Performance Exploration of Software-based Packet Processing Systems

Daniel Raumer, Florian Wohlfart, Dominik Scholz, Paul Emmerich, and Georg Carle

Technische Universität München, Department of Computer Science, Network Architectures and Services
{raumer|wohlfart|scholzd|emmericp|carle}@net.in.tum.de

***Abstract*—Software systems that are placed on commodity hardware are integral components of today's networks. They gain momentum as an attractive alternative to dedicated hardware routers, switches, and firewalls. Their big advantages are the high customizability of the software and increased cost-efficiency of the necessary hardware. To improve the performance, features intended to speed up the processing, are steadily added. For assessing influences on the performance, the software and its interaction with available hardware features have to be tested and modelled. Therefore, we evaluate the performance of the Linux Network Stack in different use cases and develop a performance prediction model. Using both black- and white-box measurements the internal behaviour of the Linux router is analysed when approaching data rates up to 10 Gbit/s and the impact of occurring performance limiting factors is studied. External measurements are restricted to the packet rate, while white-box measurements are performed using the profiling tool `perf`. From the results of these measurements, the CPU cycles required to process each packet and their distribution to the tasks executed by the involved driver and kernel functions is analysed.**

***Index Terms*—measurement, software-based networked systems, performance correlation, performance model**

I. Introduction

Through steady development the performance of commodity hardware has been continuously increased in recent years. Especially multi-core architectures have therefore become more interesting to fulfil network tasks by using software packet processing systems instead of dedicated hardware. In particular, software routers using UNIX-based operating systems (OS) are attractive, as a fully functional protocol stack that handles the processing, in this case forwarding on layer 3 of the ISO/OSI model, is implemented. The advantages of high flexibility and lowered costs are contrasted with possible higher performance and reduced energy consumption achieved with specialized hardware [1], [2]. Furthermore, for software routers, features can be rapidly deployed or modified, whereas dedicated hardware would consume extensive development cycles before being able to be upgraded.

With the continuous growth of the internet, the demands on the used techniques increase, too. While 1 GbE is common in home networks, 10 GbE and 40 GbE is common standard in carrier grade networks. Even 100 GbE is already standardised [3]. While hardware routers are able to accomplish these line rates, software routers reach their limit [1]. However, software routers are in general capable of reaching the same performance that hardware routers deliver today [1], [4]. Problems arise especially for small packet sizes: the full line rate cannot be reached as the maximum number of packets per second that can be processed is limited by the CPU. Other factors are further limiting the performance of a software router [1], [4], [5].

While new techniques improve the performance, the interaction with other mechanisms has to be understood and analysed to prevent unwanted behaviour or possible side effects. A common approach is black-box testing: the software router is tested in a certain scenario, using varying input traffic, and, for instance, the resulting packet rate is used to compare the results. Although this shows how many cycles per packet are needed to process a packet, it is not further analysed where exactly, in relation to which function of the OS, the performance is decreased.

In this paper we present a bottom-up performance prediction model of the components involved in packet forwarding on LINUX-based packet processing systems. It is intended to show which functions of the kernel are involved to accomplish each general task and which share of the CPU cycles per packet is consumed by each of them. We use our model for white-box tests, to explain the router internals and analyse the influence of performance limiting factors. Furthermore, different use cases can be compared in respect to which task consumes more or less cycles and, therefore, is responsible for performance losses or gains. This includes measurements with different router configurations and different types of traffic.

Another common test is to confront the router with different packet rates to evaluate the dynamic behaviour of the kernel. Most of the time these measurements are only carried out up to the point at which the CPU is under full load. Although this is understandable as it is not normal for a system to work continuously under overload, forcing the router under overload over a longer period of time can be caused by misconfiguration or by an attack, targeting the CPU of the software router.

By maintaining generality the model also applies to other forwarding frameworks. While the general steps that are necessary to forward packets remain the same for most frameworks, different approaches can be compared.

The remainder of this paper is structured as follows. Based on a discussion of related work about performance exploration of (Linux) software based packet processing in Section II, we describe the internals of packet processing in Linux-based

software switches and routers, in Section III. In Section IV, we discuss the components that are relevant for the performance breakdown and formulate relations as analytical models for software based packet processing systems. Section V utilizes our models in a case study, for a detailed exploration of the performance in which we discuss the used measurement methodology (in Section V-A). We end with a summary of our contributions in Section VI.

II. Software-based Packet Processing Systems

Wu and Crawford [6] gave an in-depth description of which components are involved and what each of them does when receiving a packet that is destined for an application in user space, already in 2006. To analyse the performance they developed a mathematical model of the whole process: the NIC is modelled using the token bucket algorithm, in which tokens are represented by packet descriptors, and all other processing steps are modelled as queueing processes. They come to the conclusion that, aside from the CPU, the size of the *rx_ring* (packer reception buffer) influences the bottleneck as packets can be dropped if not enough descriptors are available due to "memory pressure".

Bolla and Bruschi [5], [7] showed a deep understanding of the processing steps. They used RFC 2544 compliant test cases to analyse the performance of different hardware and data plane architectures, focusing on the Linux kernel version 2.6 [5]. They used throughput, latency and profiling measurements to evaluate an optimized Linux 2.6 kernel, the Click Modular router and the SMP Linux 2.6 kernel. According to their results, they obtained rates up to 2.5 Gbit/s with an optimized kernel version. Today, the used kernel versions underwent changes in comparison to versions used these days. Anyway, the underlying analysis of the packet processing steps, ranging from the New API (NAPI) to the transmitting NIC, contributed to our performance prediction model. Especially the interaction of interrupts and poll-rate are interesting as it is still valid with today's performance optimized NIC drivers (cf. [8]). Bolla and Bruschi stated the fundamental that *the CPU limits the number of packet headers that can be processed*, while bandwidth and latency of the I/O busses limit the total throughput [7]. Furthermore, they show based on their measurements and analysis that shared data structures like the *tx_ring* of the NIC cause extra CPU cycles, as the access to those is serialized using mutex-like techniques. In addition, the data has to be synchronized between the caches of multiple CPUs causing more overhead. Based on these results they propose a multi-core packet forwarding software architecture. The evaluation shows that they are able to reach a packet rate up to 4 Mpps.

In 2009, Dobrescu et al. [4] revisited software router architectures and performed black-box tests. They were already close to reach the 10 Gbit/s line rate when using software routers with multiple 1 Gbit/s ports and minimally sized packets. Studies from 2013 showed that the trend towards continuously increased performance continued [1]. Beifuß et al. [8] presented a study of the packet reception process and developed a simulation model of it. Emmerich et al. [9] systematically described the bottlenecks concerning hardware and software, which we kept in mind during the design of our model.

