Low Latency Network Traffic Processing with Commodity Hardware

Torsten M. Runge, Alexander Beifuß and Bernd E. Wolfinger

Universität Hamburg, Department of Computer Science, Telecommunications and Computer Networks
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany
{runge|beifuss|wolfinger}@informatik.uni-hamburg.de

***Abstract*—Packet processing on commodity hardware is a cost-efficient and flexible alternative to specialized networking hardware. In case of Linux, the classical QoS mechanisms (e.g DiffServ) assume that the outgoing link is the bottleneck. However, on commodity hardware the CPU typically becomes the bottleneck in packet processing. Taking into account current trends in the Internet, we assume that the percentage of latency-sensitive applications (e.g. VoIP, video conferencing, online gaming) will increase. Thus, we propose to extend the Linux NAPI with respect to preferring latency-sensitive traffic at the ingress before reaching the CPU. In this paper, we investigate a new concept for the Linux NAPI for low latency packet processing. Based on a model of a Linux software router, we show in a case study that our concept strongly improves the packet latency of real-time packets at acceptable low costs regarding the achievable maximum throughput. Our model is calibrated and validated based on real testbed measurements (e.g. profiling).**

***Keywords*— commodity hardware; packet processing; resource contention; simulation model; ns-3; Linux NAPI; low latency**

I. Introduction and Related Work

Packet processing with specialized networking hardware like hardware routers has been the state of the art since many years. Today cost-efficient commodity PC hardware has profited from many improvements (e.g. multi-core CPUs, multi-queue NICs, NIC offloading, DCA, DMA, PCIe) to exploit parallelism in the packet processing with software. The packet processing software like the NIC drivers and the operating systems (OS) also received several enhancements like interrupt moderation for saving CPU cycles in high-load situations. Besides, optimized networking frameworks like netmap [1] or Intel DPDK [2] were proposed to fully replace the traditional Linux networking stack. These so-called software routers are able to cope with dedicated networking hardware in many cases. In contrast to hardware routers, software routers are more cost-efficient and more easy to extend which allows for fast adaption of new features. However, in case of high-speed packet processing hardware routers are more energy-efficient and powerful.

Many researchers have shown that the CPU cores constitute the bottleneck in the packet processing with commodity PC hardware [3]–[5]. The usage of multi-queue NICs with multiple Rx and Tx rings is essential to avoid locking with multi-core CPUs. The RouteBricks project [6] investigated the software router performance for different workloads (e.g. IP routing, IPSec). In case of IP routing, RouteBricks achieved a maximum loss-free forwarding rate (MLFR) of 6.35 Gbps respectively 12.40 Mpps where the latency was estimated as 24 μs for 64 B packets [4]. However, there is only limited support for differentiated packet treatment in front of the CPU bottleneck. For instance, Linux only supports ingress packet filtering but no class-based traffic differentiation. Traditional approaches like DiffServ [7] and IntServ [8] are only applicable as queuing disciplines (qdisc) at the egress NIC because the outgoing link is assumed to be the bottleneck. This shortcoming has serious impacts on latency-sensitive applications (e.g. VoIP, video conferencing, online gaming). Thus, we propose that packet processing systems based on commodity hardware should prioritize latency-sensitive traffic.

Analytical modeling and simulations are cost-efficient methodologies to evaluate new concepts. However, many network simulators such as ns-3 [9] lack of the modeling of the intra-node latencies. Chertov et al. [10] proposed a device-independent router model for different router types (e.g. Cisco 7602) that only requires black box measurements of the modeled router. Kristiansen et al. [11] introduced a service execution model to consider the time for multi-threaded packet processing software. We too proposed a general modeling approach for a more realistic node model of a packet processing system based on multi-core CPUs and other system internal components (e.g. NIC, memory) [12]. Besides, we implemented this approach as the ns-3 'resource-management' module which we apply in this paper to conduct the case study.

In this paper, we evaluate the packet processing in Linux software routers with respect to differentiated packet treatment of latency-sensitive applications at the ingress before the CPU bottleneck. Therefore, we propose dedicated Rx rings for latency-sensitive traffic which are served by a specific scheduling strategy. We model such an optimized Linux software router and evaluate its performance.

