Building a Low Latency Linux Software Router

Alexander Beifuß[1], Torsten M. Runge[1], Daniel Raumer[2], Paul Emmerich[2], Bernd E. Wolfinger[1], Georg Carle[2]

[1]Universität Hamburg, Department of Computer Science, Telecommunications and Computer Networks
{beifuss|runge|wolfinger}@informatik.uni-hamburg.de
[2]Technische Universität München, Department of Computer Science, Network Architectures and Services
{raumer|emmericp|carle}@in.tum.de

***Abstract*—Packet processing (e.g. routing, switching, firewall) with commodity hardware is a cost-efficient and flexible alternative to specialized networking hardware. On commodity hardware the CPU typically becomes the bottleneck in packet processing. However, in well-known QoS mechanisms (e.g. DiffServ), the outgoing link is assumed to be the bottleneck. This limitation is unfavorable, in particular for latency-sensitive applications (e.g. VoIP, video conferencing, online gaming). Thus, we propose and implement a QoS concept for a Linux software router to prioritize latency-sensitive traffic at the incoming network interface. Our testbed measurements show that our prototype implementation improves the packet processing w.r.t the latency of latency-sensitive traffic even under high traffic loads.**

***Keywords*— commodity hardware; packet processing; quality of service; low latency; NIC driver**

I. Introduction and Related Work

Since decades, packet processing with specialized networking hardware like hardware routers has been the state of the art. Nowadays, cost-efficient commodity hardware has benefited from many optimizations (e.g. multi-core CPUs, multi-queue NICs, DCA, DMA, PCIe) to exploit parallelism in the packet processing with software [1], [2]. By this way, the packet processing software like the network interface card (NIC) drivers and the operating systems (OS) also received several enhancements like interrupt moderation for saving CPU cycles in high-load situations. Thus, so-called software routers became a potential replacement for specialized networking hardware in many cases (e.g. campus networks). In contrast to hardware routers, software routers are more cost-efficient and more easy to extend, which allows for fast adaptation and introduction of new features.

Many research projects show that the CPU constitutes the bottleneck in the packet processing with commodity hardware [1]–[4]. However, there is only limited support for differentiated ingress traffic treatment regarding the packet processing by the CPU bottleneck. For instance, Linux only supports packet filtering but no class-based traffic differentiation at the ingress. Unfortunately, the well-known QoS approaches like DiffServ [5] and IntServ [6] are only applicable as queuing disciplines (Qdisc) at the egress because the outgoing link is assumed to be the bottleneck. Consequently, the absence of ingress traffic classification and prioritization might have a negative impact on applications which have specific quality of service (QoS) requirements (e.g. high bandwidth, low latency, low jitter). Additionally, this problem is strengthened by the increase of real-time traffic such as voice over IP (VoIP), video conferencing, video on demand (VoD) or online gaming.

We argue that QoS-sensitive traffic should also be prioritized at the ingress network interface to achieve QoS-aware packet processing with commodity hardware. In this manner, we presented and evaluated a new QoS concept in our previous work [7], [8], in which we simulated the ingress QoS packet processing and showed improvements for QoS-sensitive traffic. In our QoS concept, the ingress traffic is classified by the NIC into dedicated receive queues (Rx rings). By this way, the traffic classification is offloaded from the CPU to the NIC. Then, QoS-sensitive traffic can be prioritized according to a configurable scheduling strategy. This QoS concept is also applicable for other packet processing systems (e.g. switches, load balancers, firewalls, end systems).

Furthermore, optimized networking frameworks like netmap [9] or DPDK [10] were proposed to bypass the Linux networking stack. These approaches achieve high throughputs with batching of multiple packets and process them more efficiently in the user space (e.g. preallocation, zero-copy). However, these approaches usually show the drawback of insufficient packet latency. To support low latency packet processing, Ueda et al. [11] introduce dedicated interrupt requests (IRQ) for the reception of real-time traffic. But this additional IRQ overhead leads to a strong decrease in the overall system performance (e.g. maximum throughput). Furthermore, Cummings and Tamir [12] propose a busy poll based concept for network interfaces which can be set by the application. Their approach is designed for end systems but it causes high CPU utilization and prevents the CPU from energy saving, even at low traffic loads.

In this paper, we present and evaluate a prototype implementation of our QoS concept for a Linux software router. As a proof of concept, the testbed measurements show that with our prototype the packet latency of latency-sensitive traffic remains very low, even under high traffic loads. This is accomplished without significant performance degradation, e.g. in terms of the achievable maximum throughput.

The remainder of the paper is structured as follows. First, the Linux packet processing is described in Section II. We present our QoS concept in Section III. In Section IV, we explain important aspects of our prototype implementation. In Section V, we evaluate our prototype based on testbed measurements. Finally, we summarize the paper in Section VI.

II. Linux-based Packet Processing

Before version 2.6, the Linux kernel followed an interrupt-driven approach for receiving network data, so that each received packet causes an IRQ. However, the throughput collapses with high offered loads due the high IRQ effort. This is known as the *receive livelock* state [13], in which the CPU is only utilized with IRQ handling and has no CPU cycles left for the actual packet processing or other processes. Therefore, Salim et al. [14] presented a new packet reception approach, the so-called NAPI (New API) which was introduced with Linux kernel version 2.6. Today, the NAPI is still a fundamental part of the Linux kernel network subsystem. The NAPI is a hybrid mechanism which combines the advantages of interrupt-driven and poll-driven approaches. In case of a low offered load, the system behaves like an IRQ-driven system whereat each packet causes an IRQ. Thus, the waiting-time of a packet is rather low which implies low packet latency. At high offered loads, the system rather behaves like a poll-driven system where IRQs are disabled and multiple packets are served in batches. Therefore, CPU cycles are mainly spent for the packet processing at high offered loads which maximizes the achievable throughput. This behavior of the NAPI is achieved as described in the following.

Each network device (aka. QVector, cf. Sec. IV) uses a dedicated IRQ line per CPU core. By default, there is a one-to-one relationship between a device and a packet reception queue (Rx ring). An IRQ which is generated by a device will cause the execution of the interrupt service routine (ISR). A NAPI-compliant driver performs the following tasks:

- The IRQ line of the device is disabled in order to ensure that no further IRQs are generated if packets arrive in the corresponding Rx ring. Nevertheless, received packets are still transferred into main memory via DMA.
- The corresponding device is enqueued in a so-called *poll list* and further packet processing is scheduled for later execution (by generating a so-called Soft-IRQ), so that the ISR quickly returns. Later, if the Soft-IRQ is handled, the poll list is served by the NAPI in FIFO manner. This means that packets are polled from the enqueued device and are passed to the IP stack for the actual packet processing (e.g. routing).