Besides general purpose Network IO Software, frameworks that focus on high-speed packet processing have been developed: e.g. netmap [10], Intel DPDK [11], or PF_RING ZC [12]. While these are designed to fulfil general network tasks, they are also able to function as a software router or switch, e.g. DPDK vSwitch [13], Click on netmap [14], and VALE [15]. We also designed our model in conformance with these approaches to packet processing. For an in depth analysis of high speed network IO frameworks we refer to our recent publication and papers referenced in there [16].

III. General Steps of Packet Processing

Our abstract model of packet processing is shown in Figure 1: The NIC at which a packet arrives (1) has to fulfil two tasks. Firstly, the packet is transferred to the main memory of the system (2). A descriptor (pointer) to the memory region is then stored in a queue (3), to keep track of arrived packets. Secondly, the NIC has to inform the OS and thereby invoke the actual processing of the packet (4). Using the pointer to the data of the packet (5), initial operations can be performed (6) before a routing decision is made. This includes for example integrity checks or the application of firewall rules. Now the actual processing takes place, a forwarding decision is made (7). A Longest Prefix Matching [17] algorithm takes the information of the IP header to find the best match in a routing table. After the next hop has been determined, the same operations as described in step (6) can be applied again (8). Furthermore, the layer 2 address for the next hop is being resolved. When the processing of the packet is finished, the egress network board, as determined through the routing lookup, is informed that the packet is ready for transmission (9). Similar to the processing of the ingress NIC, a descriptor to the memory region of the packet is stored in a queue (10), before it finally is transmitted (11).

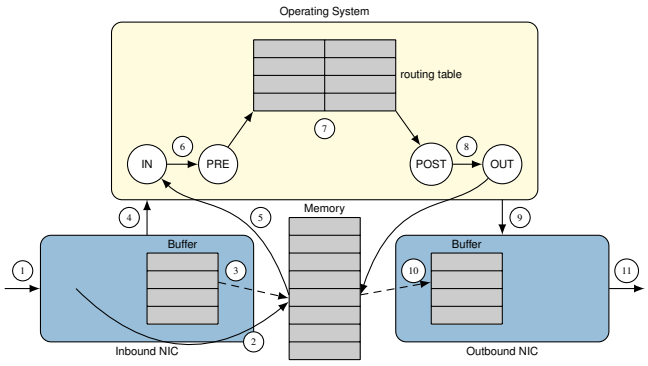

Fig. 1. Abstract model of the packet forwarding process

Our model assumes the forwarding process of network traffic consists of two main parts [18]. The NIC has to handle the transfer of the packets between itself and the main memory and the OS has to make the layer 2 and 3 forwarding decision. The abstract view reveals that the *access to main memory* and the *used data structures* are potential performance limiting factors [9], [16].

A. Packet Processing on Linux Operating Systems

Operating systems implement a general purpose network stack for processing and delivering of packets to user applications that comes with a subsystem for layer 3 forwarding. This allows Linux systems to act as software router.

The steady development process of the Linux networking code [19] requires steady updates to documentation and measurements. In addition to reading kernel source code, publications like [8], [17], [20] provide further insights and measurements. Furthermore, a book by Rosen [21] provides an in-depth explanation of the Linux networking kernel code. Our analysis is based on kernel version 3.7 [22]. For changes since this version the interested reader is referred to Märdian [19] who summarized new features that were added from version 3.7 to 3.16.

1) Reception API – NAPI: With the arrival of the packet, transfer of the packet to main memory, and invocation of further processing by informing the OS, have to be accomplished [8], [20], [21]. The Linux kernel uses the *sk_buff* structure for the internal representation of packets. The NIC stores descriptors, pointing to *sk_buff* structures, into buffers. It uses at least one buffer for packet reception (*rx_ring*) and one for transmission (*tx_ring*). When receiving a packet, the descriptor has to be initialized and allocated with an empty *sk_buff* (cf. Fig.1). The Direct Memory Access (DMA) engine of the NIC transfers the packet data to kernel memory space, the packet is stored in an *sk_buff*, and is ready for further processing. Otherwise it will be discarded by the NIC, because no packet descriptor was available [6]. This feature allows the NIC to drop packets without additional CPU load when the system is overloaded. Thus overload has no impact on the performance of the network stack [17].

To inform the kernel that a packet is available the NIC schedules a hardware interrupt through its interrupt generator. The CPU responds by calling the interrupt handler of the driver. Since the kernel version 2.4.20 the driver uses the New API (NAPI) [21]. Instead of directly enqueueing each packet into the backlog queue of the CPU, the interrupt handler invokes `napi_schedule` that adds a reference to the NIC to the `poll_list` of the CPU and triggers a soft interrupt. When this software interrupt gets served the CPU executes `net_rx_action`, which then successively uses the `poll` driver method of each device present in the `poll_list` in a round-robin fashion to poll a certain number of available packets from their ring buffer. For each of these packets `netif_receive_skb` is called, which then invokes the layer 3 handler of the network stack by executing `ip_rcv` [20].

2) Linux Network Stack: The Linux network stack is designed to support multiple networking protocols including the Address Resolution Protocol (ARP), the Internet Protocol, and Internet Control Message Protocol (ICMP), the user Datagram Protocol (UDP) and Transmission Control Protocol (TCP) [23]. The network stack processes each packet layer by layer. In order to forward a packet, however, only the layers up to the network layer are involved. For IPv6 packets separate functions exist that follow the same naming conventions (for instance `ipv6_rcv` instead of `ip_rcv`).

a) Prerouting Processing: The first function that gets invoked, `ip_rcv`, sanity checks for the length of the IP header, the checksum, and the length of the payload [17], [20]. If any of these checks fail, the packet gets dropped.

The packet next passes the netfilter subsystem. The purpose of this subsystem is to filter out packets and to perform changes [17]. It is used for firewalls implemented with iptables [24], network address translation (NAT) [25], modifying the contents of packet headers (mangling), connection tracking or gathering network statistics [20].