The rest of this paper is organized as follows. Section II explains the packet processing in a Linux software router and makes proposals for low latency improvements. In Section III, we introduce our simulation model of a Linux software router and extend it with our low latency concept. In Section IV, our software router model is applied in a case study to compare different scheduling strategies with respect to low latency packet processing. Finally, we conclude our main findings and give an outlook to the future work in Section V.

SummerSim-SPECTS, 2015, July 26-29, 2015, Chicago, IL, USA

II. Packet Processing in a Linux Software Router

Traditional general-purpose systems typically follow an interrupt request (IRQ) driven design for I/O devices in order to avoid CPU-intense polling. However, today it is known that such systems ,under some circumstances, suffer from high IRQ rates which can be generated easily by todays common GbE (Gigabit Ethernet) and 10 GbE adapters. J. Mogul et al. observed in 1997 that the throughput of Linux based systems may collapse as a consequence of a system state which is known as the *receive livelock* state [13]. In this state the system spends all its CPU resources on handling a flood of IRQs which are generated by NICs in order to trigger packet reception processes. Thus, the system has no CPU resources left for making progress in terms of packet processing, packet transmission, or other processes but wastes all CPU cycles.

Therefore, J. Salim et al. presented a new packet reception approach for Linux which they called NAPI (new API) [14]. Officially, the NAPI was introduced with the Linux kernel version 2.6 (2003) — also backported to Linux kernel version 2.4.20. However, with Linux kernel version 2.6.24 (2008) the NAPI has been revised. Today, more than ten years later, the NAPI is still a fundamental part of the Linux kernel network subsystem.

A. NAPI

Generally, the NAPI is a hybrid solution, which takes the advantages of both IRQs and polling. In case of low offered load, the system behaves like a system with an IRQ-driven design, whereat each packet causes an IRQ and the waiting-time of a packet is rather low. At higher loads, the ratio from IRQs to packets decreases and an IRQ causes the system to poll a number of packets in a row. This procedure saves costly IRQs and more CPU resources can be spent for the packet processing which in turn maximizes the achievable throughput.

In the following, we explain the basic concept of the NAPI:

When a packet arrives at the NIC it is collected in an *Rx buffer* and the NIC performs two tasks:

1) The NIC uses its DMA engine to transfer the packet from the *Rx buffer* to the main memory. Typically, a special ring-like data structure is used by the NIC driver in order to communicate with the NIC. Such a ring stores a number of descriptors that hold information like the descriptor state or addresses of allocated memory regions. With an *Rx ring* the driver provides a NIC with addresses of allocated memory regions that can be used by the DMA engine for packet reception. In turn a Tx ring is used to provide the NIC with memory locations of packets that should be sent through the link.
2) The NIC generates an IRQ (which is associated to the Rx ring) in order to inform the OS of the packet arrival.

By its nature the handling of an IRQ is a high priority task. The IRQ interrupts the CPU and must be handled by a CPU core — it is common to assign an IRQ directly to a specific CPU core. As a consequence of an IRQ, the responsible CPU core suspends its current task and executes the ISR (interrupt service routine) which is registered with the contemplated IRQ while the NIC driver is loaded. The ISR of a NAPI-compliant driver typically performs three tasks:

- Disable the IRQ (if this has not already happened in hardware). This procedure ensures that the NIC will not generate further IRQs until the IRQ is re-enabled. It is to mention that packet arrivals on *Rx rings* whose IRQs are disabled, are still transfered to main memory.
- An entry that identifies the Rx ring is enqueued in a FIFO (first-in first-out) queue (the so-called *poll list*), whereat each CPU core maintains its own queue.
- The actual packet processing task (which includes the packet reception in terms of software) is scheduled to be executed in process context, so the ISR can quickly return from interrupt context.

The packet processing task starts with a NAPI function that is responsible for the handling of the entries that are queued in the poll list. For this reason, the first entry is extracted and a virtual (driver-specific) *poll* function is executed. Anyway, the task of the poll function is to clean the contemplated Rx ring, where cleaning means to process a number of packet descriptors from the Rx ring which includes:

1) Fetching the associated packet data from the memory.
2) Wrapping the packet data into a kernel-compatible data structure a so-called SKB (*socket buffer*).
3) Passing the SKB to the higher protocols of the network protocol stack for further processing — typically the SBK is handed over to IP.

