications to send and receive packets in a fast manner. This is done by avoiding or amortizing costs, which are commonly present in general purpose operating system network stacks. The key techniques, used in netmap are [33]:

- Compact packet metadata structure, instead of `sk_buff` (see Section 2.3).
- Preallocated packet buffers (see Figure 2.1, as well as Section 2.5).
- Direct access of user-space applications to packet buffers.
- Support for batch receive and transmit.

For a wide application support, netmap also provides a `libpcap`[4] API interface, beside netmap's own high performance API.
As described in Section 2.4, it is possible to run Click in user-space at low performance. Using its `libpcap` interface, the netmap project succeeded in running the original user-space Click router on top of netmap (see Figure 2.6). This reduces the stability problems of kernel-space Click, as described in Section 2.4 and at the same time boosts performance. A performance boost of over 800% was measured by [33], when using Click on top of netmap, compared to the original user-space Click for forwarding packets, which is even faster than in-kernel Click. However no detailed routing performance tests were performed.

[4] http://sourceforge.net/projects/libpcap/

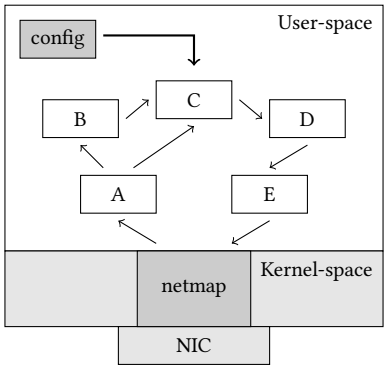


Figure 2.6: Click running on top of netmap.

Chapter 3

Foundations

In this chapter, technology is introduced, which is used as integral components and building blocks of the proposed router.

3.1 Modern high performance NICs

Modern NICs provide features which support software based packet processing. In general, two approaches are used: Firstly the interface to the host system is optimized for modern multi-core architectures and secondly common processing tasks can be directly performed on NIC hardware (offloading).
In the following, key features of modern NIC controllers, such as the Intel X540 are presented, which are used in the proposed router.

3.1.1 Multiple hardware queues

To optimize multi-core packet processing, the Intel X540 NIC supports 128 independent Rx queues, on which packets can be received, as well as 128 independent Tx queues, to which packets can be sent [22]. These queues can be bound to individual CPU cores, giving each core exclusive access to its bound NIC queues. As described in Section 2.6, this simplifies parallelization and at the same time increases performance.
Packets transmitted to the individual Tx queues are merged into one packet stream on the NIC for transmission.
Multiple methods for assigning packets to Rx queues are supported by the X540 NIC. The following two of them are used in the proposed router:

Filters—Different filters can be defined, which match packets according to their contents. Based on the filter rule, packets are assigned to a specific Rx queue (also see hardware

filters in Section 5.1.6). This allows classification of packets to be offloaded to the NIC hardware, which already is used by other software routers [13, 40].

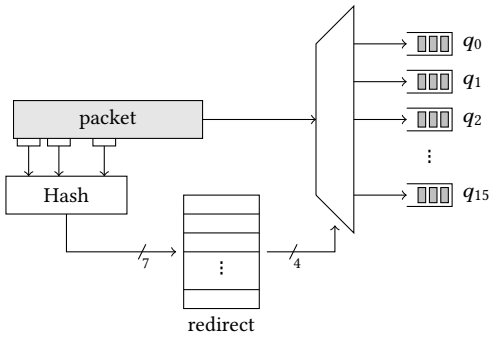


Figure 3.1: Simplified RSS working principle.

RSS—Receive Side Scaling (RSS) is a method to fairly distribute incoming packets to a set of Rx queues. Figure 3.1 shows the simplified working principle of RSS.
Incoming packets will first be classified according to their L3 and/or L4 protocol type. Depending on the protocol type, different fields of the protocol headers are extracted[1]. The extracted fields are given as an input for a hash function. Seven bits of the hash result are then used to index a user defined redirection table. Each of the 128 entries in this redirection table refers to one of the first 16 Rx queues.
This approach ensures, that packets corresponding to one traffic flow (for example all packets belonging to one TCP connection) are always assigned to the same Rx queue.

3.1.2 Checksum offloading

The Intel X540 NIC supports the automatic calculation of both IP and Ethernet checksums. On the receive part, the checksums of incoming packets are calculated and compared to the received checksum. The Ethernet checksum can be stripped from the packet before passing the packet over to the host system. The result of higher level checksum calculation is stored in the packet metadata.
When transmitting packets, the packet metadata is checked for the packet type, and if IP checksum offloading should be done. If requested and possible for the packet type, the checksum is then calculated and automatically inserted, after the packet was transferred from host memory to the NIC.

[1] The extracted fields usually are the address fields of the protocol headers. See [22] Chapter 7.1.2.8.1 for details.

3.2 Data Plane Development Kit

The Data Plane Development Kit (DPDK) is a framework, developed by Intel. Its goal is to support high speed user-space packet processing. Therefore it provides the following features:

NIC drivers—DPDK includes NIC drivers which can be used from user-space, similarly to the netmap framework introduced in Section 2.7. Those drivers are optimized for performance. Hence, only polling for packets is supported (Poll Mode Driver), to avoid interrupts and the drivers were designed with I/O batching in mind [12]. DPDK utilizes a small kernel module for device initialization and PCIe communication, but most of the driver runs in user-space [17].
For representing a packet, no separate data structure, like the `sk_buff` structure, is used, as it is known from the Linux kernel (shown in Figure 2.1). Instead, a fixed size packet buffer format, called `mbuf` is used which only includes the absolutely necessary metadata about the contained packet, as well as the packet data itself [11] (see Figure 3.2). Some space in the `mbuf` is reserved for extending the packet, or other user defined data, such as additional metadata. This approach eliminates the need for separately allocating packet buffers and packet metadata, as it is done in the Linux kernel and in netmap. Due to the small size of the metadata compared to the Linux `sk_buff`, memory overhead is avoided further improving cache efficiency.

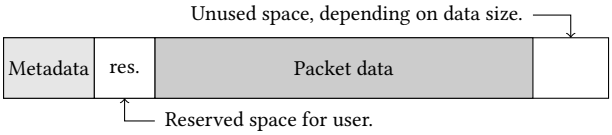


Figure 3.2: Mbuf packet buffer structure.

Packet processing library—In addition to the high performance NIC drivers, DPDK provides an extensive library of packet processing functionality. In this library, efficient cache optimized data structures like rings, hash tables and a Longest Prefix Matching (LPM) algorithm are implemented.

Measurements, performed in [18], show that DPDK provides a significantly better performance than netmap. Because of this and the included packet processing library it was used as a foundation for the proposed router.

3.3 MoonGen

MoonGen, as presented in [15], is a software packet generator based on DPDK. It allows saturating multiple 10 Gb/s network links with minimum sized packets. Thereby, a very flexible definition of the whole packet generation logic is possible via user defined Lua scripts. For efficient execution of these scripts, the LuaJIT Just-In-Time compiler[2] is used which provides high performance for packet processing tasks, as shown in [28]. MoonGen provides an API which includes interfaces (wrappers) to useful DPDK device configuration, as well as packet processing functions. This interface to DPDK functionality is realized, by using the FFI library[3], which allows calling external functions, implemented in C from inside Lua scripts.
Typically two different kinds of Lua scripts are required for running the packet generator (see Figure 3.3):

- One master script, which is executed at MoonGen startup. This script can be used to initialize the NICs and to spawn multiple instances of slave scripts, which are bound to individual CPU cores by DPDK.
- One or multiple slave scripts, which implement the packet generation or measurement functionality, using the API provided by MoonGen.

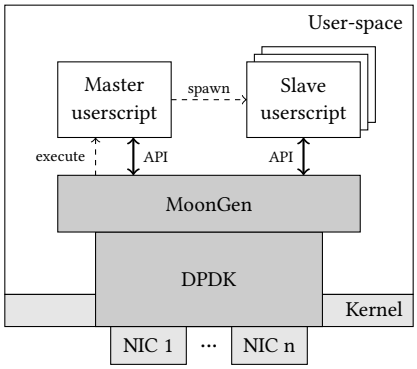


Figure 3.3: MoonGen architecture.

As this approach has shown to provide high performance while maintaining a maximum level of flexibility it is used as a basis for the proposed router. As part of this thesis, MoonGen was extended with functionality to support packet processing, with IP forwarding in mind.

[2] http://luajit.org/luajit.html
[3] http://luajit.org/ext_ffi.html

Chapter 4

Design of the router

As part of this thesis, a novel architecture for software routers was developed and a prototype was implemented.
In this chapter, design decisions for the proposed router are introduced. First the requirements and important goals are listed. Subsequently, important concepts are explained and the router architecture is elaborated in detail.

4.1 Requirements / design goals

Besides implementing the routing functionally as closely as possible to the RFC1812 [4] standard[1], the following additional requirements for the router were determined:

- High performance. One server, using currently available hardware, should be able to perform routing, saturating multiple 10 Gbit/s Ethernet ports.
- Scaling with multiple CPUs or CPU cores. As the trend in processing hardware moves from multi-core to many-core architectures, the router should be able to utilize the benefits of this development.
- High level of flexibility. It should be possible to change and add functionality easily. If possible, the already existing concept for flexibility, presented in MoonGen [15] should be adapted.
- Possibility to reuse already implemented algorithms (e.g. routing algorithms), with little or no need to adapt them to the proposed router.

[1] As the proposed router is a proof of concept, the core concept of a router described in RFC1812, but not all specified requirements are implemented in the current version.

4.2 Design principles

During the design of the router, some key problems and techniques were isolated, which arised, when trying to fulfill the design goals. Those problems usually represent common scenarios in the way a router does packet processing. The solutions to these problems are important design decisions, which affect the working principle and performance of the router. In this section, the most important design principles are elaborated.

4.2.1 Flexibility

One of the key motivations for introducing software components to networking, is increased flexibility, compared to hardware solutions.
The Click router, presented in Section 2.4 is popular because of its modular structure. However, this flexibility is restricted to modifying the graph, interconnecting the functional modules. To achieve this, a special configuration language was introduced with Click. While this allows fast reconfiguration, adding new functionality to a Click router is only possible by writing new Elements in C++.
In the proposed router, a novel approach is introduced. The whole router is implemented, using the Lua scripting language, allowing both: Easy adding of functionality, as well as the use and interconnecting of pre-defined modules. This allows fast and flexible prototyping, by changing or adding core functionality directly in Lua code, as well as efficiency, when using performance optimized modules, which may be implemented in C. In the latter case, Lua code acts as glue, joining the modules together and defining the packet flow.
As part of this thesis, performance critical modules were implemented (see Section 5.1). They form the basis for a packet processing framework, suitable for routers.

4.2.2 Parallel processing

As described in Chapter 2, adaption of packet processing frameworks to modern parallel computing environments is one of the key strategies for gaining performance. However, as seen in the Linux network stack [14] or Click parallelization approaches [6, 25, 39], synchronization of parallel processing units heavily reduces possible performance gains - mostly due to locks.
To achieve optimum scalability with an increasing number of available processing cores, the *Shared-Nothing* principle is used as a reference. This principle is known from distributed computing architectures in the web and database disciplines [38]. As however typical server systems have multiple processing units, sharing memory at some level, the *Shared Nothing* concept was adapted, to this scenario. This is also known from the Seastar project, as presented in [37].

The following design principles are respected throughout the design:

1. Use synchronization only when absolutely necessary.
2. Use lock-free queues for message based synchronization.
3. Avoid working on shared data structures.
4. If working on shared data structures, use immutable data structures, which are not written to, after creation.
5. Utilize local non-shared caches.

4.2.3 Handling of complex packets

The main task of a router is obviously to forward packets. The basic workflow for this task is to look up a route for each packet, update packet headers according to the lookup and eventually send the packets out. However, there are multiple tasks, which are directly related to forwarding, which require a different and often more time consuming workflow. Some examples are listed in the following:

ARP: To be able to forward a packet, the MAC address of the next hop has to be known. However, as routing rules are typically given with the IP address of the next hop only, ARP queries have to be performed at some time, to resolve this IP address to a MAC address. Therefore a table, which associates IP addresses with MAC addresses (ARP-table) has to be managed.
Vice versa the router has to reply to ARP requests from other hosts with its own MAC addresses, to enable reachability within the networks it is connected to.

Routing protocols: In the most trivial case, a router has a predefined, static set of rules, which specify how to forward packets. In the Internet however, routing is done in a more dynamic way, as a router has to continuously adapt to a changing network topology. Hence, various routing protocols like BGP, OSPF or RIP exist, which enable communication between routers, with the goal to exchange route information.