Afterwards, `ip_rcv_finish` continues and determines whether the packet has to be delivered locally or forwarded to another host. Hence, via a call to `ip_route_input_noref` the packet gets passed to the routing subsystem. The last action of `ip_rcv_finish` is to invoke `dst_input` of the *sk_buff*. Depending on the result of the routing lookup, this method calls the corresponding function for next steps.

b) Routing Subsystem: In Linux all threads and processes use the same shared routing table. Hence, the first task of `ip_route_input_noref` is to acquire a read-copy-update (RCU) lock [26]. As it synchronises the access with low overhead and wait-free reads, RCU-locks are widely used within the Linux kernel [27] and replace other locking mechanisms, whenever possible. Aside from sorting out for instance broadcast and multicast packets to handle them separately, a lookup in the routing table is initiated by calling `fib_lookup`. Linux uses the forwarding information base (FIB) trie [20]. The FIB trie contains the routing entries, each being a mapping of a subnet or IP address to a next-hop and outgoing interface. A lookup in the specified table is performed with `fib_table_lookup`. The basic idea of this function is to go through the FIB trie and try to find the best match for the destination IP address of the packet according to the longest prefix matching algorithm [20].

Depending on the result of the lookup, the *sk_buff* gets updated. The `dst_input` method is either set to `ip_local_deliver` for delivery to the local host or to `ip_forward` if the packet has to be forwarded. In the latter case the next-hop and outgoing interface get updated, too.

c) Postrouting Processing: If the packet gets delivered to the local host it gets passed to the next layers for further processing (for instance TCP or UDP), until it reaches the application in userspace via the socket API [28]. As this path is not subject of this paper we refer to [6], [17], [20] for further description.

In case of a forwarding decision to the next host the time to live (TTL) of the IP header is being decreased. If the TTL reaches zero, the packet gets dropped and an ICMP time exceeded message is triggered [20]. The length of the packet is compared to the maximum transmission unit of the outgoing link. Afterwards, potential IP options are processed and statistics get updated. Now a netfilter hook (`NF_INET_POST_ROUTING`) is invoked again. If fragmentation is needed and allowed it is handled via `ip_fragment`. To resolve the next-hop to a MAC address the packet is passed to the neighbouring subsystem via `__ipv4_neigh_lookup_noref`. If the entry is cached, the respective header field can be updated, otherwise the MAC address has to be resolved for instance using ARP for IPv4 or the neighbour discovery protocol for IPv6 packets. When the data link header has been completed, the packet is passed to the transmitting NIC by invoking `dev_queue_xmit`.

3) Transmission API: To control and schedule the traffic between kernel and NIC, queueing disciplines are used [29]. The default queueing discipline, *qdisc*, uses the FIFO strategy to manage the packets [20]. `dev_queue_xmit` enqueues each packet into the *qdisc* of the device that has been determined by the routing lookup. Then a spin-lock for the respective *qdisc* is acquired. These locks are the Linux implementation of active waiting-mutexes [30], hence, a transmission queue can only be used by one process at a time. When this lock is successfully acquired and the device is running and not stopped because for instance *tx_ring* is full, all the packets in the *qdisc* are handled. Another lock (`HARD_TX_LOCK`) has to be acquired before the `dev_hard_start_xmit` function of the driver is invoked. This method loads the packet descriptor into the `tx_ring` [20]. Finally, the NIC is informed of the packets that are ready for transmission.

The NIC informs the CPU with an interrupt when the packet transmission is completed After the interrupt, the meta data of the *sk_buff* structure is deallocated and the memory is freed.

B. Performance Limiting Factors

While the Linux design is convenient for running applications up to a rate of 1 Gbit/s, it rapidly reaches a limit when trying to work with rates up to 10 Gbit/s [9]. Furthermore, processing is not optimized for a forwarding-only task. Impediments for packet processing in software routers have been identified [1], [2], [5], [6], [9].

1) CPU: The most common bottleneck is the CPU. The more complex the processing of a packet is, the more CPU-cycles are consumed. A CPU operating at a frequency of CPU_{freq} Hz provides a number of CPU cycles per second $C_{available}/s$ and needs a total of C_{total} CPU cycles to process a single packet. The resulting maximum packet rate R_{max} can be calculated with

$$R_{max} = \frac{CPU_{freq}}{C_{total}} = \frac{C_{available}/s}{C_{total}} \quad (1)$$

This is valid when the CPU operates at its maximum frequency and full capacity. Thus, R_{max} can only be influenced by C_{total}.

2) Memory: The complete data of a packet, consisting of meta-data and payload, passes the system memory (cf. Section III-A). This leads to two different obstacles when forwarding packets at high-speed: costs for allocation and deallocation of memory that is influenced by the complexity of the data structure (*sk_buff*). Especially when working at high packet rates, constantly allocating and deallocating memory causes significant overhead that reduces the performance of the system [2]. The complexity of the *sk_buff*-structure that is compatible with numerous protocols and thus contains the meta-data of several protocols increases the workload. For the task of forwarding a packet, however, only the layer 2 and 3 headers are needed. Higher layer meta-data is not processed, as the packet is not passed up to these protocol handlers. Dorado et al. claim that "63% of the CPU usage in the reception process of a single 64B sized packet" [2] is consumed by *sk_buff* related operations. Another proof is the software switch, DPDK vSwitch [13] that uses a purged data structure to increase the speed compared to Open vSwitch.

3) Generality of the Software Design: At multiple occasions of the forwarding process data structures are protected with locking mechanisms to prevent faulty data caused by race conditions. One example, where multiple locks are used, is the access to the rx NIC [17]. For instance, the *qdisc* is protected with a spin-lock. Whenever a spin-lock is already locked, many CPU cycles are potentially wasted for active waiting of other processes [30]. Anyhow, locks can be omitted for dedicated software data planes where each CPU core gets packets in exclusively used rx buffers and stores them to dedicated tx buffers. So although Linux routers waste CPU cycles for obtaining and releasing these locks a Linux router scales linearly as there is no shared access on rx and tx rings.

4) Cache Size: With large data structures the size of the available CPU caches can limit the performance. If the data surpasses the size of the cache the number of cache misses can increase significantly and cycles are spent fetching the missed information. Cache thrashing causes the cycles per packet to increase as the actual processing of the packet is idling for data. This effect may occur due to the size of the routing table. For comparison today's backbone routers, which use the Border Gateway Protocol, hold up to 500,000 entries [31].

Additional processing tasks like firewalls, NAT, or collecting statistics, that can be hooked via netfilter rules in the Linux network stack at multiple occasions may require additional CPU cycles, (cache) memory, and locks.

IV. Performance Break Down

Profiling reveals how many CPU time has been spent on each function. Due to the high complexity of packet processing these results are hardly useful without further treatment. Filtering out all components that do not contribute to packet processing and grouping of the remaining functions allows for further analysis. The model which defines the groups is shown in Figure 2. It allows to compare different scenarios and to compare CPU cycles spent for each group of processing tasks.