The poll function returns due to one of the following reasons:

- The Rx ring has been completely cleaned. The corresponding entry is removed from the poll list before the IRQ of the corresponding Rx ring is re-enabled.
- A number of *poll size* packets has been processed but the Rx ring still contains processable descriptors. Then, the corresponding IRQ of the Rx ring is not re-enabled, but the entry is re-enqueued at the tail of the poll list.

Fig. 1 shows the composition of the CPU utilization in case of using NAPI. As long as the offered load is low, the CPU core is able to process a packet and re-enables the IRQ before the next packet arrives — every packet causes an IRQ. If the sum of the packet processing time and the time to process the ISR exceed the inter-arrival-time, the CPU utilization reaches 100 % (at an offered load x') and the NAPI begins to process multiple packets per poll function call. Thus, the IRQ to packet ratio improves and the system can still cope with higher offered loads without packet loss. At an offered load of x'', packets arrive as fast as they can be processed, the NAPI often does not complete the polling, and no IRQs are generated at all. Hence, all CPU cycles can be spent for packet processing, which is why the system reaches its most efficient state in terms of throughput. However, if the offered load exceeds the value of x'', the system must drop packets due to lack of CPU cycles.

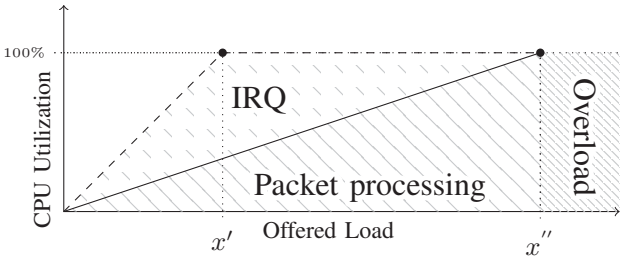


Fig. 1. CPU utilization consists of IRQ handling and packet processing

B. The ixgbe Driver

As mentioned in Section II-A, the implementation of the poll function is driver-specific. Here, we shortly explain the characteristics of the *ixgbe* driver since we calibrated our simulation model accordingly [15].

Due to the high costs of IRQs in terms of CPU utilization, Intel allows to pair an Rx and Tx ring. Both rings of a pair share the same IRQ. This means that the poll function must not only clean the Rx ring, but also the Tx ring. While an Rx ring is used by the NIC to deliver packets to the software, the Tx ring works vice versa (software to NIC). Furthermore, the Tx ring is used by the NIC to indicate the software that a packet has been sent through the link. This acknowledgment is required because the software has to free the SKB that is still allocated in the main memory, which happens when the Tx ring is cleaned.

With the actual implementation of the ixgbe driver, Intel decided to clean the Tx ring of an Rx/Tx ring pair in advance. In particular, this means when the poll function is called the driver is allowed to clean up to a number of *Tx work limit* descriptors successively before it switches to the Rx ring cleaning phase where the driver generally performs the steps that we already described in Section II-A — other NIC drivers might be implemented differently than explained here.

C. Hardware Support for Low Latency Packet Processing

Modern NICs support several features to shift packet processing tasks from the CPU to the NIC controller. For instance, the Intel 82599 [16] NIC controller provides multi-queueing and receive side scaling (RSS) [17] to distribute the packet processing efficiently across multiple CPU cores (cf. Fig. 2). NIC hardware filters allow to match on MAC or IP header fields like addresses, TCP/UDP ports, TOS (Type of Service), or VLAN tags. Thus, traffic classification can be conducted by the NIC and incoming packets are enqueued to a specific queue (Rx ring) based on specific packet attributes. After that, the packets are transferred into the main memory through DMA via the PCIe bus without CPU usage which may trigger an IRQ to a specific CPU core. If multiple Rx rings are assigned to a CPU core then the Rx rings are served in a round robin manner. In case of Linux, filtering can only be applied on the ingress traffic. Scheduling mechanisms like queueing disciplines (qdiscs) for differentiated packet treatment are only supported on the egress. Therefore, the recent