The polling of a device terminates due to one of the following reasons:

- All backlogged packets of the device have been processed. Then, the device is removed from the poll list and the IRQ line that corresponds to this device is re-enabled.
- A maximum number of packets that corresponds to the device budget (aka. poll size) has been processed but there are still further packets backlogged in the Rx ring of the device, which are waiting for being processed. Then, the IRQ line of this device remains disabled and the device entry is moved to the tail of the poll list again.

III. Conception of Low Latency Support

The NIC drivers and also the Linux NAPI [14], [15] do not support traffic classes. However, this is important for the differentiated treatment of QoS-sensitive traffic. Therefore, in our QoS concept the received packets are classified into traffic classes and accordingly directed into the dedicated waiting queues (Rx rings) of an device. Finally, the packet processing of these Rx rings with QoS-sensitive traffic can be prioritized by the corresponding CPU core. An example of such a low latency (LL) software router with the two traffic classes Real-Time (RT) and Best Effort (BE) is depicted in Fig. 1. In the following, our QoS concept is described in detail.

A. Traffic Classification

Modern network cards have various features to offload specific packet processing tasks from the CPU to the NIC. For example, the Intel Ethernet NIC controller X540 [16] supports multi-queuing (MQ) and receive-side scaling (RSS) to efficiently distribute incoming packets among the available CPU cores. Especially, the *Flow Director* is a specific NIC hardware filter which allows traffic classification based on MAC or IP header fields, TCP/UDP ports, VLAN tags and even flexible 2 Byte tuples in the first 64 Byte of a packet. Based on this information the NIC is able to classify received packets and sort them into multiple dedicated Rx rings which in turn can be assigned to specific CPU cores via the RSS feature. If a received packet does not match any of the NIC filter rules, then the NIC will direct it into the Rx ring of the traffic class with the lowest priority (e.g. BE). Thereafter, the packets are transferred by Direct Memory Access (DMA) into the main memory without any involvement of the CPU. As a consequence, an IRQ to the appropriate CPU core is signaled if the corresponding IRQ line of the device is enabled.

In our QoS concept, these NIC features are exploited by a packet processing system for traffic classification (Classifier). Each device provides a dedicated Rx ring per traffic class. A device uses only one IRQ line. Therefore, this IRQ line is shared between multiple Rx rings to save IRQ overhead and thus CPU cycles.

Deri et al. [17] also used NIC hardware filters to accelerate a traffic analysis framework by reducing the number of packets to the relevant ones only. Their case study showed that although the configuration may increase the complexity of a system, it can improve performance of CPU-based sampling approaches. Tanyingyong et al. described an OpenFlow switch where some matches were offloaded to the NIC [18]. In 2012, they also proposed a fast processing path for a router, where the routing decision for a limited number of flows (typically those with high packet rates) is offloaded to the NIC by using the Flow Director [19].

B. Traffic Prioritization

Well-known QoS mechanisms for differentiated packet treatment (e.g. DiffServ, IntServ) are only supported as a Qdisc on the outgoing network interface. On the ingoing network interface, multiple Rx rings (which are associated with the

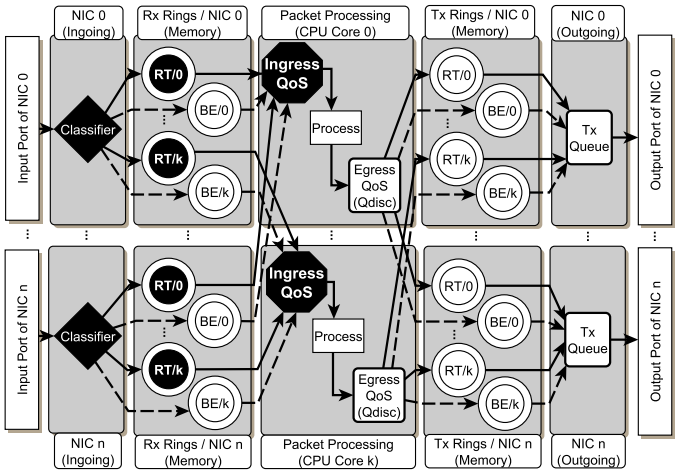


Fig. 1: Ingress and egress QoS support of a multi-core software router with dedicated Rx rings for RT and BE traffic

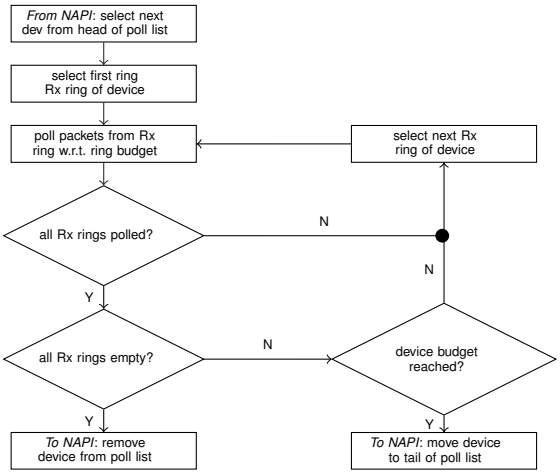


Fig. 2: Ingress QoS support

same CPU core) are simply served in a round-robin manner. So, traffic prioritization on the ingoing network interface is not supported. However, the Rx rings with QoS-sensitive traffic of the ingoing network interface should be preferred to support differentiated packet treatment. Thus, our QoS concept introduces *Ingress QoS*. By this way, a scheduling strategy (cf. Sec. III-C) is applied for the ingress traffic to prefer RT Rx rings w.r.t. the actual packet processing by the CPU core. Additionally, a Qdisc can be applied on the egress network interface to prioritize the RT traffic (Egress QoS). The extensions are highlighted in Fig. 1.

In the following our QoS concept, as illustrated in Fig. 2, is described in detail. Firstly, the Linux NAPI checks the poll list whether devices were added for packet processing. Multiple devices in the poll list are served in a round robin manner corresponding to the *device budget*. The device budget (usually 64 packets for GbE adapters) limits the number of packets which can be processed in a row by a device.