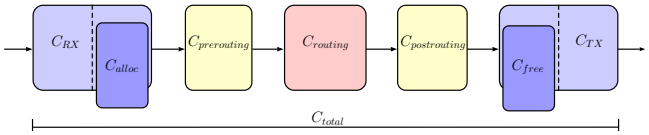

Fig. 2. Performance prediction model

We defined the following groups:

C_{RX} contains all functions for reception via the NIC and its driver. The tasks are considered to be finished after the driver handed the packet over to the OS for the actual processing of the packet headers.

C_{alloc} contains allocation and transfer of the packet data to the main memory. Though this could be part of C_{RX}, the separate group for buffer allocation accommodates for the significant impact of allocation on the overall performance.

$C_{prerouting}$ for processing steps after the packet was received from the driver but before the routing decision is made. The pre-routing processing includes for instance integrity checks and the traversal of firewalls or NAT.

$C_{routing}$ for lookup of the packet route – i.e. a FIB lookup.

$C_{postrouting}$ for tasks after the routing decision – for instance fragmentation or the resolution of MAC addresses.

C_{free} for packet buffer deallocation – analogue to C_{alloc}, the deallocation of the packet buffer.

C_{TX} transmitting side NIC tasks, analogous to C_{RX}.

C_{other} is a group for all functions that remained unassigned. It contains functions that are invoked in different groups and thus cannot be assigned to an own group and background tasks, which cannot be turned off. Fewer CPU time in this group leads to more precise statements and conclusions regarding the other groups.

Summing up the individual parts leads to the overall consumed cycles per packet

$$C_{total} = \sum_{i \in \{RX, alloc, \ldots, other\}} C_i, \tag{2}$$

With Equation 1 we can predict the maximum packet rate R_{max}.

Vice versa, one can measure R_{max} and either CPU_{load} or C_{active} - the amount of cycles that the CPU actively spent processing per second[1] - to calculate C_{total}:

$$C_{total} = \frac{C_{available}/s * CPU_{load}}{R_{max}} = \frac{C_{active}}{R_{max}} \tag{3}$$

A load that produces $CPU_{load} = 100\%$ avoids artefacts that can be caused by interrupts [8] or empty poll cycles [16]. To parametrize the model one has to measure how many CPU cycles are consumed by each function and then sort them into the groups of the model.

[1] C_{active} plus the amount of cycles spent idle must equal $C_{available}$

V. Parametrization of the Model

In this section we demonstrate the application of our model (cf. Section IV). The described (and further) measurements were performed by Dominik Scholz [32]. They were restricted to single core measurements as software routers scale linearly with the number of cores due to optimal parallelism [1]. The methodology was developed in previous measurements [1], [8], [9], [16], [33], [34].

A. Methodology

The measurements were performed on serves equipped with Supermicro X9SCM-F mainboards and Intel Xeon E3-1230 v2 CPUs with 3.3 GHz and 8 MB L3 cache, Dual channel 16 GB ECC DDR3 SDRAM clocked at 1333 MHz, and Intel X520-SR2 (DuT) and X520-SR1 (load generator and sink) based on the Intel 82599 Ethernet controller. Hyper-threading, turbo boost, and power saving features were disabled. To avoid fluctuations from CPU migrations, all processing tasks and interrupt handlers were always explicitly pinned to specific CPU cores. All offloading features of the NICs and pause frames were disabled. The size of the *rx_ring* was increased to its maximum of 4096 entries.

We ran Grml Live Linux 2013.02 with the Linux kernel version 3.7, and the ixgbe 3.14.5 NIC driver. Some experiments required a kernel with base pointers (compiler option `-fno-omit-framepointer`) to enable the profiling tool `perf` to create function call-stacks. Previous experiments showed that base pointers reduce the total throughput by about 15% [9].

All unnecessary background tasks were disabled to minimize possible interferences that affect the measurements. To generate packets up to the line rate of 10 GbE a customized version of the `pfsend` packet generator based on the pf_ring DNA packet processing framework has been used. To count the sent and received packets the statistics register of the NIC has been used.

The Linux profiling tool `perf` was used for all measurements. The sampling was restricted to be system-wide of all applications running on the single core to which the traffic was pinned. The result is the actual number of CPU cycles per second C_{active}. This value is used to calculate the cycles per packet C_{total} using Equation 3 and to scale the values obtained with `perf record`[2]. When we used the alternative kernel version, compiled with base pointers, the resulting binary file of `perf record` contained information about the function call-stack.

We used profiling data to create flame graphs. Although Flame graph visualizations are too detailed for usage in papers the visualizations of the call stacks were helpful to identify CPU-consuming functions. Such an exemplary flame graph is displayed in Figure 3. We also used a modified colouring schema for Flame Graphs that clusters functions of the Flame Graphs and paints them using identical colour.

[2] `perf record` is a tool, part of `perf` that allows sampling of counters with configurable rate and periods

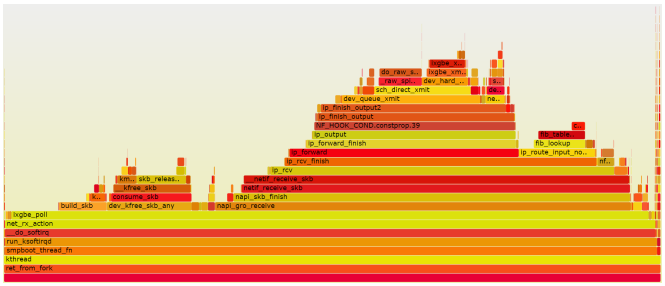

Fig. 3. Flame graphs are useful to get an overview to CPU cycle consumption: this exemplary flame graph visualizes the distribution of CPU cycles spent on the different functions of the Linux kernel

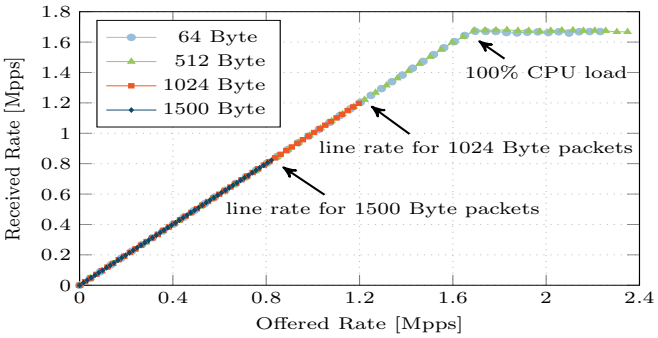

Fig. 4. Packet rates for different packet sizes

All profiling tests were run five minutes per offered rate to obtain reliable values. Additional sampling showed that under full workload the relative cycles of the functions were equal with a deviation of less than $\pm 1\,\%$ in a 95 % confidence interval. Therefore, the measurements with the maximum received rate shown in Figure 4 - thus maximum CPU load - were used to parametrize the model. For 64 Byte and 512 Byte packets the line rate of 10 Gbit/s was not reached. However, the router was at full load and, when increasing the offered load even more, started dropping packets. These packets are dropped by the NIC without stressing the CPU. Hence, for 64 Byte and 512 Byte packets the model is parametrized with mean values.