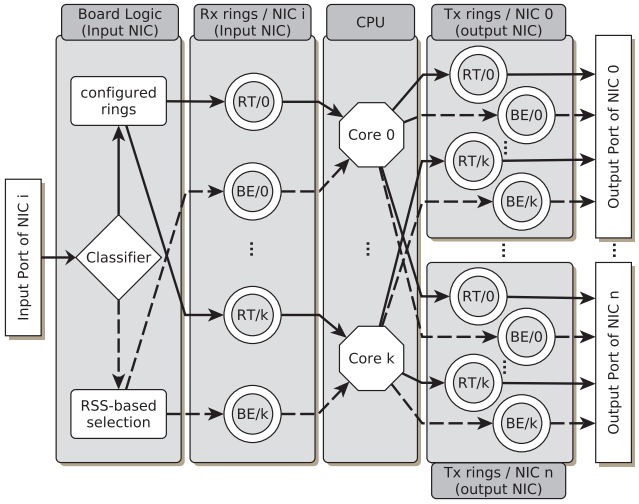


Fig. 2. Software router based on commodity hardware with RT and BE rings

QoS differentiation techniques of Linux are only applicable in scenarios where the link capacity of the outgoing NIC is the bottleneck. As a consequence, this QoS concept fails in scenarios where the CPU is the bottleneck instead of the egress link. However, many researchers and us showed that typically the CPU becomes the bottleneck in a software router [4], [18]. Based on that we argue that the incoming traffic must be classified before being processed by the CPU in order to provide efficient QoS differentiation.

We propose that the NIC classifies the incoming traffic into dedicated Rx rings for each traffic class per CPU core. Additionally, each Rx ring requires a priority corresponding to its traffic class. If an incoming packet does not match any of the NIC filters, it is by default enqueued into the best effort class. Finally, the Rx rings need to be processed corresponding to their priority with the help of a scheduling strategy (e.g. Prio). Our concept is not limited to the number of traffic classes but in the following we consider only two traffic classes, namely real-time (RT) and best effort (BE).

To implement our concept, the Linux NAPI needs to be extended that each Rx ring possesses a priority which is served based on a scheduling strategy. We expect that this NAPI extension will strongly improve the packet processing of latency-sensitive applications. We demonstrate this based on a detailed simulation model of a Linux software router by means of a case study (cf. Sec. IV). For further details, we proposed a low latency extension for the NAPI [19]. We also investigated the packet latency caused by software routers based on testbed measurements and simulations [15].

III. Linux Software Router Model

A. Modeling of Resource Management in ns-3

To model a Linux software router, we apply our general modeling approach for intra-node resource management in resource-constrained network nodes [12]. Among others, this modeling approach introduces the terms resource manager, task unit, and resource. A *resource manager* is an abstraction

of the operating system. A *task unit* (TU) models an OS process or thread which requires resources like a CPU core to be executed. All resources are managed by the resource manager according to a specific scheduling strategy.

B. Scheduling Strategies

In Unix-like operating systems the processes obtain a *nice* value to give a specific process more or less CPU time than other processes. Analogous, a task unit possesses a *task unit priority* TUP_i, $i \in \{1, 2, ..., n\}$. This priority may be used by a resource manager to arbiter between task units which compete for the same shared resource(s) (e.g. CPU). Corresponding to the scheduling strategy, the resource manager prefers a task unit with a high priority where TUP_1 is the lowest priority. Therefore, in case of resource contention, when another task unit with higher priority is requesting the shared resource at the same time, the resource manager may revoke a shared resource from a task unit with low priority depending on the scheduling strategy of the resource manager. In the following, we consider these resource management strategies.

- *Priority (Prio)*: The task unit with the highest task unit priority gets the resource(s). If a high priority task unit requires the shared resource and a task unit with low priority currently occupies it, then the resource manager immediately revokes the resource. A task unit with the highest priority does not release the shared resource until all packets are processed.
- *Round Robin (RR)*: The task unit gets the resource(s) for a constant time slice Δt. The respective task unit priorities are not considered, and thus, the time slice size is constant for all task units. The task units are served one after the other (fair queueing). Hence, no starvation occurs.
- *Weighted Fair Queueing (WFQ)*: The task unit gets the resource(s) for a time slice $\Delta t_k = \Delta t \cdot TUP_k / \sum_{i=1}^{n} TUP_i$ corresponding to its task unit priority TUP_k. Thus, the time slice of a high priority task unit is longer than the time slice of a task unit with lower task unit priority. Like RR, the task units are served one after the other.