As mentioned in Section II, there is a one-to-one relationship between a network device and an Rx ring. Instead of a single Rx ring per device and per CPU core, our concept provides a dedicated Rx ring per traffic class. Consequently, a specific device uses a dedicated Rx ring per traffic class. The packet processing of such a device is controlled by the *device budget* and dedicated *ring budgets*. While the *device budget* still defines the number of packets that may be consecutively polled from the device, the *ring budget* specifies the number of packets per Rx ring that can be processed at maximum before switching to another Rx ring of the same device. In our concept, the *ring budget* is dependent on the scheduling strategy (cf. Section III-C) and can be adapted according to the priority of the associated traffic class.

Finally, it is checked whether all Rx rings are empty. If this is true, then the NAPI removes this device entry from the poll list and the IRQ line is re-enabled. Otherwise, if any of the Rx rings still contains packets, it is checked whether the device budget was exhausted. If the device budget is not exhausted, all Rx rings are served again. Otherwise, if the device still has packets to be processed then the NAPI moves the device to the tail of the poll list for later processing.

In principle, our QoS concept supports an arbitrary number of traffic classes. Through the differentiated packet treatment at the ingoing network interface, we expect that the latency of latency-sensitive applications is significantly reduced. In Section V, the effects of our QoS concept with two traffic classes (RT, BE) are evaluated based on real testbed measurements.

C. Scheduling Strategies

In this paper, we consider the following scheduling strategies for our ingress QoS concept (cf. Table I).

- *Single Queue (SQ)*: The incoming packets are not classified but directed to one Rx ring per device. Thus, all packets are served in a FIFO manner which represents the state of the art without traffic classification and priorization.
- *Round-Robin (RR)*: The incoming packets are classified by the NIC and directed into the Rx ring of a dedicated device. This represents the state of the art with NIC-based traffic classification but without traffic prioritization.
- *Low Latency Round Robin (LL-RR)*: The incoming packets are classified by the NIC into dedicated Rx rings per device. Each device has a dedicated Rx ring per traffic class. All Rx rings have the same ring budget, thus, this represents our QoS concept without prioritization.
- *Low Latency Weighted Fair Queuing (LL-WFQ)*: Similar to LL-RR, but the budget of an Rx ring corresponds to the traffic class priority.

TABLE I: Scheduling strategies

	SQ	RR	LL-RR	LL-WFQ
Traffic classification	×	✓	✓	✓
Shared IRQ	×	×	✓	✓
Traffic prioritization	×	×	×	✓

D. Theoretical Considerations

In this section we estimate the worst case latency of our QoS concept by means of an elementary queueing model. The model consists of a server (CPU core) which processes two queues in an alternating manner (namely, the RT ring and the BE ring) as illustrated in Fig. 3. A specific ring budget exists for serving each of both queues. After reaching this ring budget the server passes over to the other queue, if the current queue has not been emptied already before reaching the ring budget. Anyway, we assume that the ring budget is only relevant for the BE ring because the ring budget to serve the RT ring is chosen sufficiently large to make sure that the RT ring is always empty when the server switches to serving of the BE ring. So, the ring budget of the RT ring is never exhausted. In the following, we use the variables depicted in Table II.

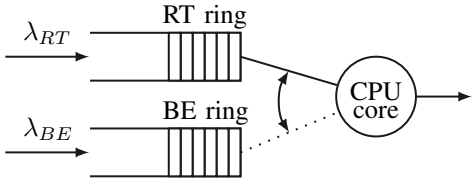


Fig. 3: Queueing model of ingress QoS

TABLE II: Used variables and relations

b	constant BE ring budget	$b \geq 1\ ,\ b \in \mathbb{N}$
x	constant packet service time	$x > 0\ ,\ x \in \mathbb{R}$
r	ratio of RT packets w.r.t. all packets	$0 \leq r \leq 1$
λ	total packet arrival rate	$\lambda = \lambda_{RT} + \lambda_{BE}$
λ_{RT}	RT packet arrival rate	$\lambda_{RT} = r\lambda$
λ_{BE}	BE packet arrival rate	$\lambda_{BE} = (1-r)\lambda$
ρ_{RT}	system utilization by RT packets	$\rho_{RT} = x\lambda_{RT}$

Now, we can determine a (slightly conservative) upper bound for the latency of RT packets. As we assume the RT ring to be empty when the CPU core switches to BE ring, the latency of a packet in the RT ring is bounded by the sum of the following times:

- The waiting time for the CPU core to return after serving the BE ring within one service cycle using its complete ring budget b corresponds to bx.
- The time needed to serve the backlog of packets in the RT ring, which accumulated during the time the CPU core spent at for serving BE ring; this first backlog consists of at $bx\lambda_{RT}$ packets in mean, leading to a mean service time requirement of a first backlog of $bx\lambda_{RT}x$.
- The time needed to serve the second backlog of packets in the RT ring, which accumulated during the time the CPU core spent at for serving the first backlog; the service time requirement of a second backlog of $bx(\lambda_{RT}x)^2$.

So we can sum all single portions of delays to get the desired bound T_{max} for the packet latency in RT ring:

$$T_{max} = bx + \sum_{i=1}^{\infty} bx(\lambda_{RT}x)^i = bx \sum_{i=0}^{\infty} \rho_{RT}^i = \frac{bx}{1-\rho_{RT}} \quad (1)$$

If we assume the system is not overloaded by RT traffic, then this geometric series converges because $\rho_{RT} < 1$. Evidently, the first term of the sum in the middle of Eq. (1), i.e. bx, represents the maximum waiting time for the CPU core to return to serving the RT ring after a complete service cycle of the BE ring. The second term of the sum, namely the geometric series, represents the maximum duration of a complete service cycle of the RT ring, cf. similarity of this result to the expected length of the busy period in M/M/1 queuing systems [20].

It should be noted that the important assumption used to derive Eq. (1), namely that, in an interval of length T, we can expect that $T\lambda_{RT}$ packets will arrive at the RT ring, may not be fulfilled for small values of T. Nevertheless, if the traffic arriving at the RT ring is rather smooth (e.g., no large bursts exist) our assumption should be sufficiently valid. Now, Eq. (1) also allows us to determine an appropriate value for the budget b if, e.g., we want to bound the maximum latency in the RT ring by a value T^* which is a multiple m of x, i.e. $T^* = mx, m \in \mathbb{N}$. This requirement is fulfilled if the following equation holds.