B. Increasing the Packet Size

Our first investigation analyses the system behaviour different packet sizes, ranging from the minimum size of 64 Bytes to a close-to-maximum size of 1500 Bytes. Previous work showed that for small packet sizes the line rate of 10 Gbit/s can not be reached [1]. Instead, when the CPU is completely utilized, additional packets are dropped by the NIC. For larger packets, however, the line rate can be reached, leading to a not fully utilized CPU.

A closer look at the different parts of our performance prediction model confirms that packet size has no influence on the packet rate (cf. Section II): The rx NIC handles all incoming packets the same, while *sk_buff*'s are always allocated to fit a maximum sized packet. The network stack only needs the information of the layer 2 and 3 headers to make a forwarding decision.

1) Impact on the Packet Rate: Figure 4 shows the received packet rates for the tested packet sizes. The offered rate equals the received rate of packets until the line rate is hit or the CPU is at full capacity. As expected, for the smaller packet sizes of 64 and 512 Bytes, the line rate is not reached. Instead, the CPU reaches full capacity when offering 1.67 Mpps. The received rate remains at this maximum when offering even more packets per second as additional packets get dropped by the rx NIC. For 1024 and 1500 Byte packets the line rate is reached at 1.2 Mpps and 0.82 Mpps respectively but the CPU is not fully utilized for these packets.

TABLE I
MAXIMUM PACKET RATE, MAXIMUM CPU LOAD AND CYCLES PER PACKET FOR DIFFERENT PACKET SIZES

Packet Size	R_{max} [Mpps]	CPU_{load}	C_{total}
64 Byte	~ 1.67	$\sim 100\%$	1979
512 Byte	~ 1.67	$\sim 100\%$	1969
1024 Byte	~ 1.20	$\sim 75\%$	2056
1500 Byte	~ 0.82	$\sim 55\%$	2203

Table I summarizes the maximum packet rate and the respective CPU load for all tested packet sizes. The resulting cycles per packet were calculated with Equation 3 with $C_{available} = 3.3 * 10^9$ CPU cycles per second for all further calculations. C_{total} is constant for different packet sizes, except in scenarios where the packets per interrupt decrease and thus the costs for interrupts per packet increase. Explicit listing of interrupt costs can help to explain this effect, but is not applicable for polling based packet reception (cf. Section II).

2) Distribution of CPU-cycles: Figure 5 shows the results of the `perf record` measurements for 64 Byte IPv4 packets. Until 1.67 Mpps, the consumed CPU cycles increase linearly for almost all functions with the offered packet rate. The functions `ip_rcv`, `fib_table_lookup` or `dev_queue_xmit` account for large shares in CPU utilization. The function with the highest percentage is

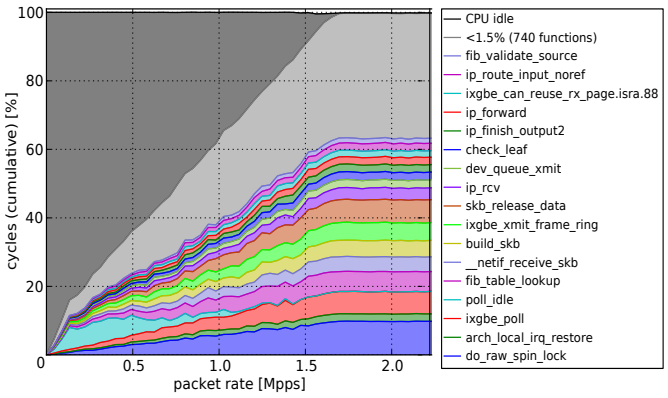

Fig. 5. Distribution of CPU-cycles across functions for 64 B packets

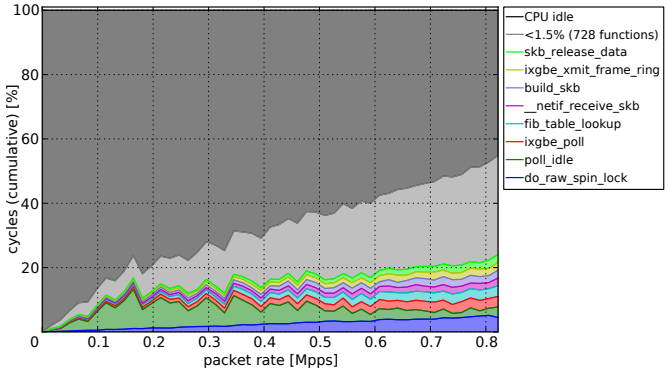

Fig. 6. Distribution of CPU-cycles across functions for 1500 B packets

`do_raw_spin_lock`, which is implementing active waiting to acquire a lock for a resource that is shared among multiple processes. This shows that locking mechanisms as they exist in general purpose systems, like Linux, are limiting the performance of the network stack significantly. As modern NICs offer multiple rx_rings these locks could be removed from the general packet transmission process in Linux as long as it is ensured that each rx_ring is used *exclusively* by one core.

The number of polls at a certain frequency is adjusted dynamically by the NAPI driver [8]. This way, the latency is reduced when only processing few packets, while under higher loads the overhead is reduced. This effect of the interrupt throttle rate explains the rapidly increasing CPU activity for the first ∼0.2 Mpps [9]. With an increasing packet rate the number of interrupts increase, too. When reaching a certain maximum, the number of packets polled per interrupt is increased instead. This then leads to the less steep slope of CPU activity after 0.2 Mpps.

As `perf` samples all the functions of the kernel, more than 750 different names appear in the output, many of which consume close to no cycles. The values of all functions are constant after hitting 1.67 Mpps. This can be explained with the *rx_ring* of the NIC. If not enough packet descriptors are polled from the ring, no free descriptors are available for incoming packets, hence, these are dropped. This is done without any time loss, resulting in no impact on the overall performance. Therefore, measurements with packet rates that lead to full CPU load are treated as if they were multiple tests for the maximum received packet rate. These are then used to calculate average values with errors bars showing the 95 % confidence intervals.