These scheduling strategies are non-preemptive. Thus, a task unit is not interrupted during the processing of the current packet. In case of resource revocation, the resource-occupying task unit finishes the processing of the current packet.

C. Modeling of a Linux Software Router

Our Linux software router model consists of $n + 1$ NICs ($NIC_0, \ldots, NIC_n$) and $k+1$ CPU cores ($C_0, \ldots, C_k$). In the network simulator ns-3 each NIC is represented by a specific *Net Device*. The *Net Device Classifier* assures a one-by-one mapping between net devices and its corresponding net device task units in the ns-3 resource-management module. A net device task unit models the behavior of the NIC controller which classifies incoming traffic based on specific packet attributes. For instance, real-time traffic can be directed to a specific Rx ring which is modeled as the input queue of the NAPI task unit. After processing, the packet is enqueued in the corresponding transmission queue of the outgoing net device.

D. Modeling of the NAPI Mechanism with Task Units

Based on our unified modeling approach, we extend the generic task unit to a specific NAPI task unit that models the specific NAPI mechanisms (e.g. IRQ handling, polling).

In a Linux software router with multi-queue NICs, there may be multiple Rx rings which are served by one NAPI thread pinned to a specific CPU core. This NAPI thread is modeled as multiple NAPI task units. Each NAPI task unit possesses a dedicated incoming queue which models a specific Rx ring. Thus, to model a software router with m traffic classes, n NICs, and k CPU cores we need $m{\cdot}n{\cdot}k$ NAPI task units, where $m = 1$ in the classical NAPI. These NAPI task units compete for the same shared CPU core resource. Thus, NAPI task units which are pinned to the same core cannot be processed simultaneously. This refers to the fact that a NAPI thread in the real system can only serve one Rx ring at a certain time.

In case of NAPI, the interactions with the resource manager can be mapped to the following generic resource management message types:

- *ResourceRequest (REQ)*: The task unit sends this message to the resource manager to apply for a shared resource. In case of NAPI, this refers to the firing of an IRQ by the NIC to notify the CPU that a new packet has arrived.
- *ResourceReply (REP)*: This message is sent from the resource manager to the task unit to allocate a dedicated resource (e.g. CPU) to the task unit. In case of NAPI, this refers to the call of the poll function.
- *ResourceRelease (REL)*: The task unit sends this message to the resource manager to give back an allocated resource. In context of NAPI, this refers to the case that the poll size was reached or the Rx ring was emptied.
- *ResourceRevoke (REV)*: This message is sent from the resource manager to the task unit to withdraw a shared resource in the case of resource contention. This is not considered in the classical NAPI. Thus, in case of Prio, we use this message type to switch the processing to an Rx ring with a higher priority.

Besides, the NAPI task unit inherits the following generic task unit attributes.

- *Priority*: The task unit priority is used by the resource manager to arbiter resource contention. This refers to the Rx ring priority which is not considered in the classical NAPI.
- *Queue Size*: The maximum incoming packets accepted by this task unit in number of packets. In case of a NAPI task unit, this parameter refers to the Rx ring size in number of SKB descriptors.
- *Processing Time*: This attribute specifies the time which is required to process a packet from the Rx ring.