ICMP replies: According to RFC1812 [4], a router must or may send ICMP messages on several occasions. For example, when a packet arrives with a destination address, for which the router has no route, a *Network Unreachable* ICMP packet has to be sent back to the sender. Another example are packets with an expired TTL value. In this case, a *Time Exceeded* message has to be sent to the sender.

In the following, we refer to all packets, which require different processing, than successfully routable packets as *complex packets*. Figure 4.1 shows the trivial way of handling complex, as well as non-complex packets in a router:
At some point during processing a decision is made, if the packet is a complex packet.

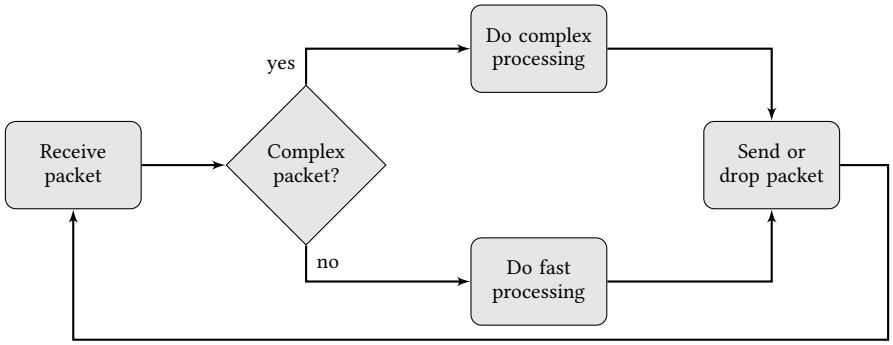


Figure 4.1: Handling of complex packets.

In the figure, this decision is made directly after reception of the packet for simplicity. In case of a non-complex packet, normal forwarding is performed (*Do fast processing*) and the packet is eventually sent to the next hop. If however a complex packet is detected, execution continues with the complex packet processing (*Do complex processing*), which eventually may result in the transmission or dropping of a packet.
This approach brings disadvantages, worsening the router performance:

- As the complex processing path takes more time to execute, latency for non-complex packets will increase, as they have to wait for the complex packets to finish processing.
- The goal of a router is to forward packets. However, as in this approach no prioritization is done for forwardable packets, non forwardable, complex packets consume valuable processing time and might eventually starve the router.
- Switching between code for fast and complex packet processing has negative effects on the instruction and data caches, worsening performance.

In classical hardware-based or mixed router architectures, it is common practice, to have a logical and in most cases even physical separation between complex packet processing and simple packet forwarding into so called *control plane* and *forwarding plane* [24, 29, 35]. Thereby, the forwarding plane is implemented in hardware, whereas the control plane is implemented in software.
The situation described in Figure 4.1 leads to the conclusion, that a separation like this also makes sense for pure software defined routers, to maximize throughput. As shown in Figure 4.2, we propose a separation into a *Fast Path* processing unit and a *Slow Path* processing unit, which run concurrently on different CPU cores. Communication between these two software modules can be implemented in the form of lock-free queues (dashed arrow in the figure). When during handling of a packet, it is determined, that the packet is a complex packet and requires more processing, it will be merely enqueued for processing by the Slow Path. Execution in the Fast Path module will then immediately continue with receiving the next packet, improving forwarding throughput.

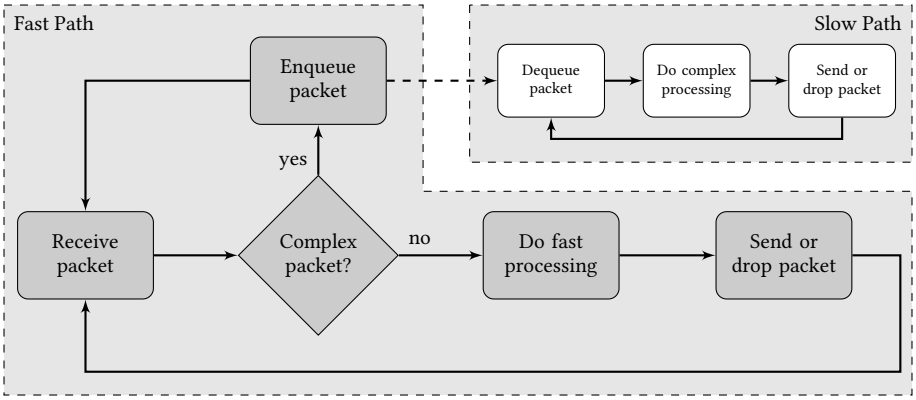
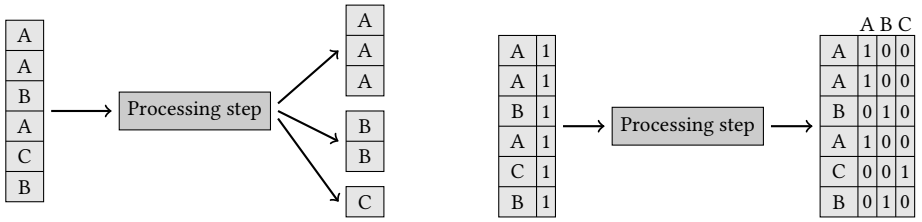


Figure 4.2: Block free handling of complex packets.

Prioritization or overload countermeasures can easily be implemented by changing parameters or functionality of the queues to the Slow Path units. For example active queue management, like a random dropping algorithm similar to RED [5], will allow fair Slow Path traffic also in overload situations. Again Figure 4.2 only shows a trivial example, where packets are classified directly after reception. In reality, multiple branches to the Slow Path module at different processing stages can be used.
It is also possible, to bind distinct receive queues of the NIC to the Slow Path module, which allows partial packet classification by hardware filters of the NIC. In this case, the Slow Path module has to serve multiple queues and arbitration between these queues has to be performed.

4.2.4 Batch scattering

The idea of computation batching is to amortize costs and optimize cache performance, by performing processing in multiple steps on batches of multiple packets. Typically a batch is represented by a list of references to packets, stored as an array. After initial creation of a batch, it is passed to different processing modules.
With that however the problem arises, that different packets in one batch might require different processing. For example invalid packets should be dropped, packets destined to the router itself, should be handled correctly and packets with expired TTL should trigger a ICMP message.
In existing packet processing frameworks like Click (described in Section 2.4), the required processing steps and their order is determined by, following paths in a graph for any given packet. Hence, different packets take different processing paths.
The consequence is, that new batches of packets have to be created after each processing step for all possible paths, the packets might take after this step. The original batch hence will be split into many smaller batches, depending on the number of processing steps and possible paths. In this thesis, we call this *batch scattering*. Figure 4.3 shows



(a) With lists. (b) With bitmasks.

Figure 4.3: Batch scattering.

methods for implementing batch scattering. In the example, a batch of six packets is processed by one processing step. The displayed processing step distinguishes between three classes (A, B and C) of packets, each requiring different further processing. In the following, different batch scattering methods are compared:

1. The first method, shown in Figure 4.3a, only uses lists with references to packets. While processing the batch, three lists are generated, one for each packet class. This lists can then be used as an input for further processing steps.
   The advantage of this approach is, that each processing step works on a very compact list of packets. However, additional memory for the result lists is required, which reduces cache efficiency. Additionally memory copies are required for each packet, to copy the references into the corresponding result list.

2. The approach, shown in Figure 4.3b uses bitmasks, to specify on which packets of the batch, a processing step should operate. During processing, a bitmask is generated for each packet class, masking the corresponding packets.
   As the memory requirement for bitmasks is very small compared to lists with references, this approach provides better cache performance than the first method and does not require reference copies, as the same list of references is used in each processing step. However, the whole batch has to be iterated in each step and checked for masked packets. As this has to be done, even when only few packets of the batch need to be processed, processing overhead and an increased number of processor pipeline flushes due to branch prediction errors[2] are introduced.

3. A third approach, not shown in the figure, is a variation of the first approach shown in Figure 4.3a. Thereby the resulting lists are not immediately used for further processing, but only, when they are filled with a minimum number of packets. Hence, packets from multiple original batches might be accumulated. This introduces even more management overhead, than the first approach, but allows greater cost amortization in the subsequent processing steps. In the following this method is called *rebatching*.

[2]The programmer has to give a hint to the compiler, if packets usually are masked or not masked for a specific processing step (branch likely, or unlikely to be taken). If the input batch, however does not fulfill the expectations, the processor has to perform many time-consuming pipeline flushes.

In the proposed router, the third approach is implemented for one processing module (see Section 5.1.5). All other modules use a hybrid batching method, influenced from both the first and the second approach. We call this approach *Drop-out batching*. Thereby, the negative effects of both approaches could be compensated by inducing some router design restrictions, which have little impact on performance, as long as the following assumptions hold[3]:

- Packets, requiring high processing performance, traverse the same processing path (meaning the same set of processing steps in the same order).
- Packets, which require different processing paths than the previously mentioned path, may be treated with low priority or dropped.
- Most of the packets require high processing performance or may be dropped.

Drop-out batching works by using bitmasks for processing along the single high performance processing path, avoiding scattering and queues (*Drop-out queues*), which accumulate all other packet classes. Packets in these queues can then be separately processed, using one of the traditional batching methods, or no batching at all. Figure 4.4 illustrates the working principle: In the example, again three packet classes are distinguished. This time, it is assumed, that all packets, labeled with A in the figure belong to the dominant class, requiring high processing performance, whereas packets belonging to the classes B and C can be processed with low priority. The processing step processes all masked packets in the batch, and will generate a bitmask (which may use the same memory location as the input bitmask) masking all packets, corresponding to class A. In this example, all other packets are enqueued in a single queue for separate processing.
This method works hand in hand with the architecture of Slow and Fast Path processing, proposed in Section 4.2.3. In this case the high performance path is processed as the *Fast Path*, using bitmasks only and the low priority packets are redirected to the *Slow Path*, by using the queue to the Slow Path as a drop-out queue for the Fast Path processing steps.

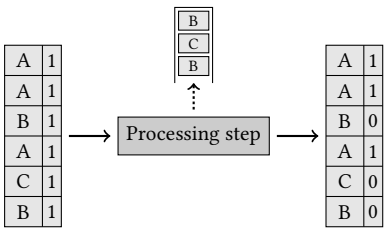


Figure 4.4: Drop-out batching.

[3] All modules can easily be modified, to support batch scattering with bitmasks, or multiple Drop-out queues, if the application fails to hold the assumptions.

4.3 Architecture

The proposed router is composed of three different types of software units, which are described in detail in the following sections:

Fast Path: Performs fast packet forwarding according to forwarding rules, specified in a routing table. Packet processing is done in batches using multiple processing steps $(p_1, p_2, ..., p_n)$.

Slow Path: The Slow Path handles multiple tasks, which can be split into separate units on demand:

- Handle packets, which can not be processed by the Fast Path (ICMP, ARP, Broken Packets).
- Perform synchronization for fast and Slow Path units (Manage shared data).
- Provide a user interface.
- Exchange information with the underlying Linux operating system.

Linux: The underlying operating system, providing a general purpose network stack for advanced packet processing and protocols.

All three components are threads, which are scheduled by the operating system and run asynchronously. Hence, they are also referred to as *Asynchronous units*.
To allow scaling with an increasing number of CPU cores, multiple identical instances of Fast Path units are deployed. Thereby, each Fast Path unit is bound to a distinct CPU core.
In addition to the software units, the router also contains one or multiple Network Interface Cards, each providing one or multiple connections to a physical network (port). Part of the packet processing tasks can be offloaded to the NIC hardware, which is why NICs are also considered as asynchronous processing units. We thereby split the functionality of a NIC port wise in receive related functionality (Rx port) and transmit related functionality (Tx port).
In the following, we define the number of NICs, which receive packets to route as n_{rx} and the number of NICs, to which packets are routed as n_{tx}. The number of Fast Path processing units shall be n_f. An instance of the router can then be characterized by the triple (n_{rx}, n_{tx}, n_f). Figure 4.5 shows an example of the router architecture with a $(1, 2, 2)$ configuration.