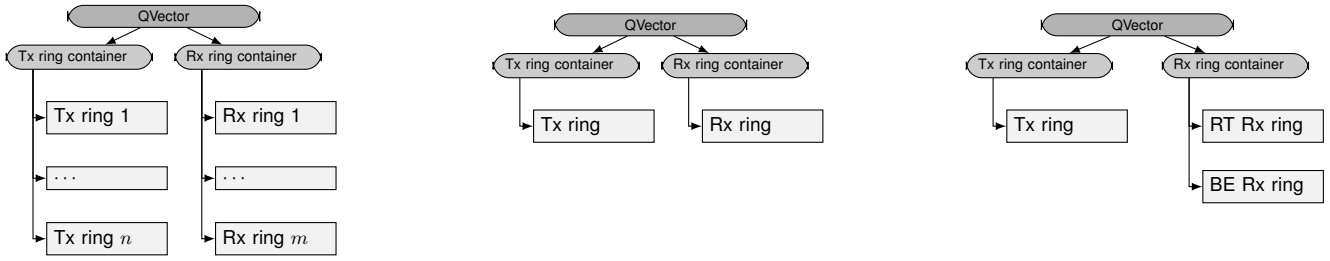
$$\frac{bx}{1-\rho_{RT}} \leq T^* = mx \quad \Leftrightarrow \quad b \leq m(1-\rho_{RT}) \quad (2)$$

As a specific example, let us determine an acceptable value for b if $m = 6$ and $\rho_{RT} \leq 0.3$. In this situation, $b \leq 6 \cdot 0.7 = 4.2$ holds and this means that we could choose $b = 4$, because we want to have b as large as possible in order to minimize the switching overhead between the RT ring and the BE ring. Analogously, for $\rho_{RT} \leq 0.5$ (and still $m = 6$) we could choose $b = 3$. Determining the value of the ring budget to serve the RT ring is trivial, because it is sufficient to take a value which is large enough so that the budget is nearly never reached in order to completely serve the RT ring in an RT service cycle.

IV. Implementation of Low Latency Support

In this section we give an overview of the modifications that we applied to the Linux driver module of a 10 GbE adapter in order to realize our prototype implementation. Since our testbed is equipped with 10 GbE NICs from Intel, we implemented the proof of concept prototype for the corresponding *ixgbe* driver (version 3.22.3).

Within the ixgbe driver, Intel provides a data structure which is wrapped into the NAPI device structure to be compatible to the NAPI. This data structure is called *QVector* (short for Queue Vector) and it is responsible to store information about Rx rings. Fig. 4(a) shows a schematic view of how such a QVector is structured. Each QVector holds two distinct containers. One container for Tx rings and another one for Rx rings. The main purpose for this data structure is to save IRQs, as all referenced rings of a QVector share the same IRQ line. For instance, this is exploited by a technique called virtual machine device queues (VMDq) where the NIC sorts packets into specific receive queues (Rx rings) which are then grouped by a QVector and thus share an IRQ line. Therefore, a single IRQ causes the virtual machine monitor (VMM) to handle



(a) General structure (b) Default RSS setups (c) Our ingress QoS concept

Fig. 4: Schematic view of Intel's Queue Vector (QVector) data structure

multiple Rx rings (for different VMs) in batch and provide the VMs with bulks of packets to improve I/O performance. A standard RSS setup without VMs also benefits from IRQ mitigation due to QVectors. For each NIC and for each CPU core a dedicated QVector mitigates Rx and Tx IRQs while it also avoids locking in case of parallel processing. Compared to the general VMDq case with multiple Rx and Tx rings per ring container (i.e. as many as VMs are hosted), each of both ring containers now references exactly one ring (cf. Fig. 4(b)).

The purpose of our modifications is to extend the ixgbe driver to provide QVectors within RSS setups that has two Rx rings, as illustrated in Fig 4(c), since our QoS concept (cf. Sec. III) requires one Rx ring for RT traffic and another one for BE traffic. In the following, we describe the extensions that we applied to the standard ixgbe driver in order to realize our QoS concept prototype implementation (cf. Section III).

A. Grouping of Multiple Rx Rings

The first part of our driver modification refers to group two Rx rings per QVector (cf. Fig. 4(c)). We extended the driver to provide a new module parameter which we named *per ring buffer* (PRB). PRB accepts an arbitrary long array of integers (cf. Listing 1), whereby each integer causes the driver to add an Rx ring to each created QVector. Thus, an arbitrary number of traffic classes is supported by the QVectors. The PRB values refer to the ring budgets (cf. Sec. III-B).

B. Polling with Ring Budgets

Our second driver modification refers to the poll function `ixgbe_poll` which is called by the NAPI and which is responsible to fetch the queued packets from the Rx rings and pass them to the IP stack, one by one. For this purpose, the NAPI provides `ixgbe_poll` with a reference device from the poll list. Based on this device, `ixgbe_poll` determines the associated QVector and starts to free packets buffers of already sent packets that reside in the Tx rings within Tx ring container. Afterwards, the poll function will process the packets from the Rx rings. For VMDq, the standard implementation of `ixgbe_poll` will equally distribute the device budget between all Rx rings that remain in the Rx ring container. For example, if the device budget is 64 (default) and if there are two Rx rings, then, up to 32 packets from the first Rx ring are processed before up to 32 packets from the second Rx ring are processed. Afterwards, the driver returns to the NAPI which switches to the next device. However, this behavior is not suitable for QoS-sensitive packet processing because an RT poll might be interrupted for up to $32 \cdot 0.6\,\mu s \approx 19\,\mu s$ if we assume a packet service $x = \frac{1}{1.75\,Mpps} = 0.6\,\mu s$ according to the maximum throughput (cf. Sec. V-B1). Therefore, we propose to apply smaller per ring budgets for the BE Rx ring in order to improve the latency of RT traffic (cf. Sec. III-D). Additionally, we also want to exploit the device budget of 64 due to efficient usage of the CPU resource. For this purpose, we extended `ixgbe_poll` by an additional loop that allows both Rx rings to be polled alternately with small ring budgets until the poll size is exhausted (cf. Fig. 2).