In addition to the generic task unit attributes a NAPI task unit possesses the following attributes:

- *ISR Cycles*: The number of CPU cycles which are required to handle the ISR.
- *Poll Size*: The poll size defines how many packets can be processed in a batch from a specific Rx ring before

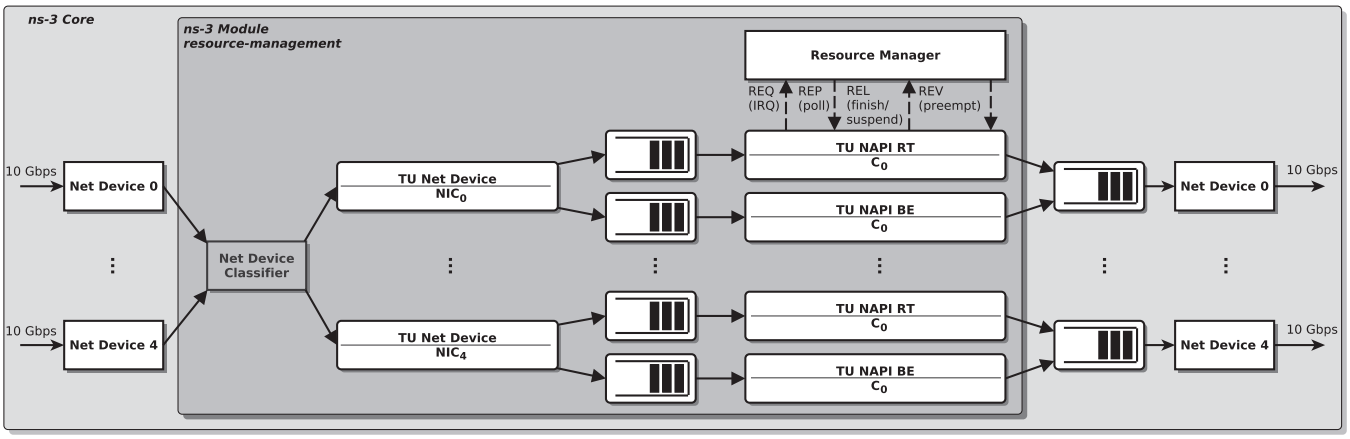


Fig. 3. ns-3 node model of the software router used in the case study

switching over to another Rx ring. The time slice size of RR and WFQ is interpreted as the poll size.

- *Tx Processing*: This attribute specifies the time which is required to free the corresponding SKB after packet transmission.
- *Tx Ring Size*: The Tx ring size defines the number of packets (represented as SKB descriptors) which can be stored in the transmission queue of the outgoing NIC.
- *Tx Work Limit*: This attribute specifies how many Tx descriptors can be cleaned during the Tx clean phase.

E. Model Limitations

The applicability of the software router model is subject to several limitations which are listed in the following.

- The simulation model assumes that the CPU states the bottleneck during the packet processing. Furthermore, we assume a linear scaling in the throughput when applying multiple CPU cores in parallel. This behavior was also shown by us [18] and other researchers [4].
- The simulation model considers modern NICs with multi-queue support which means that each CPU core has at least one receive and transmit queue per NIC. This assures that every CPU core is able to exclusively access its dedicated Rx/Tx ring without resource contention. Thus, the modeling of locking mechanisms to access the Rx/Tx rings can be omitted. Besides, the processing of a specific packet always remains at the same CPU core to avoid cache misses. Nonetheless, these facts align with common recommendations for parallel packet processing in multi-core PC systems [4].
- We assume that the outgoing link is not the bottleneck. Therefore, the model does not implement any queuing disciplines (qdisc) at the egress NIC. This is in contrast to many network simulators like ns-3 [9] which assume the outgoing link to be bottleneck.
- The model neglects concurrent processes or threads which may lead to context switches. However, concurrent processes can be easily modeled by introducing additional task unit which apply for the same shared CPU resource. Furthermore, the OS scheduling overhead for managing multiple processes is neglected.
- Additional latencies introduced by other hardware components (e.g. DMA, PCI) are omitted but can be modeled in more detail with the help of our ns-3 resource-management extension to set up more fine-grained case studies.

Further details of the NAPI modeling are given in [15].

IV. Case Study: Low Latency in Linux Software Routers

In this section we evaluate and optimize the packet processing performance in a Linux software router with the help of our ns-3 *resource management* extension.

A. Simulation Scenario

The ns-3 simulation scenario consists of five end systems connected to a Linux software router which acts as the device under test. On each end system a UDP client as well as a UDP server application is installed. The UDP clients generate traffic with a constant packet size of 64 B according to a Poisson stream which is uniformly distributed among the router ingress NICs. The traffic is a composition of real-time and best effort packets corresponding to a fixed mixing proportion (e.g. 30 % real-time traffic). The data rates of the point-to-point links of 10 Gbps are consciously chosen as very high-speed data rates to ensure that the links themselves will not become the bottleneck when applying data transmissions at a high level of offered load between the end systems.