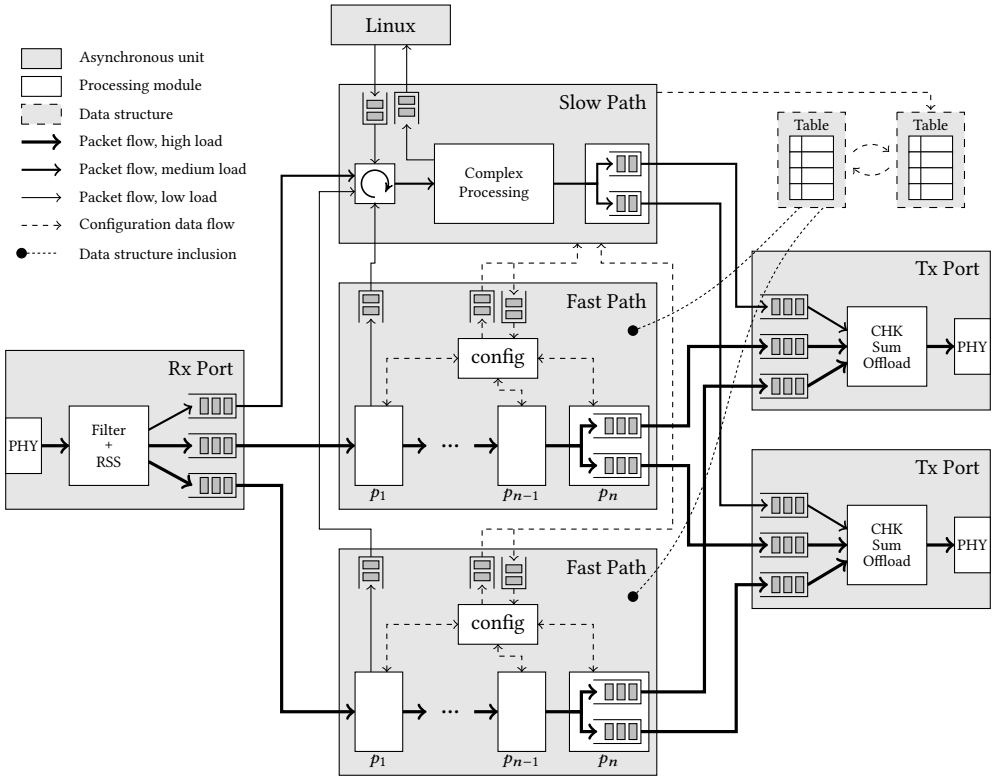


Figure 4.5: General router architecture.

In the following, the interactions of the router components and data structures are illustrated for a generic configuration.

Rx port—On each Rx port, incoming packets are distributed to different Rx queues. Each queue is then bound to one specific asynchronous unit. Hence, in total $n_f + 1$ Rx queues of a port are used, one for the Slow Path and one for each of the n_f Fast Path units.
For the queues, associated to the Fast Path units, packets are distributed, using the hash based RSS technique (described in Section 3.1.1). This ensures equal load on all Fast Path units and at the same time prevents reordering of packets, which belong to the same traffic flow.
Packets for the single queue, bound to the Slow Path unit, are determined by hardware filters, configured from the Slow Path itself.

Tx port—Similarly to the Rx ports, on each Tx port n_f+1 Tx queues are used, one for each asynchronous unit. This avoids synchronization when asynchronously transferring data from the units. Packets in these queues are merged by the NIC and checksum offloading is performed, before the packets are sent to the network.

Fast Path—Each Fast Path unit reads packets from its dedicated Rx queue on each Rx port. In case of a router configuration with multiple Rx ports, packets are received from all n_{rx} Rx ports in a round robin fashion. In Figure 4.5 arbitration is not shown, as in this example, only one Rx port is used.
The last processing step p_n in a Fast Path unit is a distribution step. In this step, packets are distributed to the n_{tx} transmit ports, using the respective Tx queue, which is associated with the corresponding Fast Path instance.
Communication with the Slow Path is done via single producer, single consumer lockless queues:
As *Drop-out batching* (see Chapter 4.2.4) is performed in the Fast Path, a dedicated packet queue is available in each Fast Path instance. This is used to forward packets to the Slow Path unit for further processing.
Additionally each Fast Path instance has two queues, for passing and receiving configuration messages to and from the Slow Path unit. These are used for data exchange and synchronization with the Slow Path unit.

Slow Path—The Slow Path unit incorporates a round robin arbiter, for receiving packets from its dedicated Rx queues on all n_{rx} Rx queues, all Fast Path instances, as well as packets from the Linux network stack.
Like with the Fast Path units, packet processing in the Slow Path ends with a distribution step, distributing packets to all n_{tx} Tx ports.

Routing table—As displayed in Figure 4.5, one routing table is shared among all Fast Path instances. This is a deviation from the *Shared-Nothing* principle, introduced in Section 4.2.2, which improves performance with typical memory hierarchies, seen on modern CPUs. The Intel Xeon CPU for examples implements a three stage cache hierarchy. Thereby, each CPU core has its own Layer 1 and 2 cache, whereas the 8 MByte L3 Cache is shared among all cores.
As the Fast Path units only perform read operations on the routing table, no cache synchronization is required for the L1 and L2 caches. Hence, a shared routing table yields the same L1 and L2 utilization, as a dedicated routing tables for each Fast Path unit.
However, because the L3 cache is assumed to be shared among all cores, having dedicated routing tables, which require a minimum of 32 Mbyte each (see Section 5.1.2 for details on LPM table memory consumption), introduces redundancy, which reduces the L3 cache performance. As an example, assume two Fast Path units have to route a packet to the same destination address, at roughly the same time. When both units have an own (identical) copy of the routing table, two cachelines in the L3 cache are filled with the same data, wasting space.
Because of this effect, a shared routing table is expected to provide better L3 cache

performance, leaving the L1 and L2 cache performance untouched, as long as on the shared data no writes are performed.
The Slow Path unit is responsible for updating and synchronizing the routing table with the clients. However, to avoid locks and writing to shared data structures, a so called *double buffering* approach is used. The Slow Path unit thereby maintains a second routing table, to which updates are written, while the other table is used by the Fast Path units. On demand, the two tables are switched, which is communicated and synchronized via the configuration queues (Details on the implementation see Sections 5.3 and 5.2).
While this approach is similar to the Read Copy Update (RCU) mechanism, seen in the Linux kernel [30], the proposed approach does not perform a copy of the routing table before updating, but re-generating the routing table from scratch. Hence, an existing routing table is exclusively used by the Fast Path units, once it is deployed.

Chapter 5

Implementation of the router

In this chapter, details about the implemented prototype of the proposed router architecture and its building blocks are explained.

5.1 Packet processing modules

In the context of this thesis, multiple processing modules, which can be used in packet processing contexts, were developed. Most of the functionality is provided by the DPDK library and was modified, to support bitmask-based batch processing. In this thesis, a processing module provides the functionality of a processing step. The functionality of the router is then defined by the sequential execution of multiple processing steps.
All modules are implemented in a mixture of C and Lua code and provide a documented interface for use in the Lua programming language[1]. For most of the modules multiple implementations are available. These different implementations typically vary in the fraction of used C code with respect to the used Lua code.

5.1.1 IP validity checking

According to RFC1812 [4], it is required for a router, to perform the following validity checks on incoming IP packets (excerpt from [4]):

1. The packet length reported by the Link Layer must be large enough to hold the minimum length legal IP datagram (20 bytes).
2. The IP checksum must be correct.

[1] One Lua module not necessarily provides one processing module, but may provide implementations for many processing modules. Also processing modules may be split over multiple Lua modules.

3. The IP version number must be 4. If the version number is not 4 then the packet may be another version of IP, such as IPng or ST-II.
4. The IP header length field must be large enough to hold the minimum length legal IP datagram (20 bytes = 5 words).
5. The IP total length field must be large enough to hold the IP datagram header, whose length is specified in the IP header length field.

This module implements the mentioned checks according to the requirements[2]. Thereby, the checks number 2, 3 and 4 are offloaded to the NIC. In these cases, the implementation merely checks the results, provided in the packet metadata.

5.1.2 Longest Prefix Matching

Using this module, it is possible to perform fast Classless Inter-Domain Routing (CIDR), as described in [16]. The module is based on the LPM library, provided by DPDK [10], which was modified to support bitmasks and batches. In the following, the working principle of the lookup algorithm will be explained.
The used algorithm is a variation of the DIR-24-8 architecture, which was originally described by P. Gupta [19, 20]. The basic idea of this algorithm is, to achieve increased lookup performance, by an increased use of memory.
The functionality of an CIDR algorithm is to provide a table, which accepts an IP address as an index and returns next hop information. The trivial approach would be to implement a lookup table for all possible IP addresses and use the IP address as an index to this table, explicitly linking each possible address to its next hop.
This table can be implemented as an array, allowing easy indexing via memory location. Hence, each lookup would require a simple arithmetic calculation for the memory address and one memory lookup.
However, the table would contain 2^{32} entries for IPv4 addresses, which is possible with current hardware, but will not be very cache efficient.
Instead, the lookup table is split into instances of three different types of tables (see Figure 5.1). Thereby, `tbl24` and multiple instances of `tbl8` form a trie with a depth of two. The third one (`nextHop`) is used, to store information about the route to be taken.

[2] In the current version, which was used for all benchmarks, the last check deviates from the required standard, as the IP total length field is not compared to the IP header length field, but against 20.

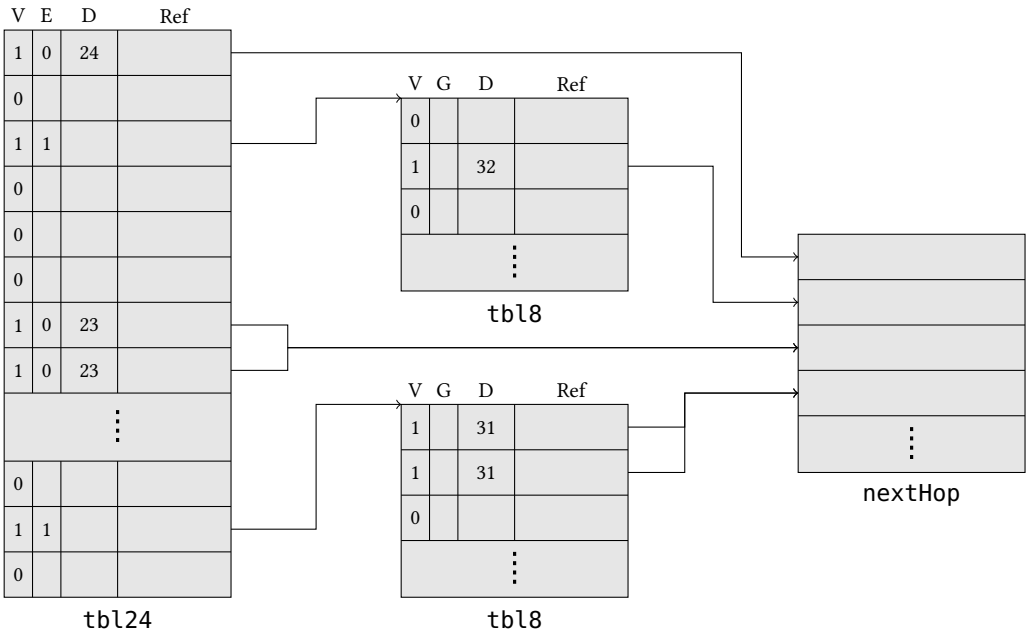


Figure 5.1: DIR-24-8 algorithm data structures.

A lookup for a certain IP address is then performed in two steps:

1. The 24 most significant bits of the IP address are used as an index for the `tbl24`. If the valid bit (V) of the indexed entry is not set, no route can be found for this IP address.
   If the extend bit (E) is not set, the next hop information can be found in the table `nextHop` at the index, stored in the reference (Ref) field. The lookup is finished.
   If however, the extend bit (E) is set, the following second step has to be performed.

2. The 8 least significant bits of the IP address are used as an index for the `tbl8`. If the valid bit (V) is set, the next hop information can be found in the table `nextHop` at the index, stored in the reference (Ref) field. The lookup is finished.
   Otherwise, no route for this IP address can be found.

Adding routes to the data structure has to guarantee the priority of rules with longer network prefixes. For CIDR, routes are identified by a network IP address and a network prefix, specifying the number of significant bits in an IP address (depth of the route), which define the network.
When adding a new route, the following is done:

- The `nextHop` table is searched for an entry, matching the route destination information. If no matching entry is found, a new one is created. The received entry will later be referenced in the `tbl24` and if neccessary in the `tbl8` tables.