C. Usage of Modified Driver

Listing 1 shows an example for loading our modified driver[1]. The PRB parameter defines two ring budgets, thus, the QVector groups two Rx rings. The order of the PRB values from left to right refers to the Rx rings with increasing traffic class priority. Therefore, the ring budget of the BE Rx ring refers to 4, whereas the RT Rx ring gets a ring budget of 60. Furthermore, the driver arranges one QVector for our ingress NIC (RSS=1) and the interrupt throttling is disabled (ITR=0).

```
root@DuT ~/# modprobe ixgbe PRB=4,60 RSS=1 ITR=0
```

Listing 1: Loading of `ixgbe` driver with the PRB parameter

V. Evaluation

Our goal is to investigate whether our QoS concept has positive effects on the latency of real-time traffic and whether the throughput, which is potentially decreased due to the overhead of our implementation, is still acceptable for practical usage. Thus, we measure and evaluate the performance of our ingress QoS driver (LL-RR, LL-WFQ) and compare it to the performance of the state-of-the-art driver (SQ, RR). The measurements were conducted in our testbed wherein we already performed various other performance tests [21], [22].

A. Measurement Setup

1) Hardware Configuration: The device under test (DuT) which serves as the software router is equipped with a SuperMicro X9SCL/X9SCM motherboard, a 3.20 GHz Intel Xeon CPU E31230, 16 GB RAM, and an Intel X540-T2 NIC.

[1]publicly available at: http://www.informatik.uni-hamburg.de/memphis

All measurements were performed with Linux kernel version 3.16.7 and ixgbe version 3.22.3. We deactivated features like Intel Turbo Boost and Hypertheading as they introduce unpredictable behavior. Additionally, we configured the CPU to run at a fix rate of 3.20 GHz. Furthermore, we deactivated the interrupt throttling (ITR) which would increase the packet latency. Ethernet flow control was disabled to conduct meaningful measurements even in overload situations.

2) Software Configuration: For comparison, our set of measurements was performed on four differently configured DuTs (cf. Section III-C). For all configurations the Rx ring size is 512 (default) and the device budget is 64 (default).

In our previous work, we showed that the maximum throughput of the packet processing scales nearly linearly with the number of CPU cores [23]. Thus, we simplified the measurements and configured the DuT to utilize only one CPU core for Linux IP packet processing.

The first two scenarios (SQ, RR) represent the state-of-the-art case and are conducted with the original ixgbe driver. With the SQ scheduling strategy, all traffic (RT, BE) is sorted into the same Rx ring. With RR, RT and BE traffic is distributed to the corresponding Rx rings of two different devices. These devices are served according to the NAPI, which is some kind of round robin polling (cf. Section III).

The other scenarios (LL-RR, LL-WFQ) are performed with our modified ixgbe driver. For LL-RR we configured the ring budget to be 4 (as deduced in Section III-D) for both Rx rings. Thus, the CPU is equally shared between the traffic classes if both rings are sufficiently utilized. We assume that the offered load to the RT ring is at most 30 % of the maximum throughput. Therefore, we expect in case of 4 arrived BE packets $4 \cdot 0.3/(1-0.3) \approx 1.7$ RT packets in mean. Thus, an RT ring budget of 4 is slightly oversized which has the advantage that backlogs, due to small bursts (e.g. if the short-period real-time percentage is above 30%), decrease faster.

With LL-WFQ the RT ring budget is 60 while the BE ring budget remains 4. Thus, the RT ring budget is sufficiently large (as assumed in Section III-D) but not that large that the poll size of 64 is exceeded. Otherwise the BE traffic potentially might starve in overload situations.

3) Methodology: Although, we are preliminary interested in throughput and latency, we also recorded DuT internal meters like CPU utilization and IRQ rates, that help to explain several performance related effects. The IRQ counts and the CPU utilization of the DuT are obtained via the process file-system (/proc/interrupts) and *perf* (a performance evaluation infrastructure/tool for Linux), respectively. Throughput and latency is sampled by our packet generator MoonGen [21], which is packet source and sink at the same time. The DuT and the packet generator are directly connected.

4) Network Load: We measured the throughput for one CPU core of the DuT at different packet rates ranging from 0.05 Mpps to 2.0 Mpps in steps of 0.05 Mpps and different real-time percentages of the overall traffic ranging from 0% to 100% in steps of 5%. Each measurement took 60 s during which test traffic based on a Poisson traffic was applied.

In our previous work [23], we showed that the CPU constitutes the bottleneck in a software router for small packet sizes. Hence, we prevent that Ethernet links become the bottleneck by limiting the packet size to 128 Byte (10 GbE links are theoretically able to cope with approx. 8.45 Mpps of this size).

B. Measurement Results

1) Throughput: Firstly, we analyzed the maximum throughput achieved by one CPU core of the DuT for all scheduling strategies, in order to compare whether the throughput of the RR and LL strategies deviate too much from SQ which would be unsuitable for the practical usage. Therefore, for each strategy and for each RT ratio, we determined the maximum offered load that is achieved. This is done by stepwise increasing the offered load until packet loss is encountered. Since we are not interested in a very fine-grained rendering of the maximum throughput, we chose a step size of 0.05 Mpps in order to reduce the efforts for measurements.

Fig. 5(a) illustrates the maximum throughput of the different strategies at RT ratios from 0% to 100%. SQ is our baseline and achieves nearly a constant maximum throughput of about 1.75 Mpps, as the RT and BE traffic share the same Rx ring, whereby the throughput is independent from the RT ratio. Additionally, it is to mention that SQ reaches the highest maximum throughput we observe for RT ratios from 0% to 85%. From 90% to 100% it might be suggested that LL-WFQ reaches a much higher throughput than SQ. However, this is an artifact that results from the coarse step size. In contrast to that, RR has the worst maximum throughput which is between 1.6 Mpps and 1.65 Mpps. LL-RR and LL-WFQ show a slightly worse behavior than SQ but are still better than RR and also reach a high maximum throughput of at least 1.65 Mpps.