Our Linux software router possesses limited resources. Thus, our ns-3 resource management module must only be deployed to the software router. The router model consists of 5 NICs with an Rx ring size of 512 packets per ring (cf. Fig. 3). We use only 1 CPU core running at 3.3 GHz because we showed that the performance of multi-core CPUs linearly scales with the number of CPU cores [18]. To distinguish between real-time and best effort, we require 2 traffic classes which implies 10 Rx rings (5 NICs $\times$ 2 traffic classes $\times$ 1 CPU core).

Based on real testbed measurements our Linux software router model was calibrated and validated for IP packet processing [15]. We profiled the networking stack of our Linux software router and observed that the service time is constant and independent of the packet size because each IP packet requires a similar number of CPU cycles for the IP header processing (e.g. routing table lookup). The packet processing consists of the Rx and the Tx phase which require ca. 441 ns resp. ca. 90 ns for each packet. In case that an incoming packet causes an IRQ, the ISR requires ca. 223 ns. Besides, we use a poll size of 64 and a Tx work limit of 256 which refer the defaults of the Linux NAPI.

As proposed in Section II-C, the real-time and best effort traffic is classified by the NIC based on the TOS field of the IPv4 header. To process packets, the task unit has to request the resource manager for the resource, here it is a CPU core. If there are resources available in the resource pool then the resource manager allocates a resource to the task unit. In case of resource contention and depending on the scheduling strategy, the task unit with the highest priority is preferred.

B. Simulation Results

We analyzed the mean packet latency and the corresponding maximum loss-free forwarding rate (MLFR) for real-time and best effort traffic with respect to several scheduling strategies to find an optimal trade-off between these contradicting goals. The MLFR is a performance metric of a packet processing system. It describes the maximum throughput in terms of million packets per second (Mpps) where no packets must be dropped due to overload. The real-time percentage is the mixing proportion between real-time and best effort traffic. In this case study a value of 10 % means that on average every tenth packet is a real-time packet and all other packets are best effort packets. In real use cases, we assume a real-time percentage between 5 % and 30 %.

As a point of reference to the state of the art, the default configurations of a Linux software router are also represented as single-queue (SQ) and round-robin (RR) scheduling strategies. In case of SQ, the router only possesses one queue per NIC per core with a size of 1024 to have the same total capacity as the other scheduling strategies. Thus, no QoS differentiation is applied and all packets are served in a first-come first-served order. With RR, the router has a dedicated queue for each traffic class per NIC per core but the real-time queue is not prioritized. The RR case can easily be configured with multi-queue NICs where the NIC classifies the traffic (cf. Sec. II-C).

1) Real-Time Percentage: Fig. 4(a) shows the percentage of real-time traffic of the total traffic on the x-axis and the mean packet latency in microseconds on the y-axis, stated with 95 % confidence intervals. Each of the lines represents a different scheduling strategy with respect to real-time or best effort traffic. The offered load is kept constant at 90 % of the MLFR of Prio for a specific real-time percentage. Thus, if the real-time percentage increases then the corresponding offered load decreases due to the decreasing MLFR of Prio (cf. Fig. 4(b)). Consequently, we observe a hyperbola-like curve progression for all scheduling strategies. As a result, the same offered load is applied for all scheduling strategies with respect to a specific real-time percentage. This allows us to compare the scheduling strategies with each other.