- If the depth d_r of the route to add is smaller or equal to 24, all 2^{24-d_r} matching[3] entries in the `tbl24` are iterated.
  If the extend bit (E) for a matching `tbl24` entry is not set, the entry is updated. If the extend bit (E) however is set, all entries of the referenced `tbl8` are updated.
- If d_r is greater than 24, the most significant 24 bits of the network IP are used, to index a `tbl24` entry.
  If the extend flag (E) is not set, it has to be set and the Ref field has to be set to point to a new `tbl8` instance. If in this case, the `tbl24` entry's valid flag (V) was set, the new `tbl8` instance has to be filled with the existing information from the `tbl24` entry.
  Now the `tbl8` fields, whose index is matching the 8 least significant bits of the IP address are updated.
- When updating an entry, the depth field (D) is compared with the depth d_r of the route to add. Only, if the depth of the new route is greater, the entry is changed:
  - The depth (D) is updated to the new depth.
  - The valid flag (V) is set.
  - The reference (Ref) is set to point to the selected `nextHop` table entry.

  This will ensure, that the algorithm always returns the route with the longest matching prefix.

The field G in each `tbl8` entry is used by the implementation for allocating and deallocating the `tbl8` instances and not relevant for understanding the algorithm working principle.
Assuming, that in the Internet only few networks smaller than /24 are announced, this algorithm will only require one memory access for determining the next hop information. Otherwise a maximum of two memory accesses are required, to traverse the two table stages.
The memory consumption M in bytes for this algorithm depends on the usage characteristics and can be expressed as follows:

$$\begin{aligned} M = \; & 2^{24} \cdot (1 + \lceil \max(\log_2(\#tbl8), \log_2(\#nextHops))/8 \rceil) \\ & + 2^8 \cdot \#tbl8 \cdot (1 + \lceil \log_2(\#nextHops)/8 \rceil) \\ & + \#nextHops \cdot nextHopEntrySize \end{aligned} \tag{5.1}$$

Thereby, $\#tbl8$ is the maximum number of `tbl8`s and $\#nextHops$ represents the maximum number of entries in the nextHop table. The $\#tbl8$ parameter set at compile time of DPDK (default is 256). The parameter $nextHopEntrySize$ may be set at runtime. How-

[3] Matching entries are all entries, for which the first d_r bits of their index are the same as the network IP address of the route.

ever, the current implementation does not allow the maximum number of next hops (*#nextHops*) to be changed and its value is hard coded to 256. As presented in [3], an analysis of common routers in the Internet suggests, that the mean number of distinct next hops in a routing table is 219.5 with a standard deviation of 202.9. Hence, a nextHop table with 256 entries is enough for smaller routers.

5.1.3 Route application

A separate module exists, for copying next hop information into packets. This can be used to copy the next hop MAC address from an `nextHop` table entry into the destination MAC address field of packets.
Route application is separated from the Longest Prefix Matching (LPM) module described in Section 5.1.2 for two reasons:

1. It allows the LPM module to be more flexible, as the entry structure in the `nextHop` table does not need to have a fixed format, predefining its contents. This is possible, as the interpretation of a `nextHop` entry is done by the route application module in a separate processing step. Hence, if the structure of the entries changes, only the route application module has to be changed.

2. It enables an increased performance gain for batch processing. This is achieved, as route lookup and route application is split into two processing steps. In this case, the LPM module mostly works on `tbl24` and `tbl8`, whereas this module accesses the `nextHop` table, which separates the memory regions for each processing step, optimizing cache efficiency.

5.1.4 TTL decrement

This module updates the TTL field in the IP header according to the requirements for Internet routers, specified in [4]. If the TTL value is greater than one, it is decremented by one. Otherwise the packet is determined to have exceeded its maximum Time to Live.

5.1.5 Packet distribution

Even though, packet processing can be done in few processing paths most of the time, eventually packets have to be distributed to different NIC ports and take separate paths from then on. As shown in Figure 5.2, this module implements a packet multiplexer. Additional information for each packet (i.e. nextHop table entry) is used as an input to a redirection table, which in the end determines the corresponding transmit queue of a NIC, the packet should be sent to.

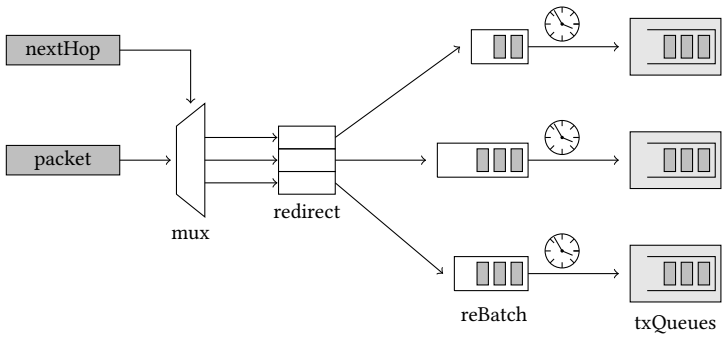


Figure 5.2: Packet distribution.

The module supports *rebatching*, which allows to accumulate packets into bigger batches, each destined for a distinct Tx queue. Thereby, the maximum batch-size can be specified individually for each Tx queue. This method is used, as with the bitmask approach, for all masked packets the descriptors would have to be sent to the NIC individually. When using *rebatching* however, costs for sending the packet descriptors to the NIC over multiple packets are amortized.
To avoid high latencies, or the starving of flows with little traffic, additionally a timeout can be set for each Tx queue separately. The accumulated packets are then flushed, as soon as the oldest packet in the batch is older than the set timeout. It hence allows the worst case latency of this module to be configured.

5.1.6 Packet filters

Three different filters can be used to perform packet classification, allowing the implementation of firewalls or pre-filtering for protocol processing like ARP:

Hardware L2 Filter—This module, shipped with MoonGen allows adding filter rules, which matching the Ethernet protocol field. Processing for this filter is offloaded to the NIC hardware. Matching packets are placed in one of the hardware Rx queues, which was specified at filter rule creation.

Software 5-tuple filter—This filter is a powerful software based packet classification module, based on the DPDK ACL library [9]. For each 5-tuple filter instance, rules can be added, matching packets according to the following properties:

- The protocol field in the IPv4 header.
- An IPv4 source address subnet.
- An IPv4 destination address subnet.

- A range of Layer 4 source port numbers.
- A range of Layer 4 destination port numbers.

A rule associates each matching packet with a set of categories, specified at rule creation. In addition, each rule also assigns a numeric value to matching packets. At filter initialization, bitmasks can be bound to individual categories, which will - after classification of a batch - identify the packets, corresponding to the bound category. This performs bitmask based batch scattering, as introduced in 4.2.4.

Hardware 5-tuple filter—Like the previously introduced software based 5-tuple filter, this filter works on the Layer 3 and 4 address and protocol fields. However, address ranges and multiple categories are not supported. One filter rule only matches one distinct tuple. For each added rule, a NIC Rx queue has to be specified, to which matching packets are forwarded. The filtering is then offloaded to the NIC hardware.

5.2 Fast Path

To achieve high performance, the implementation of the Fast Path unit is kept as small as possible, avoiding unnecessary computation. Thereby, the functionality is described by interconnecting the processing modules, introduced in Section 5.1, using the Lua language. The complete Fast Path source code consists of only about 100 lines of Lua code, including comments (see Appendix A). Figure 5.3 shows a control and information flow diagram of the Fast Path unit. Execution can be split into three parts:

Packet reception—First, one of the Rx ports, bound to the unit is selected in a round robin scheme (select Rx Port). After that, a batch of packets are requested from the corresponding NIC queue (Batch receive). This request may result in a number of received packets from zero to the maximum configured batch size. If packets were received, execution continues with packet processing, otherwise packet processing is skipped.

Packet processing—To allow bitmask based batch processing, first bitmasks have to be initialized. Three bitmasks (tx_mask, rx_mask and valid_mask) are required during processing. Thereby, the rx_mask is initialized to mask all received packets, whereas the other masks are initialized to zero.
After bitmask initialization, six processing steps are executed, interacting with the bitmasks, as shown in Figure 5.3. Thereby, the TTL decrement and route lookup step may redirect packets to the Slow Path unit for further processing, by enqueuing them to a dedicated queue (see Figure 4.5).

The last step is to drop all packets, for which no processing could be done. In the current implementation, these are all packets, which were sorted out by the Validate IPv4 step.

Periodic tasks—The Fast Path unit periodically performs two tasks. The first one is to handle the timers for the packet distribution processing module. (Handle tx timers) As described in Section 5.1.5, the distribution module performs re-batching and ensures a maximum latency by flushing accumulated packets if necessary.
The second task is to perform synchronization with the Slow Path unit (Handle commands). This is done by checking for command messages in the configuration queues (see Figure 4.5) and responding to them. In the current implementation, the only implemented command is to switch to a new routing table, which requires an acknowledgement message to be sent back to the Slow Path after a successful switch.

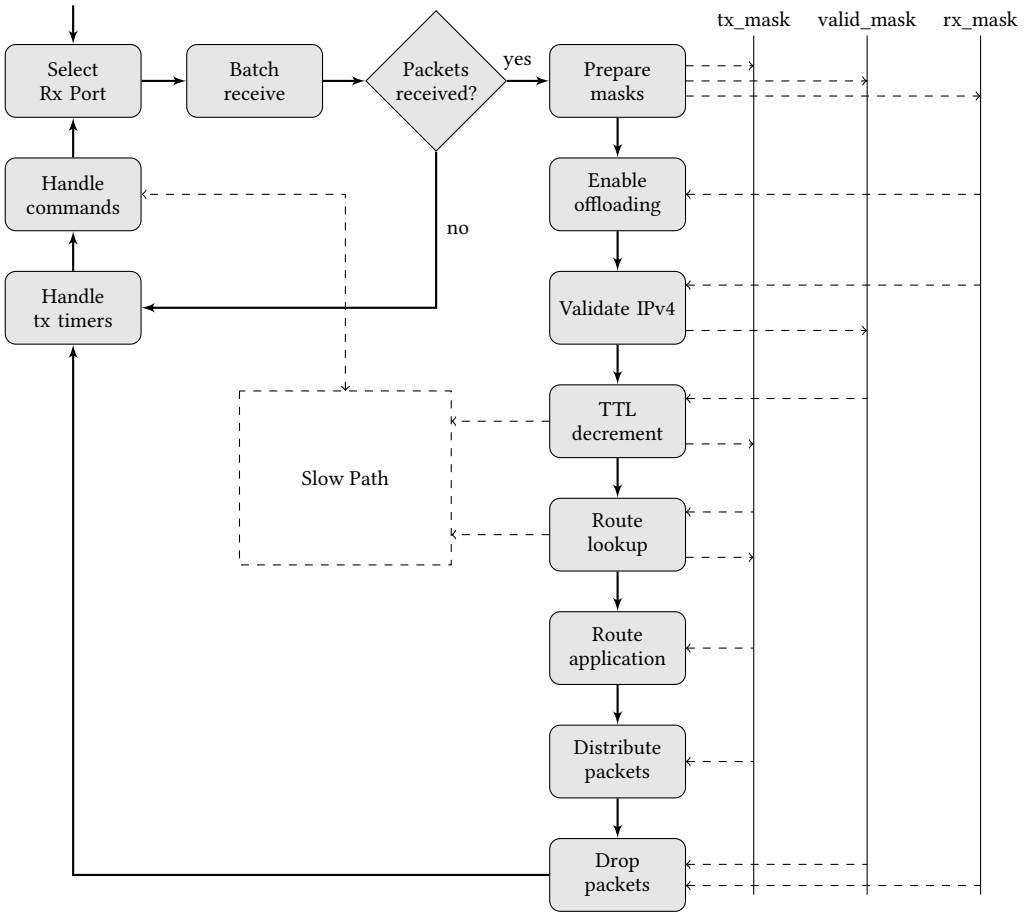


Figure 5.3: Fast Path control and information flow.

5.3 Slow Path

This Section not only covers the Slow Path for packet processing, but also all other tasks, which run at low priority during the router runtime, which are also referred to as Slow Path tasks. In the current implementation of the router, only one Slow Path instance is supported, which is bound to the same CPU core, the Linux Operating system is running on. However, the design can easily be extended to allow an arbitrary number of Slow Path packet processing units, by moving the synchronization and user interface tasks into a separate unit.

5.3.1 Routing table management

As already mentioned in Section 4.3, the routing table is *double buffered*, to allow lock free-operation. The Slow Path is thereby responsible for updating and synchronizing the tables. Three sources for retrieving lists of valid routing rules are currently available:

Static routes: A Lua table, containing fixed rules, which are defined in the router configuration file.

Dynamic routes: A Lua table, containing rules, which can be modified during runtime (currently only via the telnet user interface).

Linux routes: A method for extracting rules from the routing table of the underlying Linux operating system (see Section 5.3.2).