The effect that 1500 Byte packets need slightly more cycles compared to smaller packets can be explained with the associated profiling graph in Figure 6. As the line rate is hit at a rate of 0.82 Mpps, the CPU is not under full load and the graph equals the one for 64 Byte packets for 0 Mpps to 0.82 Mpps (cf. Figure 5). Therefore, the idle functions, in particular `poll_idle`, are still present and consume additional cycles compared to smaller packet sizes.

The last step to parametrize the performance prediction model is to divide each function into a group of the model. Table II shows the resulting values for the model when using the measurements with different packet sizes. Figure 7 shows a plot of these values.

TABLE III
TOP 5 CYCLE-CONSUMING FUNCTIONS IN C_{other} (64 B PACKETS)

Function	Cycles	appears in
do_raw_spin_lock	194	all Groups
arch_local_irq_restore	43	C_{RX}, $C_{prerouting}$
arch_local_irq_save	21	C_{alloc}, C_{free}
__phys_addr	20	C_{RX}, C_{free}, C_{TX}
spin_unlock	15	C_{alloc}, $C_{postrouting}$, C_{free}, C_{TX}

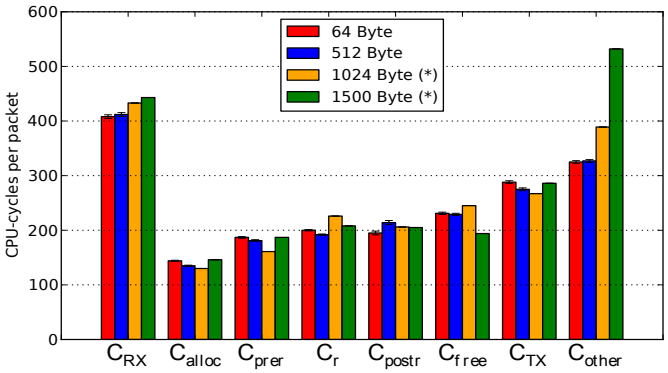

Fig. 7. Per packet CPU-cycles distribution for different packet sizes

The processing by the rx and tx NICs and their respective drivers consumes approximately 35 % of the whole cycles with minimal increases for larger packets at the receiving side, which are explained by the increased DMA transfer time. The allocation and deallocation is independent from the size of the packet, because a maximum sized *sk_buff* is allocated anyway. However, both together consume up to 20 % of the total cycles, showing that per packet allocation and deallocation are a costly task. The processing of the network stack, represented by $C_{prerouting}$, $C_{routing}$ and $C_{postrouting}$, uses only the information of the necessary headers, hence the size of the payload does not matter. Each individual part takes about 10 % of the cycles, without the use of any additional processing.

A downside of our model is the inaccuracy caused by the functions in C_{other}. To solve this issue, a more sophisticated method to distribute the functions into the groups of the model can be used, factoring in what share of each function is used per group.

C_{other} represents the final 15 % of CPU cycles, including functions that appear in more than just one group or only consume a close to zero percentage. The five functions with the biggest share are shown in table III. Furthermore, it is shown in which parts of the model they occur. Function `do_raw_spin_lock` consumes close to 10 % of the CPU cycles. Approximately 90 % of the calls to `do_raw_spin_lock` occur when transmitting a packet by the tx NIC, hence 90 % of the cycles for this function

TABLE II
PER PACKET CPU-CYCLES DISTRIBUTION FOR DIFFERENT PACKET SIZES (* AND RATES)

Packet Size	C_{RX}	C_{alloc}	$C_{prerouting}$	$C_{routing}$	$C_{postrouting}$	C_{free}	C_{TX}	C_{other}	C_{total}
64 Byte	408	144	187	200	195	231	288	325	1979
512 Byte	412	135	181	192	214	229	275	327	1969
1024 Byte (*)	433	130	161	226	206	245	267	389	2056
1500 Byte (*)	443	146	187	208	205	194	286	532	2203

can be added to C_{TX}. Doing so leads to 40-45 % for rx and tx NIC processing. This conforms to previous research [2], [10]. Anyway, for higher accuracy we continue to count `do_raw_spin_lock` to C_{other}.

For larger packet sizes, in particular 1500 Byte, C_{other} consumes even more cycles. This is again explained by the idle functions, for instance `poll_idle` with $\sim$ 40 cycles, are added to this group. This was done mainly for the reason that these functions do not actually process the packet and would just obscure the results, making them less comparable with the smaller packet sizes.

In summary we see a constant behaviour across all parts of the model. Only C_{other} increases due to the increased relative impact on a packet rate that is decreased due to the limiting Ethernet link bandwidth (see (*) in table II). Therefore, our following tests only use 64 Byte packets to reduce the overall number of test cases. Our tests show impediments of locking mechanisms and per-packet allocation and deallocation of packets. This is the reason why most high-speed packet processing frameworks implement different techniques to solve especially these issues [10].

C. Increasing the Number of Flows

In the following we increase the number of flows, each represented by a packet with an unique destination address, to analyse the behaviour of the routing subsystem for a growing routing table. The measurements with 64 Byte packets of the previous section are used as reference, representing one flow. The general assumption is, that only the behaviour of the routing subsystem differs for varying numbers of flows. As all packets are pinned to the same core, neither the NIC nor the network stack treats the packets differently in any way. The traffic itself uses repetitive, linear increasing IPv4 addresses as destination. This was done to test the worst case, constantly changing destinations, which means that the results of the lookup can not be cached. However, as kernel version 3.7 uses no routing cache anyway, this behaviour is given by the design of the routing subsystem. The expected result is, that as long as the necessary data of the FIB trie fits into the cache of the CPU, the performance does not decrease significantly.

The packet rates for different numbers of flows are shown in Figure 8. Concurring with previous results, the rate decreases slightly ($\sim$ 0.1 Mpps) for tests with more flows. The run with more than one million flows however shows a maximum packet rate of only 1.39 Mpps. The packet rate decreases when increasing the offered rate even further. This effect can be explained by the cache behaviour of the CPU (cf. V-D).