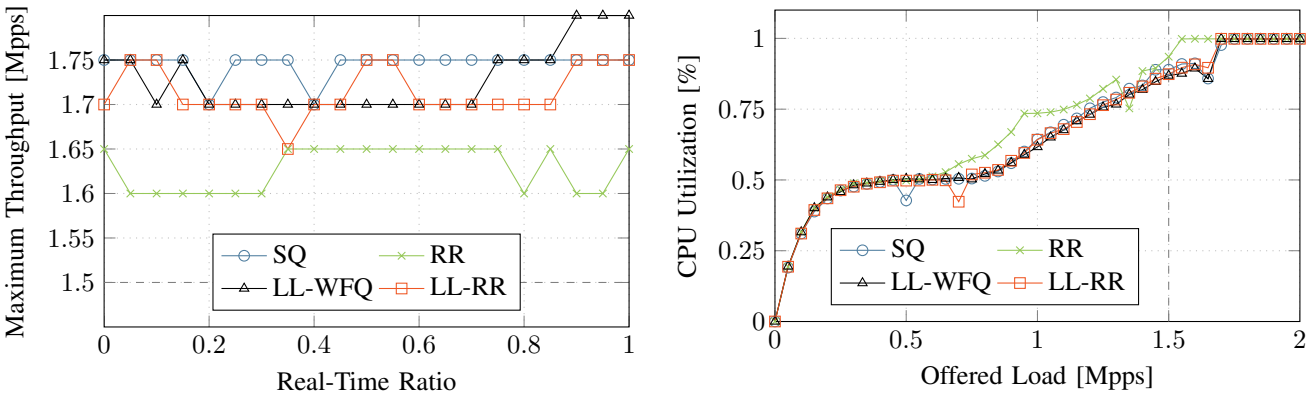
In summary, we observe that all strategies cope with an offered load of 1.5 Mpps. Hence, a software router, which is equipped with 10 CPU cores, will be able to satisfy the line rate of a 10 GbE adapter (14.8 Mpps for 64 Byte packets).

In order to be able to equitably compare all strategies, we further focus on two specific measurements. (1) We evaluate measurements at an offered load of 1.5 Mpps in experiments where the offered load is fixed. (2) We investigate measurements with a fixed RT ratio of 30%, since we assume that todays RT traffic in the Internet is 30% at most.

2) CPU Utilization: Fig. 5(b) illustrates the CPU utilization of the DuT for offered loads, ranging from 0.05 Mpps to 2.0 Mpps at a fixed RT ratio of 30% in steps of 0.05 Mpps.

We observe that LL-RR, and LL-WFQ basically exhibit the same behavior as SQ. Between offered loads of 0.05 Mpps and 0.2 Mpps we see a steep increase in the CPU utilization, which is caused by IRQ handling. Then, from 0.2 Mpps to 0.7 Mpps the CPU utilization is nearly constant, as the IRQ rate gets throttled by the NAPI. Afterwards, the CPU utilization starts to increase linearly with growing offered load until 100% is reached at approx. 1.7 Mpps.

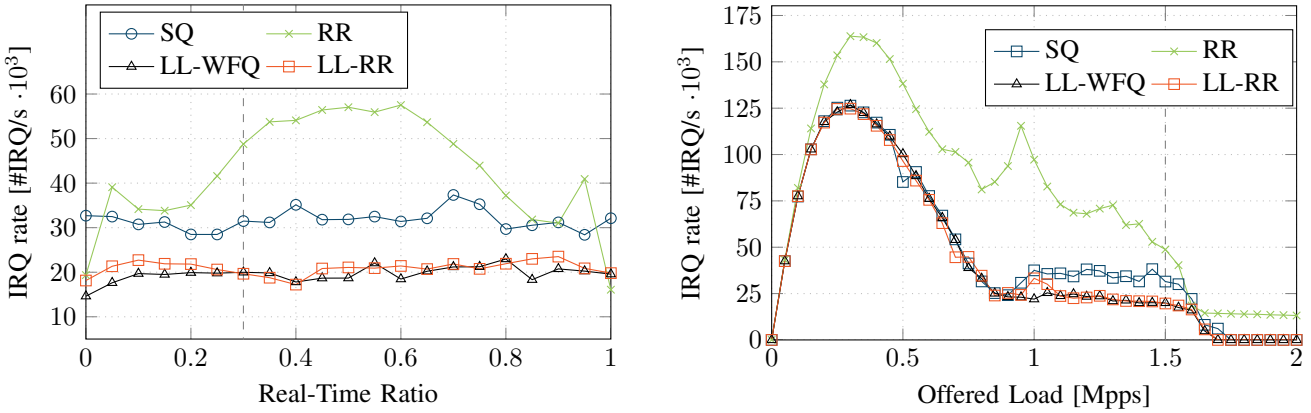
Compared to all other strategies RR behaves differently, since RR reveals an increased CPU utilization at offered loads above 0.7 Mpps. Interestingly, RR reaches 100% CPU



(a) Maximum Throughput

(b) CPU utilization at an real-time ratio of 30%

Fig. 5: Maximum throughput and CPU utilization for different scheduling strategies



(a) Interrupt rate at an offered load of 1.5 Mpps

(b) Interrupt rate at an real-time ratio of 30%

Fig. 6: Interrupt rate for different scheduling strategies

utilization earlier than the other strategies (i.e. at 1.55 Mpps), which explains the decreased throughput we observed before. Hence, for an offered load of 1.5 Mpps, which we previously chose for experiments with fixed offered loads, we observe that all strategies lead to nearly full CPU utilization but not to overload, which is a reasonable and fair operating point to compare the QoS characteristics of all strategies.

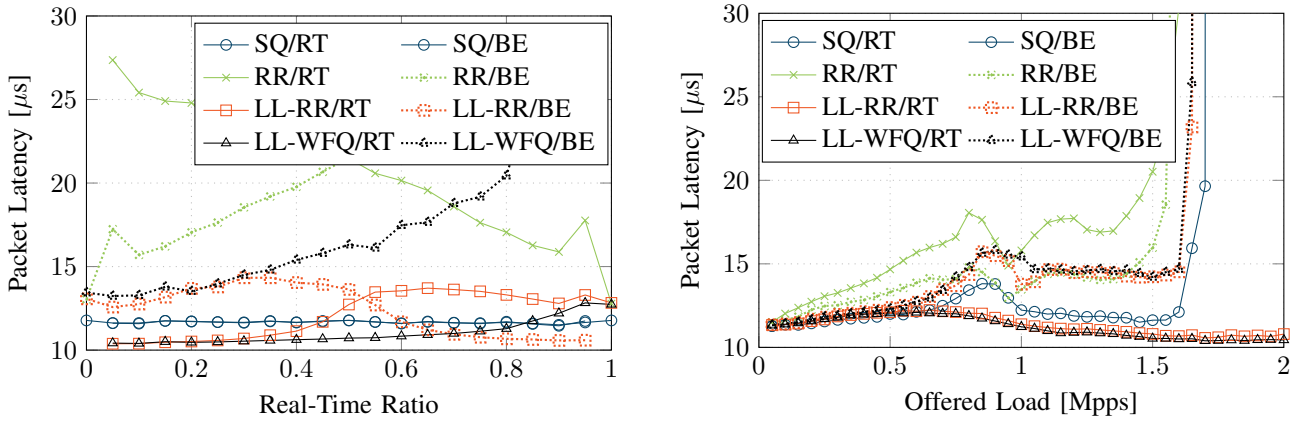
3) IRQ Rate: Fig. 6(a) illustrates the IRQ rate of the DuT for different RT ratios, at a fixed offered load of 1.5 Mpps. The IRQ rate is averaged over the 60 s interval.