When a routing table synchronization is required, the following steps for performing a *double buffered* routing table update are executed:

1. A new LPM routing table (described in Section 5.1.2) is allocated.
2. The newly allocated routing table is filled with rules retrieved from the previously mentioned sources. Thereby next-hop IP addresses are resolved into MAC addresses, to allow faster forwarding.
3. A message, containing a reference to the new routing table is sent to all n_f Fast Path units, using the command queue to the respective unit.
4. The command queues for receiving messages from the Fast Path units are repeatedly polled, until every Fast Path unit confirmed the switch to the new routing table, by sending an acknowledgement message.
5. The old routing table, which was previously in use by the Fast Path units is freed.

5.3.2 Interaction with the Linux network stack

The router can be configured, to interact with the Linux network stack by exchanging packets, or even to perform L3 forwarding in behalf of the Linux kernel. To achieve this, the Kernel Network Interface (KNI) kernel module and corresponding user-space library shipped with DPDK is used. For each NIC port a corresponding virtual network interface with the same MAC address is created[4]. Those interfaces are managed by the Slow Path as follows:
All packets sent from the Linux network stack to one of these virtual network interfaces are forwarded to the corresponding real NIC ports.
Arriving IP Packets on a specific port, which are addressed to the IP address of this port, are respectively forwarded to the virtual interface. To allow the Linux network stack to fill its ARP table, additionally all arriving ARP messages are forwarded to the virtual interface, matching the port on which the packet arrived.
When updating the routing table, the Slow Path is also able to extract the routing rules, currently present in the Linux routing table and merge it into the LPM table, which will be passed to the Fast Path units.
This approach allows advanced routing protocol implementations like BIRD or Quagga (see Section 2.3), which where developed to run on top of Linux, to be used with the proposed router, handing over its increased forwarding performance.

5.3.3 Packet processing

The Slow Path receives and processes packets, from three sources:

NIC ports—Currently only ARP packets are redirected to the Slow Path by the NIC. Those packets are handled by the Slow Path, to maintain an ARP lookup table and to respond to ARP requests accordingly.

Fast Path units—All packets, redirected by the Fast Path could not be routed successfully and hence need more thorough processing.
Thereby, no information is transferred, indicating the cause, why the packet could not be processed. This was done to relieve the Fast Path units from spending processing time on complex packets and to reduce complexity in the Slow Path.
Instead, the Slow Path implements a fully capable routing algorithm, performing all necessary steps, which would also be done in the Fast Path packet by packet.
In case of routing problems however, ICMP messages are generated and sent according

[4]The current version of the DPDK KNI library does not support configuration of the interface. Hence, the interface is created and afterwards automatically configured by executing the external `ifconfig` tool, which is part of the `net-tools` collection, shipped with Linux.

to the RFC1812 [4] requirements.
If a packet is destined to the IP address of the port it was received on, it is forwarded to the Linux network stack via the corresponding virtual interface.
The Slow Path also supports efficient routing to local subnets, which is by default not possible in the Fast Path (i.e. routing of packets for which the next hop is the destination host): In case a packet is destined to a host in a local network, the Slow Path first performs an ARP lookup for the destination IP. If the lookup is successful, the packet is forwarded to this host. After that, the Slow Path adds a /32 routing rule for this host to the dynamic routes table and triggers a synchronization of the routing tables. Subsequently, packets to this host are processed by the Fast Path units.

Linux network stack—Packets, which are received on any virtual network interface, are directly sent out on the corresponding interface, as all packet processing and routing was already done by the Linux network stack.

5.3.4 User Interface

The Slow Path provides an interactive telnet user interface, which is implemented using the *luaSocket* library. Currently the commands listed in Table 5.1 are implemented.
The user interface can easily be extended, by adding commands and their respective handlers to the `cmds` Lua table.

Command	Alias	Arguments	Description
help			Displays all available commands.
addRoute	ar	networkIP networkPrefix nextHopPort [nextHopIP | nextHopMAC]	Adds a routing rule to the dynamic routing table.
updateRoutes	ur		Forces synchronization of the routing tables to all FastPath units.
showRoutes	sr		Shows all active routing rules.

Table 5.1: User Interface commands.

5.4 Router initialization

This section describes the steps, which are performed by the MoonGen master script, to initialize the router for a given configuration.

Configuration file—On startup, a configuration file, written in Lua is read. This configuration file is split into three logical categories:

- General router configuration: Configures batch and queue sizes, the user interface, the Linux kernel interface, as well as the CPU cores that should be used for Fast Path units.
- Port configuration: Configures, which NIC ports to use. For each port, a list of Fast Path units can optionally be specified, which will be used for processing arriving packets on this port. By default, all available Fast Path units are used.
- Static routing table: Persistent routing rules can be added in the configuration file. Either the next hop MAC address or a next hop IP address may be specified.

An example configuration file can be found in Appendix B.

Device initialization—After evaluating the configuration, the specified network ports are searched and initialized. For initialization, memory pools for packet buffers, as well as the descriptor rings have to be preallocated. The descriptor rings must have a size of more than the batch size, which is received or transmitted, to allow smooth operation. In the current implementation descriptor rings are initialized with a size of

$$N_{RxDesc} = 4 \cdot rxSize \tag{5.2}$$

$$N_{TxDesc} = 4 \cdot txSize \tag{5.3}$$

to give enough safety margin, in case of polling delays. For packet buffer preallocation, the maximum amount of packets, which may reside inside the router has to be considered:

$$N_{inRouter} = rxSize + \#Distributors \cdot txSize + n_f \cdot SlowQueueSize + 1 \tag{5.4}$$

Enough packet buffers have to be reserved, to back up the descriptor rings, as well as the packets being processed. Therefore in total

$$N_{Bufs} = 2 \cdot N_{RxDesc} + N_{inRouter} + \#Ports \cdot 2 \cdot N_{TxDesc} \tag{5.5}$$

buffers are allocated for each port - again with a factor of two as a safety margin[5] - during router initialization.

Asynchronous units—After device initialization, all Fast Path units are started on their assigned CPU cores. Thereby, the initial routing table is passed as an argument.
The CPU core used for router initialization will be used for the Slow Path. If required, the virtual interfaces are registered and configured. Consequently the Slow Path main program loop is entered and the execution of all Slow Path tasks begins.

[5]Due to replenishment delays and caching effects in DPDK, more buffers typically need to be provided, than are really required. The given safety margins are empirically determined.

Chapter 6

Performance evaluation

To prove, that the proposed router architecture is able to compete with commonly used software router architectures, some basic performance measurements were done as part of this thesis. In this chapter, the used test environment and eventually the performed tests are detailed.

6.1 Test environment

In this section, the test environment and basic test strategy is explained. This environment was used for testing, optimizing, as well as performance measurements:

Hardware—Performance measurements were conducted, using servers, equipped with Intel Xeon CPUs and 10 Gbit/s Intel X540 or 82599 family NICs. Table 6.1 shows the specific hardware configuration of each used server. As shown in the table, the servers were grouped in pairs with similar hardware. Each server pair was directly interconnected via one or two Ethernet cables. For controlling the servers, a separate management network was used, running on separate NICs, which were not used for testing.

Name	CPU	Cores	Freq.	Mainboard	NIC
cesis	Xeon E5-2640 v2	8	2.0 GHz	X9SRH-7F/7TF	2x X540-T2
nida	2x Xeon E5-2640 v2	2x8			
palanga tallinn	Xeon E3-1230	4	3.2 GHz	X9SCL/X9SCM	X540-T2
klaipeda	Xeon E3-1230	4	3.2 GHz	X9SCL/X9SCM	2x 82599ES
narva	Xeon E3-1230 v2		3.3 GHz		2x 82599EB

Table 6.1: Servers used for testing.

Software—For generating test traffic and performing network measurements, MoonGen was used, which was capable of handling traffic at a rate of 10 Gbit/s per NIC port in all performed tests.
For reading CPU performance registers and profiling, the *perf*[1] tool was used. For measuring CPU cache behavior, *pmu-tools*[2], an extension for *perf* was used, which allowed reading cache events of the Xeon E31230 v2 CPU.
Profiling and optimizing Lua code was done, using the profiler, shipped with LuaJIT.

Strategy—All experiments were performed, using one server of a pair as a load generator. On the other Server, the router was running, configured with a routing table, to route all packets back to the load generating Server. In case of two active NIC ports, the router was configured to route packets, destined to one /1 subnet back over the first port and packets to the other /1 network over the second port. By randomizing the packet destination address when generating test packets, the load could be fairly distributed to the two ports.

An *experiment* E is defined by the tuple

$$E := (C, P, M, P_s, n). \tag{6.1}$$

Thereby, C is the persistent test configuration, which was not changed during the experiment, P is a set containing the type of all *parameters*, which are modified during the experiment (for example P = {Packet size, CPU frequency}) and M represents the set containing the type of all *measurements*, which are performed (for example M = {Throughput}). During the experiment, different sets of values for all elements in P are chosen. One chosen set of values is called a *parameter configuration* P_c. P_s is then called the *parameter set* which contains all *parameter configurations* P_c, used in the experiment:

$$P_s := \{P_{c1}, P_{c2}, ..., P_{c|P_s|}\} \tag{6.2}$$

for $|P_s|$ parameter configurations.
The Experiment is done by performing n *test runs* for all $P_c \in P_s$. After each test run, the router and traffic generator is restarted. This leads to

$$N := n \cdot |P_s| \tag{6.3}$$

test runs, which are performed in each experiment.

A test run for a specific parameter configuration is performed in four steps, which are executed in the following order:

[1] `https://perf.wiki.kernel.org/index.php/Main_Page`
[2] `https://github.com/andikleen/pmu-tools`

1. The router is started with the configuration, matching P_c.
2. The traffic generator is started to generate traffic, matching the parameter configuration P_c.
3. The measurement result for this parameter configuration is determined.
4. The router, as well as the traffic generator is stopped.

In the following sections different experiments are presented. In each section, first the persistent test configuration C is explained, followed by the parameter types P. Additionally the measured values in M and additional experiment properties are described. As long as not specified otherwise the following persistent configuration and experiment properties were used:

- The packet size of the test traffic was 64 bytes.
- The Rx batch size was 128.
- The Tx batch size was 128.
- Four test runs were performed for all parameter configurations ($n = 4$).
- If no used measurement type is specified, throughput measurements were performed on the server, receiving the routed packets. This was done by counting the incoming packets for one second.

6.2 Single core forwarding throughput

Experiment setup—For measuring the throughput on a single CPU core, the servers tallinn and palanga were used. The router was set up at tallinn and was configured to use one NIC port and one FastPath unit, routing all packets back over this port. Palanga acted as a traffic generator and throughput counter.
The experiment was performed with two parameters in P:

- The CPU frequency.
  To change the CPU frequency, the power management library, provided by DPDK was used at tallin.
- The traffic pattern.
  To test the routing algorithm, the destination IP address of the sent packets was partly randomized. The following parameter configurations were used for the traffic pattern: One configuration with a fixed, non random, IP address for all sent packets, as well as three additional configurations with 20, 21 or 24 most significant bits of the IP address being randomly chosen for each sent packet.

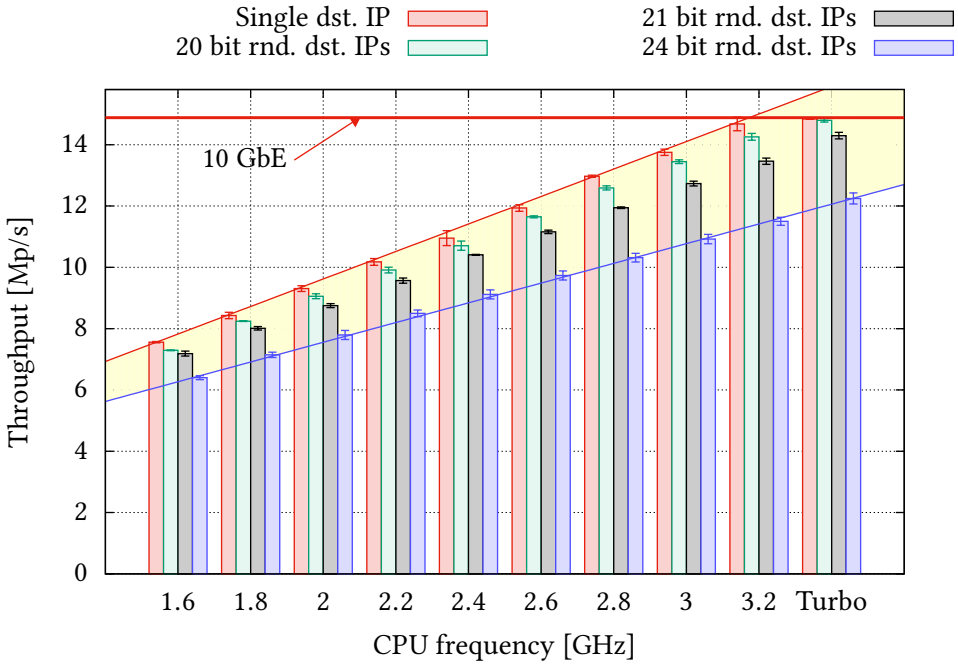


Figure 6.1: Scaling with the CPU frequency.