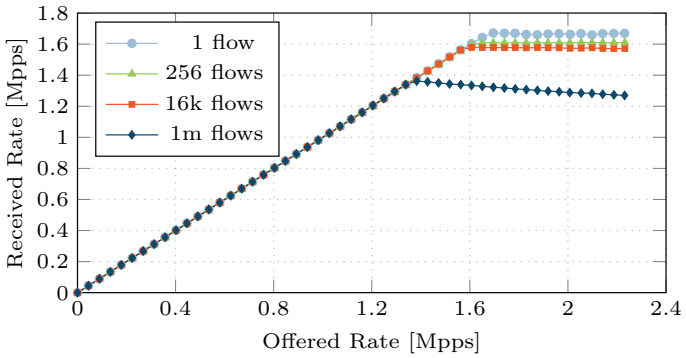

Fig. 8. Packet rates for different number of flows

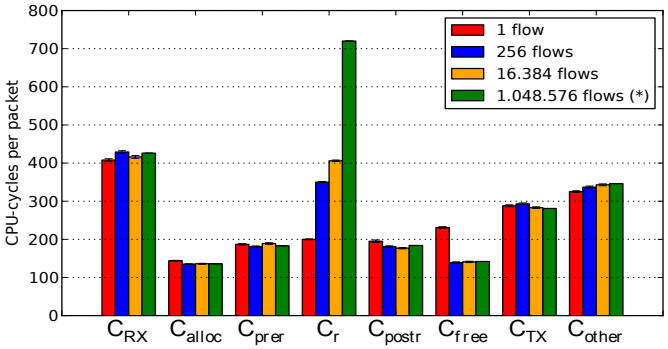

Fig. 9. Per packet CPU-cycles distribution, different number of flows, 64 B

Parametrizing the model with the results of these measurements lead to the values shown in table IV and presented in Figure 9. As expected, more flows have no impact on the processing by the NIC and its driver. The only anomaly is given by C_{free}. When having a closer look at which functions, that are divided into C_{free}, cause this behaviour, only `skb_release_data` shows a significant deviation. In the last row of table V, for 1 flow this function consumes approximately 133 cycles, while for the other tests this is reduced to under 40 cycles. A possible explanation for this is the dynamic memory management of the OS. For one flow, the packet rate is slightly increased, hence, more packets per second have to be managed at the same time. This means that more locks must be acquired to release the data again, which could cause this increased consumption of cycles.

The pre- and postrouting processing is constant, as the same operations are performed, independently of the destination address. Only the routing subsystem shows an increased

TABLE IV
PER PACKET CPU-CYCLES DISTRIBUTION FOR DIFFERENT NUMBER OF FLOWS, 64 BYTE

Number of Flows	C_{RX}	C_{alloc}	$C_{prerouting}$	$C_{routing}$	$C_{postrouting}$	C_{free}	C_{TX}	C_{other}	C_{total}
1	408	144	187	200	195	231	288	325	1979
256	429	135	181	350	181	139	293	337	2048
16.384	416	136	189	406	177	141	283	343	2093
1.048.576 (*)	426	136	183	720	184	142	281	346	2420

TABLE V
FUNCTIONS WITH THE LARGEST DIFFERENCES IN THEIR RESPECTIVE GROUP FOR DIFFERENT NUMBERS OF FLOWS

Function	Group	1 flow	256 flows	1.048.576 flows
fib_table_lookup	$C_{routing}$	116	238	423
check_leaf	$C_{routing}$	45	89	361
skb_release_data	C_{free}	133	37	35

consumption of cycles for increasing amounts of routing table entries. This is matching the assumptions, as no routing cache is implemented. Only two functions, listed in table V have a significant impact on this increase of up to 3.5 times the value for one flow. Both are directly involved for finding the best match in the FIB. This means that finding the best match in the FIB trie lasts longer, as the size of the tree-like structure that represents the routing table is increased, hence, it takes more time to traverse it until the correct leaf is found.

D. Limitations through the Cache Size

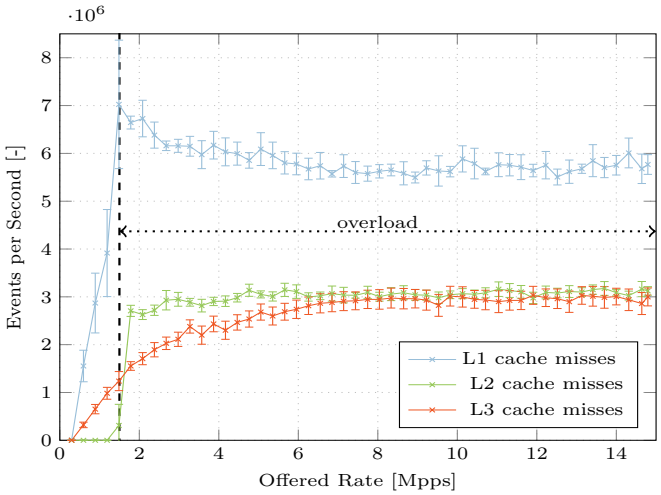

Fig. 10. Cache misses for 1.048.576 Flows.

The effect that for 1.048.567 flows the received rate decreases after hitting the peak rate when further increasing the offered rate (cf. Figure 8), is deviant from all other measurements. To better understand this behaviour we repeated the test with a bit rate up to the maximum of the 10 GbE links. The received rate decreases until it approaches approximately 1.1 Mpps at an offered rate of 8 Mpps and stays at this level. A decreased packet rate means the cycles per packet increase. When having a look at which functions consume these additional cycles, only functions in $C_{routing}$ show an increased consumption.

Figure 10 shows the misses across the level 1, 2 and 3 caches of the CPU. The curve of the L1 misses is similar to the packet rate, while the L2 cache misses are at a nearly constant level of half of the L1 misses. The most interesting is the progression of the last level cache misses. In the beginning they are rapidly increasing which is not changing when surpassing the mark of the maximum packet rate - in this case the maximum L1 cache misses. Instead the L3 cache misses approach the L2 misses, until all L2 cache misses miss the L3 cache, too. We attribute this effect to the increased number of packets that are handled by the router, resulting in more requests per second to the routing table. Anyhow, the structure is too large to fit completely into the L3 cache. Already a low estimated value for one routing entry of 20 Bytes (4 Byte IPv4 address, 4 Byte subnet mask, outgoing interface, metric, meta-data, ...) leads to a routing table of approximately 21 MiB for 1.048.576 flows, which is vastly larger than the 8 MiB level 3 cache[3]. Together with the fact, that always different destination addresses are looked up, referencing different parts of the FIB trie, this causes more and more L3 cache misses. This in turn slows down the whole processing of the packets as they are backed-up, causing the cycles per packet to increase. The worst performance is reached when all L3 references miss the cache and thereby cause the maximum number of memory accesses.