For SQ we observe a nearly constant IRQ rate for all RT ratios, which is approx. 32k IRQ/s. Compared to that, we see a behavior of RR that is dependent from the RT ratio. The IRQ rate of RR starts with approx. 20k IRQ/s and increases to approx. 58k IRQ/s until a RT ratio of approx. 50% is reached. Afterwards, the IRQ rate decreases. This effect is a consequence of the two separate devices that are used by this strategy (cf. Section III-C), whereby both devices generate IRQs independently but according to the offered load. Thus, the RT device generates as much IRQs at an RT ratio of 30% as the BE ring generates at an RT ratio of 70%. As the IRQ rates of both devices sum up, we observe a symmetric IRQ rate which is in worst case (at 50% RT ratio) approx. two times higher than the IRQ rate of SQ. For LL-RR and LL-WFQ, we observe roughly the same behavior of the IRQ rate, which is nearly constant at approx. 20k IRQ/s. Interestingly, the IRQ rate of both LL strategies is 10k IRQ/s lower than the IRQ rate of SQ, which corresponds to a decrease of approx. 30%.

Fig. 6(b) illustrates the IRQ rate for offered loads, ranging from 0.05 Mpps to 2.0 Mpps, at a fixed RT ratio of 30%.

For SQ we see an interesting effect at offered load from 0.8 Mpps to 1.0 Mpps. First, the IRQ rate drops to 25k IRQ/s before it increases to 32k IRQ/s where it remains until an offered load of approx. 1.6 Mpps is reached. At an offered load of 1.7 Mpps the IRQ rate drops to zero, as the the NAPI prevents from the reactivation of the IRQ line.

For RR we observe that the IRQ rate is generally higher than for SQ at all offered loads larger than 0.15 Mpps, which is again due to the two devices that are used. The more mentionable effect is, that RR still generates approx. 17,000 IRQ/s in case of overload (1.7 Mpps to 2.0 Mpps). These IRQs are generated by the RT device, as RR allows to spend



(a) Paket latency at an offered load of of 1.5 Mpps

(b) Packet latency at an real-time ratio of 30 %

Fig. 7: Packet latency for different scheduling strategies

50% of CPU resource for RT packet processing. Therefore, RR is theoretically able to reliably process an RT load of $1.6\,\text{Mpps} \cdot 0.5 = 0.8\,\text{Mpps}$ which is above the actual RT load at an offered load of 2.0 Mpps with 30% RT traffic which is $2.0\,\text{Mpps} \cdot 0.3 = 0.6\,\text{Mpps}$. Hence, it is very likely that the IRQ line of the RT device is often re-enabled since the device budget is not exhausted.

For both LL strategies we observe the same behavior of the IRQ rate with an exception at an offered load of 1.0 Mpps where LL-RR has a higher IRQ rate than LL-WFQ. LL-RR and LL-WFQ only differs in the budget of their RT Rx rings (cf. Section III-C). Thus, we conclude that in case of LL-RR the low per ring budget of both rings ensures a fair distribution of the device budget, which helps to avoid backlogs in any of both rings in case of bursts, as caused by the Poisson traffic pattern. LL-WFQ has a much higher RT Rx ring budget and in turn the probability for BE backlogs is high if RT bursts arrive. Thus, it is plausible that the IRQ line is less often re-enabled with LL-WFQ. As SQ does not differentiate between RT and BE, the IRQ re-enabling is less complicated and SQ reaches higher IRQ rates at offered loads larger than 1.05 Mpps.

4) Latency: As latency optimization is the major objective of our QoS concept we discuss our latency related measurement results in-depth. In Fig. 7(a) the average packet latency is plotted against the RT ratio at a fixed offered load of 1.5 Mpps.

For SQ, we see that the latency is constant at approx. 11.7 μs and therefore independent of the RT ratio, since a traffic classification is missing. Thus, each packet experiences the same mean waiting time in the Rx ring before it is processed.

Obviously, the RR strategy has poor real-time properties, as all measured latencies for both traffic classes are above that of SQ, which is again due to the numerous IRQs (cf. Fig. 6(a)). Also, we observe that the RT latency is above the BE latency at RT ratios below 50%, as the RT Rx ring will suffer from the long poll phase as caused by the highly utilized BE Rx rings.

Compared to RR, the LL-RR implementation behaves similar regarding the effect that RT and BE latencies cut across at an RT ratio of 50%. However, as the per ring budget is relatively low (i.e. 4 for both traffic classes), even highly utilized BE Rx rings will not interrupt RT Rx rings for longer than the processing of 4 packets. Thus, we observe a low RT latency of approx. 10.4 μs, which is approx. 11% better than the latency of SQ (11.7 μs). However, this improvement comes on cost of the BE latency, which is notably higher than for SQ at RT ratios below 50%.

For LL-WFQ we observe nearly the same low latency of RT traffic as for LL-RR (10.4 μs) but LL-WFQ is able to provide lower RT latencies than SQ for RT ratios between 50% and 80%. This is due to of the high per ring budget of 60 for the RT ring compared to the BE ring budget of 4. Another effect is that both LL strategies suffer from higher latencies (compared to SQ) in the extreme cases where we load the DuT with 0% or 100% RT traffic, respectively. We refer this increase of latency to the overhead that is introduced by our modifications.

Fig. 7(b) illustrates the packet latency at different offered loads, ranging from 0.05 Mpps to 2.0 Mpps, at the specific RT ratio of 30%. Regarding SQ we see two interesting effects. The first one is, that the latency increases significantly between 0.75 Mpps and 1.0 Mpps. We refer this effect to the NAPI, as we observe IRQ throttling within this range of offered load (cf. Fig. 6(b)). Special attention should be drawn to the second effect, which is that the latency of SQ will abruptly increase at an offered load of 1.65 Mpps as bursts cause backlogs in the Rx ring and thus increase the waiting time of packets.

RR shows higher RT than BE latency at all offered loads, as RT packets are not prioritized and therefore suffer from long waiting times. Thus, we conclude that RR is not acceptable for latency-sensitive applications.