Test results—Figure 6.1 shows the measurement results for the different traffic patterns and CPU frequencies in a graph. For the *turbo* step, marked with *Turbo*, the *Turbo* mode of the Xeon CPU was activated. In this mode, the CPU's frequency is increased by 0.4 to 0.5 GHz, depending on the CPU [23].
A linear fit over the measurement results shows, that the throughput of the router scales linearly with the CPU frequency. The last frequency step is excluded in the linear fit, as line rate was reached at 14.9 Mp/s, which limited throughput by link saturation.
The second effect, which becomes visible is, that the randomness in the destination IP address influences the router performance. The reason for this is the routing table implementation (see Section 5.1.2). The first LPM table (`tbl24`) is indexed by the 24 most significant bits of the destination IP address. The more randomly this routing table is accessed, the less efficient are the CPU caches. The worst case being total random access to the `tbl24`. Hence, the throughput of the proposed router in a real network will be in the yellow marked area of the graph. The impact of the routing table on the router performance will be analyzed in more detail in Section 6.5.

The test showed, that with few active traffic flows (meaning few destination IPs), a single CPU core is almost able to route minimum sized IP packets at line rate of a 10 Gbit/s NIC port, reaching 14.6 Mp/s. Table 6.2 shows a comparison with the other available software router solutions, which were presented in Chapter 2. The table consists of published measurements, which were adapted to be comparable with the measurements, performed with the proposed router. Thereby, it was assumed, that throughput scales

Implementation	Source	CPU freq.	Mp/s	Mp/s scaled to 3.2 GHz	Relative
Proposed router	—	3.2 GHz	14.6	14.6	100%
Batching Click	[25]	8x2.66 GHz	41.7	6.27	43%
FreeBSD 11-routing	[7]	8x2.0 GHz	9.54	1.91	13%
Route Bricks	[8]	8x2.8 GHz	12	1.71	12%
Linux 3.7	[14]	3.3 GHz	1.67	1.62	11%
Click	[27]	0.7 GHz	0.35	1.60	11%
FreeBSD 10.2	[31]	4x2.13 GHz	1.76	0.67	4.6%
Linux 2.2.14	[27]	0.7 GHz	0.095	0.43	2.9%

Table 6.2: Maximum single core forwarding performance comparison.

linearly with the number of used cores, as well as with the CPU frequency. The comparison shows, that even the fastest competing router implementation, Batching Click, only gives 43% of the performance, the proposed router is able to reach. The big performance difference between Linux 2.2.14 is caused by the introduction of NAPI. The FreeBSD benchmark results vary heavily with the used operating system configuration. In the table, the best seen performance is used. The measurement labeled with FreeBSD 11-routing was performed with a modified freeBSD version, which was optimized for routing.

6.3 Performance at different packet sizes

Experiment setup—For this test, the server cesis with its CPU clocked to 1.2 GHz was used to run the router in a $(2, 2, 1)$ configuration, using one Fast Path unit. Nida was utilized, to run the traffic generator and to perform the measurements. Two NICs per server were used in the way, which was described in Section 6.1. Arriving packets on both NICs are routed back to nida, arriving at one of the NICs, dependent of the IP destination address.
The experiment was performed with two parameters in P:

- The destination address pattern:
  Two different destination address patterns were used in the test:
  - Only the most significant bit of the address are randomly chosen. This ensures fair distribution of traffic to both NICs, while not causing excessive load on the LPM algorithm. A performance of the routing table lookup similar to non random lookups is expected.
  - All 32 bits of the IP address are randomly chosen. This gives full load to the LPM algorithm (only the `tbl24` was used).

- The packet size:
  The test was performed for different packet sizes, ranging from 64 bytes to 544 byte per packet.

For each configuration $n = 2$ test runs were performed.

Test results—Figure 6.2 shows the measurement results. The graph contains the same data, plotted in two different units. The left vertical axis (valid for the bar graphs) shows the throughput in Gbit/s while the right axis (valid for the line graphs) uses Mp/s units. The graphs labeled with *Single* correspond to the first address pattern, whereas the graphs labeled with *Random* correspond to the second address pattern.
It is clearly visible, that while the throughput is linearly rising with increasing packet size, the packet rate stays constant, until the maximum possible line rate of 20 Gbit/s is reached. When still increasing the packet size, after reaching the line rate, less packets are required, to keep the bitrate constant.
This result proves that the assumption made in Section 2.1 holds for the proposed router, as the packet size has little to no influence on the processing performance of the router.
The graph in Figure 6.2 additionally includes published forwarding (no routing was done) measurement results from the Route Bricks project [8], which were linearly scaled to be comparable. While yielding significantly less throughput than the proposed router, the packet processing rate of the Route Bricks router also does not stay constant, even though the Route Bricks project did not hit the line rate limit.

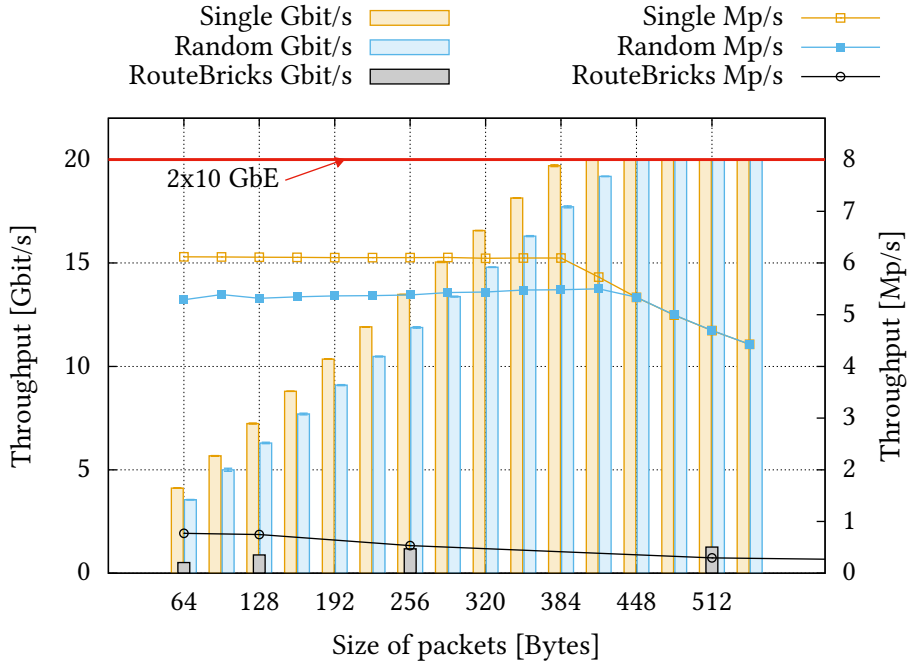


Figure 6.2: Effect of the packet size on throughput (1.2GHz).

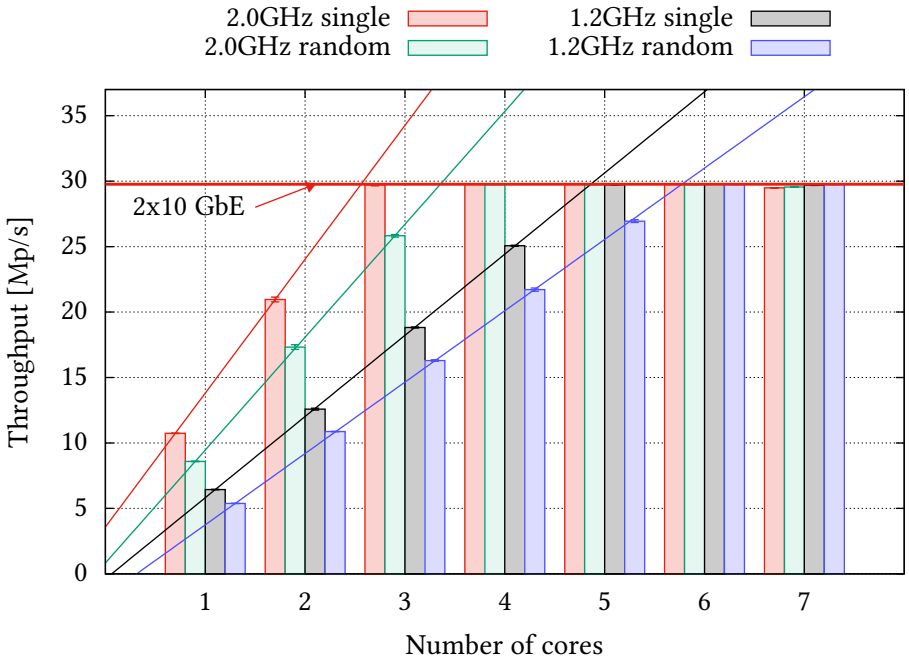


Figure 6.3: Scaling with the number of CPU cores.

6.4 Multi-core routing throughput

Test Experiment setup—For this experiment, again cesis was used to run the router, utilizing two NICs and nida was used for traffic generation, the packet size was fixed at 64 byte.
In the experiment, three parameters in P were used:

- The destination address pattern. Therefore, the same two address patterns as described in Section 6.3 were used.
- The CPU frequency. The values 2.0 GHz and 1.2 GHz were used.
- The number of used CPU cores, running Fast Path units.

Test results—The measurement results are shown in Figure 6.3. The graphs labeled with *single* correspond to test traffic with only the most significant IP address bit being randomly chosen, whereas the graphs labeled with *random* are generated using completely random IP addresses.
As shown in the graph, linear fits could be perfectly done for all traffic patterns and CPU frequencies. Thereby, only results smaller than the link saturation packet rate of 29.8 Mp/s were used for the fit. This shows, that the proposed router achieves linear scaling with the number of cores. Two 10 Gbit/s Ethernet links can be saturated at 2.0 GHz using a minimum of three cores and in the worst case using four cores. At

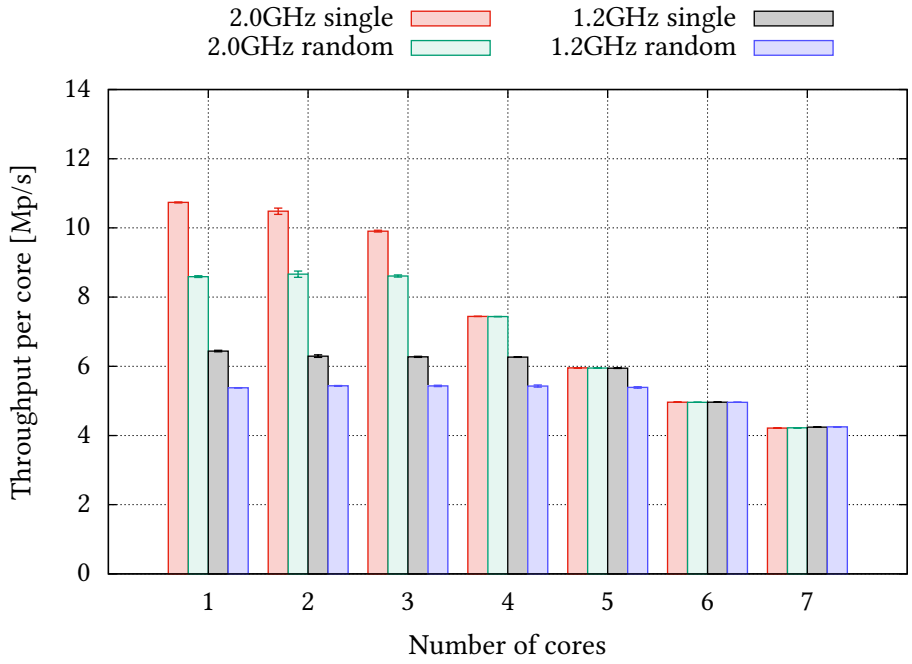


Figure 6.4: Per core throughput for different numbers of CPU cores.

1.2 GHz five or six cores are needed respectively. In Figure 6.4 the same data is displayed as in Figure 6.3, but this time, the throughput per CPU core is plotted. It can be seen, that the throughput is roughly constant for increasing numbers of CPU cores, until line rate is reached. This shows that the linear fit has a low offset and allows to deduce, that little interaction between the CPU cores take place.

6.5 Routing table performance

For analyzing the routing table performance, two experiments were performed on different server pairs. The experiments are described in the following.