In summary the performance is not decreased significantly when increasing the number of flows. The only identified difference is, as expected, finding the best match in the routing table, as this structure keeps growing with the number of flows. Once this number exceeds a certain value, however, another limiting factor, the size of the L3 cache, gains importance. If the FIB is larger than the cache, the overall performance is reduced because of worse cache behaviour. Generally this shows, that the size of used data structures must be kept in mind, as bad cache behaviour may lead to unexpected effects which drastically reduce the performance.

While a routing table with more than one million entries may be unrealistic this loss of performance might already occur for a number of flows between 65.536 and 262.144. Hence, this effect is relevant, as BGP routers can have more than 500.000 routing entries nowadays [31]. These values are continuously rising, causing other problems, too, as for instance backbone routers run out of memory [35].

[3]For reference, using the same estimated size for a routing entry, 16.384 entries need a total of only 328 KiB, hence the complete routing table fits into the level 3 cache.

VI. Summary

We presented an analytical model for software based packet processing systems that can predict upper bounds for the maximum throughput. Our model can be parametrized to fit arbitrary systems. Our case study where we broke down the performance of a Linux router described the influence and relationship of FIB size, network stack, and packet processing tasks on the performance. We encourage to use our profiling data for model parametrization.

Acknowledgments

This research has been supported by the DFG as part of the MEMPHIS project (CA 595/5-2) and the BMBF in context of the EUREKA-Project SASER (01BP12300A). We acknowledge the valuable contributions from Alexander Beifuß, Torsten Runge, and Sebastian Gallenmüller.

References

[1] T. Meyer, F. Wohlfart, D. Raumer, B. E. Wolfinger, and G. Carle, "Measurement and Simulation of High-Performance Packet Processing in Software Routers," *Proceedings of Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und Verteilten Systemen*, September 2013, 7. GI/ITG-Workshop MMBnet 2013.

[2] J. L. Garcia-Dorado, F. Mata, J. Ramos, P. M. S. del RÃo, V. Moreno, and J. Aracil, "High-performance network traffic processing systems using commodity hardware," *Data Traffic Monitoring and Analysis*, pp. 3–27, 2013.

[3] IEEE, "IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Forceofficial web site," http://www.ieee802.org/3/ba/, accessed: May 2015.

[4] 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," *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 15–28, 2009.

[5] R. Bolla and R. Bruschi, "Linux Software Router: Data Plane Optimization and Performance Evaluation," *Journal of Networks*, vol. 2, no. 3, 2007.

[6] W. Wu and M. Crawford, "The Performance Analysis of Linux Networking - Packet Receiving," *International Journal of Computer Communications*, 2006.

[7] R. Bolla and R. Bruschi, "PC-based Software Routers: High Performance and Application Service Support," *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pp. 27–32, 2008.

[8] A. Beifuß, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle, "A Study of Networking Software Induced Latency," in *2nd International Conference on Networked Systems (NetSys)*, March 2015.

[9] Paul Emmerich and Daniel Raumer and Florian Wohlfart and Georg Carle, "Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems," in *Fourteenth International Conference on Networks (ICN 2015)*, Barcelona, Spain, Apr. 2015.

[10] L. Rizzo, "Revisiting Network I/O APIs: The Netmap Framework," *Commun. ACM*, vol. 55, no. 3, pp. 45–51, Mar. 2012.

[11] "Impressive Packet Processing Performance Enables Greater Workload Consolidation," in *Intel Solution Brief*. Intel Corporation, 2013, Whitepaper.

[12] "ntop.org: PF_RING ZC," http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/, last visited 2015-07-10.

[13] I. O. S. T. Center, "Packet Processing - Intel DPDK vSwitch," https://01.org/packet-processing/intel%C2%AE-onp-servers, accessed: September 2014.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[15] L. Rizzo and G. Lettieri, "VALE, a switched ethernet for virtual machines," *CoNEXT*, pp. 61–72, 2012.

[16] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of Frameworks for High-Performance Packet IO," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2015)*, Oakland, USA, May 2015.

[17] C. Benvenuti, *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2005, vol. 1.

[18] R. Bolla and R. Bruschi, "An Effective Forwarding Architecture for SMP Linux Routers," *Telecommunication Networking Workshop on QoS in Multiservice IP Networks*, pp. 210–216, 2008.

[19] L. Märdian, P. Emmerich, and D. Raumer, "What's New in the Linux Network Stack?" in *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2014*, München, Germany, Apr. 2015.

[20] M. Rio, M. Goutelle, T. Kelly, R. Hughes-Jones, J. Martin-Flatin, and Y. Li, "A Map of the Networking Code in Linux Kernel 2.4.20," *Technical Report DataTAG-2004-1*, March 2004.

[21] R. Rosen, *Linux Kernel Networking: Implementation and Theory*. Apress, 2013, vol. 1.

[22] F. Electrons, "Linux Cross Reference Version 3.7," http://lxr.free-electrons.com/source/?v=3.7, accessed: August 2014.

[23] J. W. Buse, "Linux Network Stack," http://www.linux.org/threads/linux-network-stack.4620/, accessed: May 2015.

[24] I. set, "iptables," http://ipset.netfilter.org/iptables.man.html, accessed: May 2015.

[25] K. Egevang and P. Francis, "RFC 1631: The IP Network Address Translator (NAT)," 1994.

[26] LWN.net, "Using read-copy-update," http://lwn.net/Articles/37889/, accessed: May 2015.

[27] P. E. McKenney, "RCU Linux Usage," http://www.rdrop.com/users/paulmck/RCU/linuxusage.html, accessed: May 2015.

[28] die.net, "socket - Linux man page," http://linux.die.net/man/7/socket, accessed: May 2015.

[29] T. L. D. Project, "Components of Linux Traffic Control," http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/components.html, accessed: May 2015.

[30] Wikipedia, "Spinlock," http://goo.gl/f1Dqhx, accessed: May 2015.

[31] APNIC, "BGP Routing Growth in 2011," http://labs.apnic.net/blabs/?p=25, accessed: May 2015.

[32] D. Scholz, "A Model for Performance Prediction in PC-based Packet Processing Systems," in *Bachelor's Thesis*, München, Germany, 2015.

[33] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "A Study of Network Stack Latency for Game Servers," in *13th Annual Workshop on Network and Systems Support for Games (NetGames'14)*, Nagoya, Japan, Dec. 2014.

[34] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance Characteristics of Virtual Switching," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet'14)*, Luxembourg, Oct. 2014.

[35] BGPMON, "What caused today's Internet hiccup," http://www.bgpmon.net/what-caused-todays-internet-hiccup/, accessed: August 2014.