Regarding LL-RR and LL-WFQ we observe that RT latencies are quite similar at all offered loads. Thus, we conclude that a BE and RT ring budget of 4 is sufficient for RT ratios of 30% and less. For LL-RR and LL-WFQ we also see that the RT latency increases between 0.05 Mpps and 0.75 Mpps while it decreases between 0.75 Mpps and 1.7 Mpps. This behavior

might not be obvious on the first sight, but due to a low and stable IRQ rate (cf. Fig. 6(b)) we conclude that the NAPI is mostly in polling mode. Thus, the packet latency is not that much impaired by the time it takes to handle an IRQ. Instead, the packet processing basically behaves as explained in Section III-D, where we stated that new RT packets might arrive while others are processed. Such packets will have small waiting times, thus, the mean latency is low at high offered loads and even in overload situations.

In contrast, we observe that the BE latency for LL-RR and LL-WFQ is notably higher than for RT and like for SQ it also increases significantly in case of overload. However, for LL the RT latency remains low, even in case of overload. This improvement of the RT latency comes on the cost of the BE latency, but we argue that BE traffic has no real-time requirements and high delays are acceptable.

In conclusion, we found that RR has no benefits compared to SQ. More IRQs burden the CPU and as a consequence less packets are processed whereby the throughput decreases (1.6 Mpps). Additionally, RR provides poor RT latency, as highly utilized BE Rx rings in combination with large ring budgets cause long waiting times for RT packets. In contrast, our LL driver exploits separate ring budgets to provide low RT latencies (approx. 11.4 μs), even in case of overload. Since the design of our LL driver minimizes additional IRQ and packet reception overhead, it achieves almost the same throughput as SQ (1.7 Mpps), which is the upper limit that is given by the CPU speed and the networking module of Linux itself.

VI. Summary

Over the last years we face the trend of growing real-time traffic which has to share the limited resources of the Internet with other traffic classes. Thus, we see the demand for mechanisms that efficiently allocate these resources to these traffic classes in order to provide QoS. While QoS is not a new topic, all previous concepts assume the outgoing link to be the bottleneck. However, seeing the trend towards flexible CPU based data plane devices where general purpose hardware in combination with software serves arbitrary needs, we argue that more routers will be CPU bounded in future.

As traffic prioritization behind the bottleneck (i.e. CPU) introduces avoidable latency, we proposed a new QoS concept for software routers that prioritizes traffic *before* being served by the CPU (ingress QoS). Based on our concept we extend the driver code of a common 10GbE NIC. Afterwards, we conducted extensive real-world measurements to compare the performance of our new QoS concept with the state-of-the-art. An in-depth analysis regarding throughput and latency showed that our QoS concept improves the latency of real-time traffic while the throughput is nearly unaffected. These satisfying and meaningful results demonstrate that software routers are able to cope with real-time traffic, even at high offered loads.

Acknowledgment

This research has been supported by the German Research Foundation (DFG) as part of the *MEMPHIS* project. We thank our colleagues Florian Wohlfart and Sebastian Gallenmüller for their valuable feedback.

References

[1] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism To Scale Software Routers," in *ACM Symposium on Operating Systems Principles (SOSP)*, October 2009.

[2] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward Predictable Performance in Software Packet-Processing Platforms," in *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2012.

[3] R. Bolla and R. Bruschi, "PC-based Software Routers: High Performance and Application Service Support," in *ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, August 2008, pp. 27–32.

[4] S. Han, K. Jang, K. Park, and S. Moon, "Building a Single-Box 100 Gbps Software Router," in *IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)*, May 2010, pp. 1–4.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," *RFC 2475*, 1998.

[6] J. Wroclawski, "The Use of RSVP with IETF Integrated Services," *RFC 2210*, 1997.

[7] T. M. Runge, D. Raumer, F. Wohlfart, B. E. Wolfinger, and G. Carle, "Towards Low Latency Software Routers," *Journal of Networks*, vol. 10, no. 4, pp. 188–200, 2015.

[8] T. M. Runge, A. Beifuß, and B. E. Wolfinger, "Low Latency Network Traffic Processing with Commodity Hardware," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, July 2015.

[9] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," in *USENIX Annual Technical Conference*, April 2012.

[10] *Data Plane Development Kit: Programmer's Guide, Rev. 6*, Intel Corporation, January 2014.

[11] K. Ueda, T. Kikutani, and T. Yakoh, "Parallel Implementation of Real-Time Communication and IP Communication by Using Multiple Ring Buffers," in *IEEE Workshop on Factory Communication Systems (WFCS)*, May 2014, pp. 1–8.

[12] J. Cummings and E. Tamir, "Open Source Kernel Enhancements for Low Latency Sockets Using Busy Poll," http://www.intel.de/content/www/de/de/ethernet-controllers/open-source-kernel-enhancements-paper.html, 2013, Intel Corporation.

[13] J. C. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driven Kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[14] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond Softnet," in *Annual Linux Showcase & Conference*, vol. 5, 2001, pp. 18–18.

[15] J. H. Salim, "When NAPI Comes to Town," in *Linux Conference*, 2005.

[16] *Intel Ethernet Controller X540 Datasheet Rev. 2.7*, Intel Corporation, March 2014.

[17] L. Deri, J. Gasparakis, and F. Fusco, "Wire-Speed Hardware Assisted Traffic Filtering with Mainstream Adapters," in *NEMA*, 2010.

[18] V. Tanyingyong, M. Hidell, and P. Sjodin, "Using Hardware Classification to Improve PC-based OpenFlow Switching," in *International Conference on High Performance Switching and Routing (HPSR)*, July 2011, pp. 215–221.

[19] ——, "Improving Performance in a Combined Router/Server," in *International Conference on High Performance Switching and Routing (HPSR)*, June 2012, pp. 52–58.

[20] J. Sztrik, "Basic Queueing Theory," *University of Debrecen, Faculty of Informatics*, vol. 193, 2012.

[21] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference*, 2015.

[22] 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 *International Conference on Networked Systems (NetSys)*, March 2015.

[23] T. Meyer, F. Wohlfart, D. Raumer, B. E. Wolfinger, and G. Carle, "Validated Model-Based Performance Prediction of Multi-Core Software Routers," *Praxis der Informationsverarbeitung und Kommunikation (PIK)*, vol. 37, no. 2, pp. 93–107, 2014.