Experiment setup 1—The servers tallinn and palanga were used. The router was set up at tallinn and was configured to use one NIC port, routing all packets back over this port. Palanga acted as a traffic generator and throughput counter.
Three parameters in P were used for the experiment:

- The CPU frequency.
- The number of used CPU cores, running Fast Path units.
- The number of used /24 subnets for the destination IP address. This was done, by changing the number of randomly chosen bits for the 24 most significant bits.

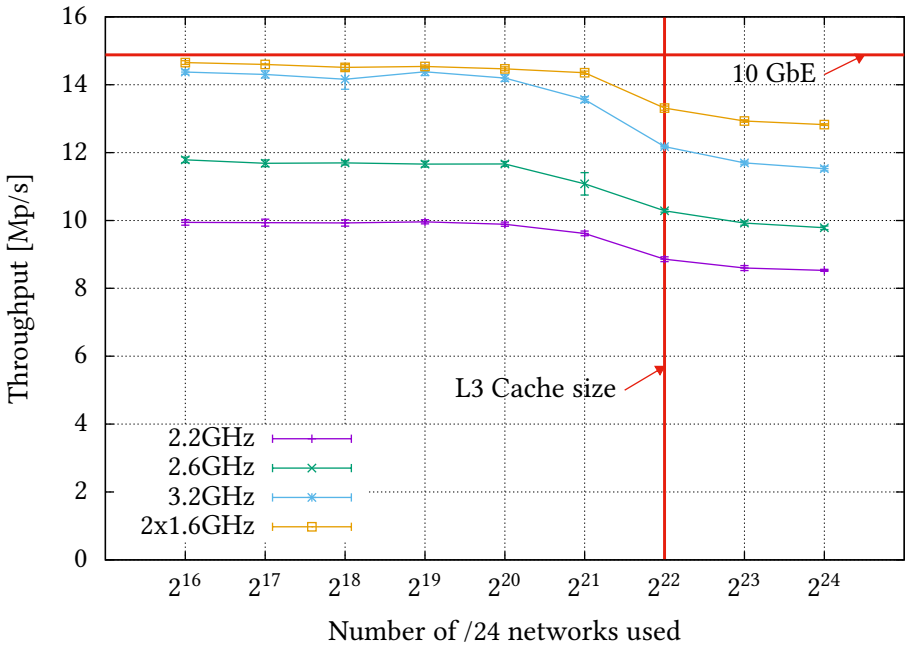


Figure 6.5: Effects of the routing table on throughput.

Test results 1—Figure 6.5 shows the results of the first benchmark for different CPU frequencies and number of used cores. The vertical red line indicates the number of `tbl24` entries (i.e. the number of /24 subnets), which would completely fill the 8 Mbyte layer 3 cache.
Several observations can be made:

- The throughput stays relatively constant, as long as less than 2^{21} subnets are used. However, as soon as more `tbl24` entries are randomly accessed, the performance is dropping. This effect is caused by the Layer 3 cache, which is at some point not able to hold all frequently accessed `tbl24` entries.
- An increased CPU frequency influences the throughput significantly. However, the previously mentioned effect of dropping performance with increasing `tbl24` usage, is worsening.
- The throughput of two cores, running at 1.6 GHz, is in all measurements higher than the performance a single CPU core is able to achieve, running at 3.2 GHz. This can be explained by the separate L1 and L2 caches for each core.
- Two CPU cores also reduce the performance drop, with increasing number of used /24 subnets. For totally random `tbl24` access, the dual core configuration reaches 12.8 Mp/s which is a performance boost of 11%, compared to the single core configuration, running at twice the frequency which only achieved 11.5 MP/s. This can again be explained as an effect of separate caches.

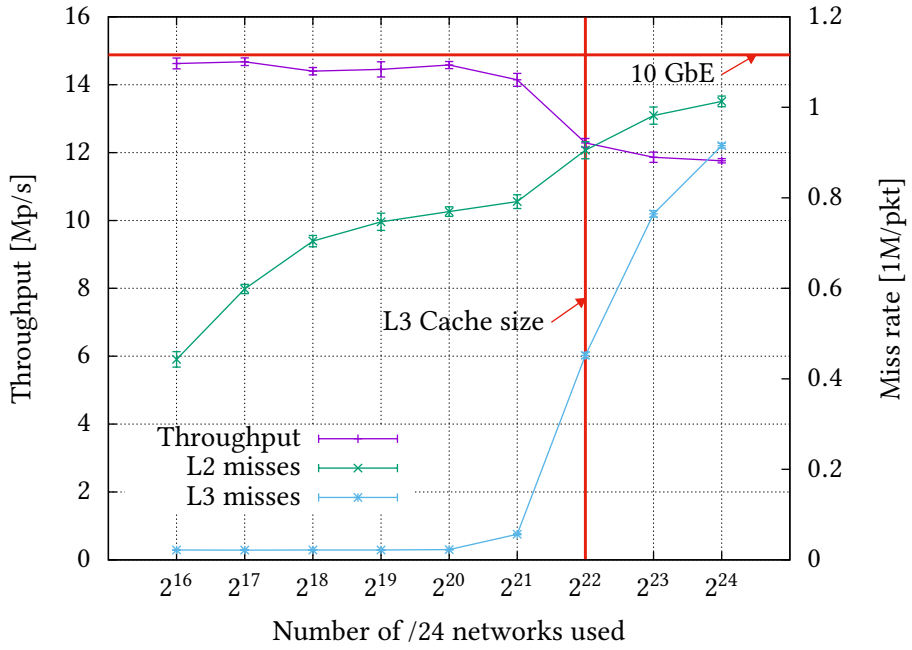


Figure 6.6: Cache effects caused by the routing table (2.9GHz).

The cache effects are analyzed in more detail in the following paragraph, using results from the second benchmark.

Experiment setup 2—In the second experiment, the router was executed on narva, constantly running at 2.9 GHz. Klaipeda was used as a traffic generator.
For this experiment, only the number of randomly chosen /24 subnets was used as a parameter.
This time three types of measurements in M were performed for each parameter configuration in P_s:

- The throughput was measured by counting the incoming packets on klaipeda for one second.
- The L2 cache miss rate was measured, using the *perf*-based tool *pmu-tools*. This was done by observing the corresponding cache related CPU performance registers of narva for one second, while the router was actively routing.
- The L3 cache-miss rate was measured similarly to the L1 cache miss rate.

Test results 2—The measurement results of this experiment are shown in Figure 6.6. The left vertical axis is valid for the graph showing the throughput, while the right vertical axis is valid for the two graphs showing the L2 and L3 cache miss rate. For the

throughput, again the same effect as in the first experiment is visible. When comparing the throughput with the cache miss rates, it is possible to conclude, that the performance drop with increasing /24 subnet usage is in fact caused by cache effects: As more `tbl24` entries are accessed, first not all accessed entries fit in the L2 cache (a maximum of 2^{17} entries). As a result the L2 cache miss rate is increasing. With even more randomly accessed entries, at some point also the L3 cache can not hold all used entries. Hence, eventually the L3 miss rate is increasing. This heavily influences performance, as the probability increases, that slow main memory accesses have to be done for `tbl24` entries. As seen in the graph, the L2 miss rate also increases, the more /24 subnets are used. This effect will be reduced, in a dual core configuration, because each core has its own L1 and L2 cache, effectively increasing cache size.

6.6 Effect of batching

For analyzing the effect of batching on the throughput of the router, two experiments were performed:

Experiment setup 1—The server tallinn was used for running the router with a single Fast Path unit handling NIC port, routing all received packets back over this port. The CPU frequency of tallinn was set to 3.0 GHz. Palanga was used as a traffic generator, generating packets with completely randomized IP destination addresses.
For the experiment two parameters were used in P:

- The number of packets in a Rx batch (input and computation batching).
- The number of packets in a batch, which is filled with packets in the distribution processing step by performing rebatching (TxB).

For each configuration $n = 2$ test runs were performed.

Test results 1—The graph in Figure 6.7 shows the measurement results as a function of the Rx batch size, whereas in Figure 6.8, the results are shown as a function of the Tx batch size. Measurements for the Batching Click project (see Section 2.5), taken from [25] are also included in Figure 6.7[3]. However, the used traffic pattern for these measurements was not published.

[3]The published measurement results were linearly scaled for number of CPU cores and CPU frequency, to be comparable.

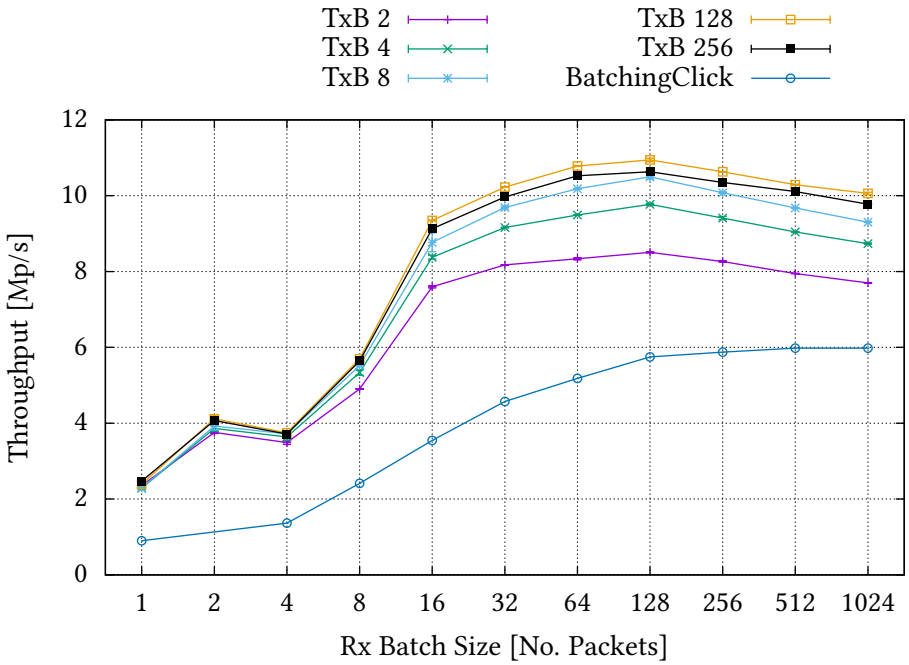


Figure 6.7: Effect of Rx batching on throughput (3.0GHz).

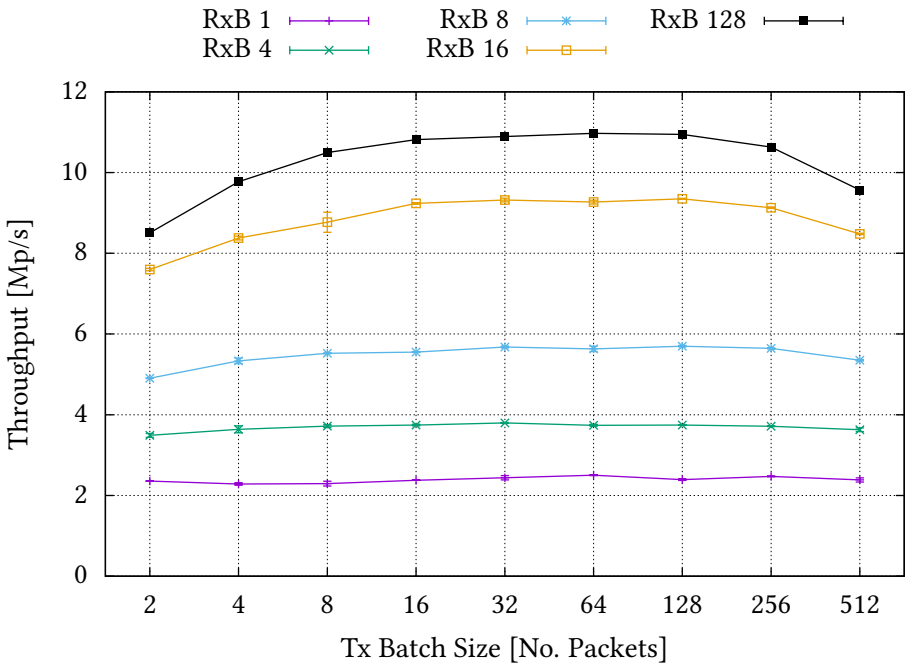


Figure 6.8: Effect of Tx batching on throughput (3.0GHz).

Four observations can be made:

- Looking at Figure 6.7, it is visible that the Rx batch size has a huge impact on performance. Batch sizes bigger than 16 give an increase in throughput of at least 100%, compared to a batch size of two. With a Rx batch size of 128 an optimum is reached and performance slowly decreases with bigger batch sizes.
- Compared to the measurement results from the batch enabled Click project, the proposed router achieves a better performance for all batch sizes.
- While not having an impact as huge as Rx batching, Figure 6.8 shows, that Tx batching is able to increase performance. An increase in performance of up to 28% is reached, when using Tx batching. The best performance is achieved with a batch size between 32 and 128.
- Tx batching gives no significant performance improvement for Rx batches smaller than 16.

Experiment setup 2—To analyze the effect of Rx batching on the CPU utilization, a second experiment was performed. This time narva was used to run the router at 2.9 GHz using one Fast Path core. Klaipeda was used as the traffic generator. The Tx batch size was set to 128 for this experiment. The only Parameter in P was the Rx batch size. This time multiple measurements in M were performed for each parameter configuration in P_s:

- The throughput was measured by counting the incoming packets on klaipeda for one second.
- The L1, L2 and L3 cache miss rate, as well as the branch miss rate were measured, using the *perf*-based tool *pmu-tools*. This was done by observing the corresponding CPU performance registers of narva for one second, while the router was actively routing.

Test results 2—Figure 6.9 shows the measurement results. The left vertical axis is valid for the graph, labeled with throughput. The right vertical axis is valid for all other graphs, showing miss rates. Three observations can be made:

- As expected (see Section 2.5), batching influences branch prediction of the CPU. With increasing batch size, the branch miss rate is reduced, reaching a very low level for batch sizes greater than 8. At the same time the throughput increases strongly.
- For very big batch sizes of over 128, the miss rates of the L1 data cache, and later also the L2 data cache rises, which again reduces the throughput. This effect occurs, because with an increasing batch size, the amount of data which is

worked on is also increased. If the batch size is too big, the data, required in one processing step will not fit into the caches.

- The L3 cache miss rate stays constant for all tested batch sizes. Hence, bigger batch sizes, do not lead to more main memory accesses, than single packet processing. All memory which would be read from the CPU caches for packet wise processing can also be read from the caches for batch processing.

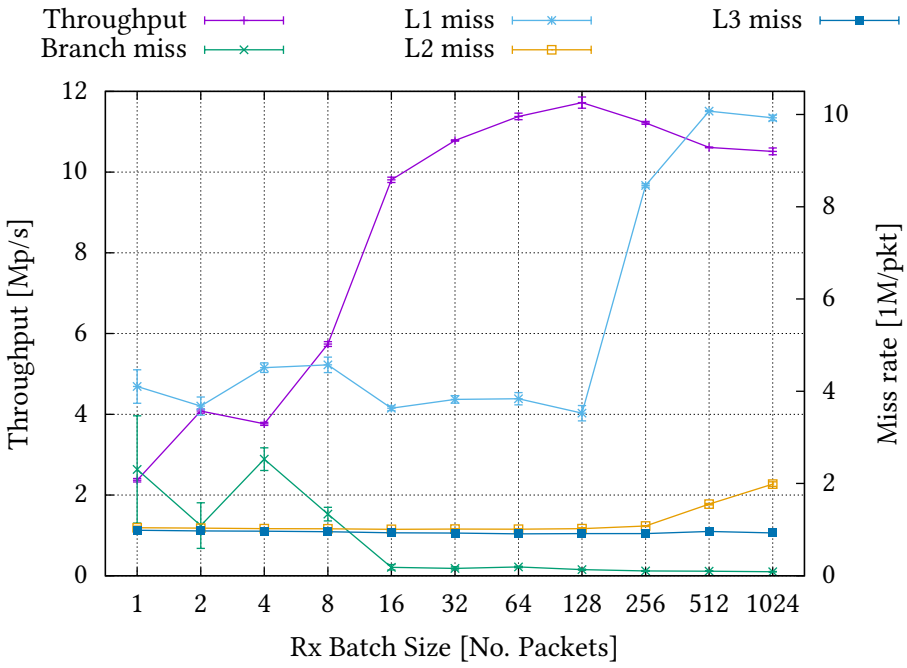


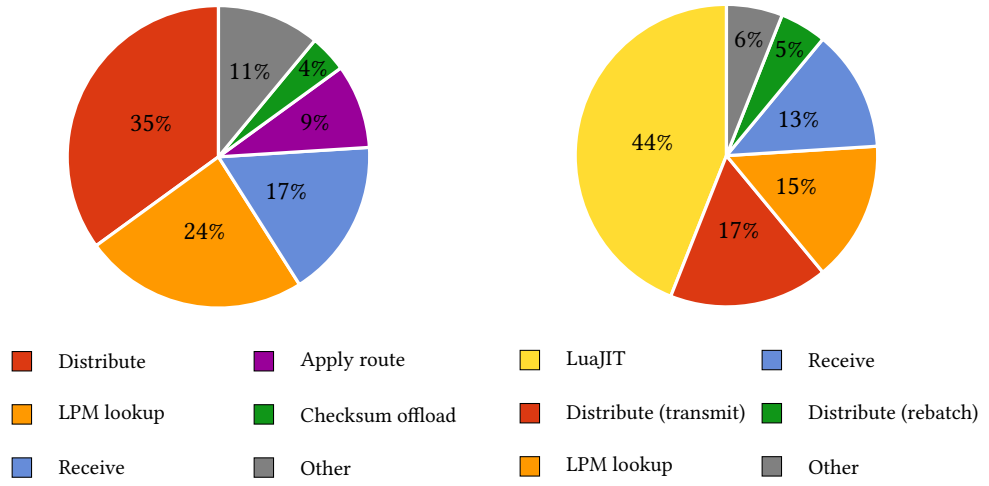
Figure 6.9: Cache effects caused by batching (2.9GHz).

6.7 Code profiling

To find out, how much time is spent in each function of the router, the following experiment was performed:

Experiment setup—The servers tallinn and palanga were used. The router was set up at tallinn, running at 2.6 GHz and was configured to use one NIC port, routing all packets back over this port. Palanga acted as a traffic generator, sending packets with completely randomized destination IP address. No variable parameters in P were used for this experiment. The experiment was performed two times, each time doing one of the following measurements at tallinn, while the router was under full load:

- Use the profiler, provided by LuaJIT, to estimate the time spent at different Lua code positions. 10k samples were recorded with a rate of 100 Hz.
- Use the *perf* tool, to estimate the time spent in different C functions. 1M samples were recorded wit a rate of 4 kHz.



(a) Lua functions (including calls to C). (b) Only C functions.

Figure 6.10: Time spent in Fast Path functions.

Test results—Figure 6.10a shows the estimated time, spent in each Lua function of the Fast Path. As expected, both input and output tasks (Receive and Distribute) consume a considerable amount of computation time. The LPM routing table lookup and route application are with 24% and 9% the most computation intensive processing steps. Interesting is, that configuring the packet metadata to enable checksum offloading requires as much as 4% of processing time, which is subject for improvement.
All other processing modules in the Fast Path require less than 4% processing time per module and are accumulated in the slice labeled *other*.

In Figure 6.10b the result of the *perf* profiler is shown. As the implementation for the distribution step is split into two parts, one for rebatching and one for actually sending the packets, it appears twice in the graph. This shows, that the computation cost for rebatching is with 5% significantly less than the actual transmission cost of the batch. All processing of Lua code is done by LuaJIT and hence contained in the corresponding slice. This indicates that 44% of the computation time is spent executing Lua code, which leaves room for improvement, to even further increase performance.

Chapter 7

Conclusion

In this thesis, the adaptability of flexible software routers to modern hardware architectures and future network standards, exceeding data rates of 10 GBit/s, was investigated. Based on this, a prototype was developed and evaluated. In the following, the findings are summarized and possible future work is elaborated.

Architecture—To achieve high throughput it proved as essential to utilize the possibilities of parallel computing architectures. Packet based communication is well suited for parallelization, as packets can be processed independently, under the premise, that the packet order for distinct flows is not disturbed. Thereby, explicit synchronization via locks have to be avoided. For high efficiency it is of advantage not to split the processing of a single packet to multiple CPU cores, but to distribute the packets to cores, from which each is performing all required processing [8].
To avoid excessive load caused from processing relatively few packets which require more processing, than most of the packets, prioritization can be used by splitting the router into asynchronous Fast Path and a Slow Path parts. Lower priority complex packets can be moved to the Slow Path via queues for further processing, while the Fast Path is able to continue forwarding packets at a high rate.
Besides parallelization and prioritization, batch processing is a technique, which gives huge performance increase by amortizing costs and optimizing CPU utilization. As shown in Section 6.6, a performance gain of more than 100% is possible with batching.
One of the main motivations for software routers is increased flexibility compared to hardware based solutions. The proposed architecture therefore provides for splitting the processing task into small steps, each realized by a processing module, with a clearly defined interface. The interconnection of these modules can be implemented using a scripting language, which also allows the implementation of new modules on demand.

Prototype—The prototype, presented in this thesis, is based on MoonGen, a flexible packet generator, using Lua script for describing packet generation logic. Thereby, DPDK was used to provide low level device drivers, which were capable of sending and receiving packets in batches with minimum overhead. MoonGen was extended with efficient modules for packet processing and routing, each providing a Lua programming interface. The presented router is then composed of Lua scripts describing the module interactions, as well as additional functionality for the Slow Path and Fast Path.

Evaluation—The performed performance evaluation gave an insight into the behavior of the router in different situations and allowed a comparison with existing approaches for software routers.
The prototype achieves linear scaling with both CPU frequency, as well as number of used CPU cores. Thereby, a single CPU core is able to route minimum sized IP packets with up to 14.6 Mp/s, which is 98% of the maximum rate in a 10 Gbit/s network. Serving multiple 10 Gbit/s Ethernet ports is therefore no problem, when multi-core CPUs are used.
The performed measurements also showed, that using multiple CPU cores at lower clock frequency, yields better performance than using a single CPU core with equivalently higher frequency presumably due to positive cache effects.
In all tests, the proposed router performed superior to the known existing software router implementations. The best published implementation only manages 43% of the throughput, the proposed router is able to achieve.

Future work—The results of this thesis represent a foundation for a new generation of software routers. However, the performed benchmark tests are merely basic throughput tests. For a better understanding and to further optimize the architecture, more thorough testing is required.
The limits of scaling are yet to be explored, as only tests were performed with up to seven Fast Path cores and two active NICs.
Also performance tests with heterogeneous traffic patterns, which challenge interactions with the Slow Path unit are to be performed and analyzed.
As the profiling test, presented in Section 6.7 showed, still much processing time is spent processing Lua code, even though most of the processing is currently implemented in C, with Lua only being used for connecting the modules and minor processing. Hence, more testing and profiling has to be performed, to accurately analyze the performance impact of Lua packet processing methods compared to C implementations.

Besides further testing, the proposed prototype can be extended, with the goal to develop a sophisticated and fully fledged software router.
In the following some interesting topics, which are not yet fully supported in the architecture or the prototype implementation, are listed:

- Methods for supporting Non Uniform Memory Architectures (NUMA) still have to be revised in the current architecture.
- The prototype can be extended to support multiple Slow Path instances on demand.
- The Slow Path implementation can be extended to more closely implement the RFC1812 standard.
- Advanced routing features, like support for multiple next hops or multicast routing have to be mapped to the proposed architecture.
- Currently only IPv4 is supported by the prototype. IPv6 can be implemented by adapting the LPM algorithm to support multiple trie stages.

Bibliography

[1] *ALS '01: Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, Berkeley, CA, USA, 2001.

[2] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy. Can Software Routers Scale? In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 21–26, New York, NY, USA, 2008.

[3] H. Asai and Y. Ohara. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 57–70. ACM, 2015.

[4] F. Baker. Requirements for IP Version 4 Routers. RFC 1812, RFC Editor, June 1995.

[5] B. Braden, D. D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309, RFC Editor, April 1998.

[6] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *USENIX Annual Technical Conference, General Track*, pages 333–346, 2001.

[7] O. Cochard-Labbé. fbsd11-routing.r287531. `https://github.com/ocochard/netbenches/tree/master/Xeon_E5-2650-8Cores-Chelsio_T540-CR/fastforwarding-pf-ipfw/results/fbsd11-routing.r287531`, 2015. Accessed: 2015-10-12.

[8] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28. ACM, 2009.

[9] DPDK. ACL Library. `http://dpdk.org/doc/guides/prog_guide/packet_classif_access_ctrl.html`, 2015. Accessed: 2015-09-18.

[38] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[39] W. Sun and R. Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 25–36. IEEE Press, 2013.

[40] V. Tanyingyong, M. Hidell, and P. Sjödin. Improving performance in a combined router/server. In *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*, pages 52–58. IEEE, 2012.