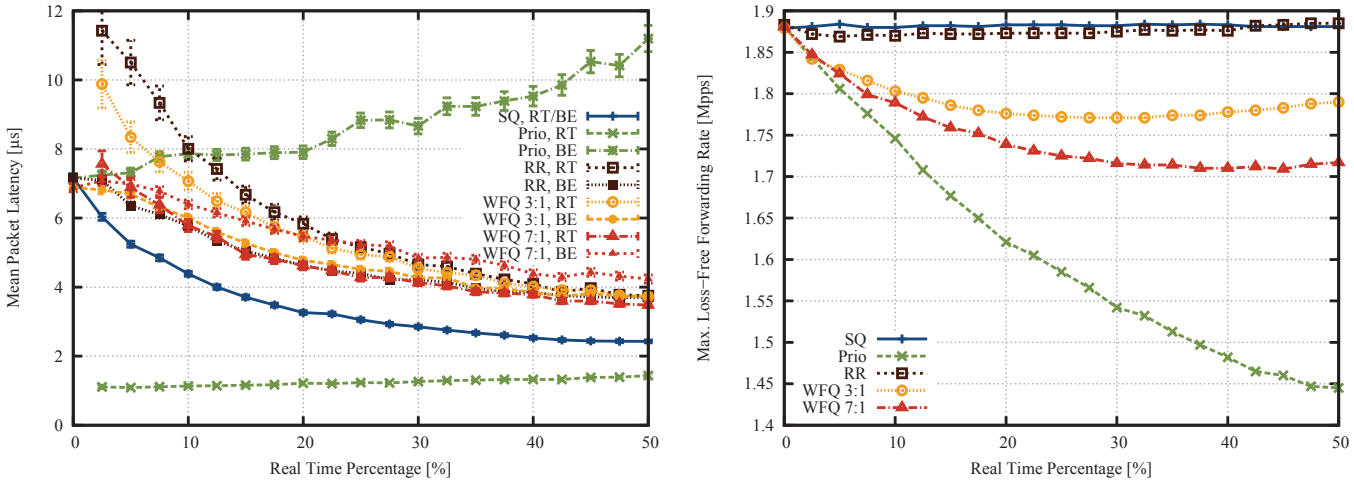
With SQ, all packets incur the same mean latency independent of the amount of real-time traffic. In case of RR, the mean packet latency for RT as well as for BE strongly increases because the dedicated Rx rings for the traffic classes cause additional IRQ overhead. For all real-time percentages, the RT packets show worse latency than the BE packets because the rarely present real-time packets often have to wait that the poll size of the BE ring is reached to get processed. Thus, RR is not applicable for low latency packet processing. In case of WFQ, a relation 3:1 means that the poll size of the real-time ring is three times larger than the poll size of the best effort ring. However, especially for low real-time percentages, WFQ works insufficient in RT traffic treatment independent from the weight relation. When applying Prio, a RT packet is always served before a BE packet. Thus, RT packets are served faster at the expense of the BE packets. This effect is more dominant for lower percentages of real-time traffic. With higher percentages of real-time traffic, there are many real-time packets in the RT Rx ring which are served in a FIFO order. Thus, the mean packet latency of real-time packets slightly increases. In case of no real-time traffic, our NAPI extension behaves like the state of the art if no scheduling strategy would be applied.

2) MLFR: Fig. 4(b) shows the real-time percentage on the x-axis and the MLFR in Mpps on the y-axis. Each of the lines represents a different scheduling strategy. The MLFR reaches the best values for SQ and RR because here the IRQ-induced overhead caused by RT traffic is lower in comparison to other scheduling strategies. In case of Prio, the MLFR strongly decreases if the amount of RT traffic increases. Nonetheless, we argue that in most cases, this is acceptable because we only assume real-time percentages for up to 30 % in reality.

3) CPU Utilization, IRQ Rate, Latency: Figs. 5 and 6 show the offered load in Mpps on the x-axis and the CPU utilization, the IRQ rate, and the mean packet latency on the y-axis. Each of the lines represents a different scheduling strategy with respect to real-time or best effort traffic where Fig. 5(a) illustrates the CPU utilization spent for actual packet processing (PP) and for IRQ handling. The real-time percentage is kept constant at 30 % in all cases. The CPU utilization is a dimensionless metric for the usage of the CPU as the relation between the busy time and the total time of observation. The IRQ rate is a measure for the IRQ overhead in number of IRQs per second fired by the NICs.

At offered loads below 1 Mpps, the CPU core is often idle. We observe that the CPU utilization as well as the IRQ rate linearly increases with the offered load because almost every incoming packet triggers an IRQ and is instantaneously processed. The mean packet latency is nearly equal for all scheduling strategies.

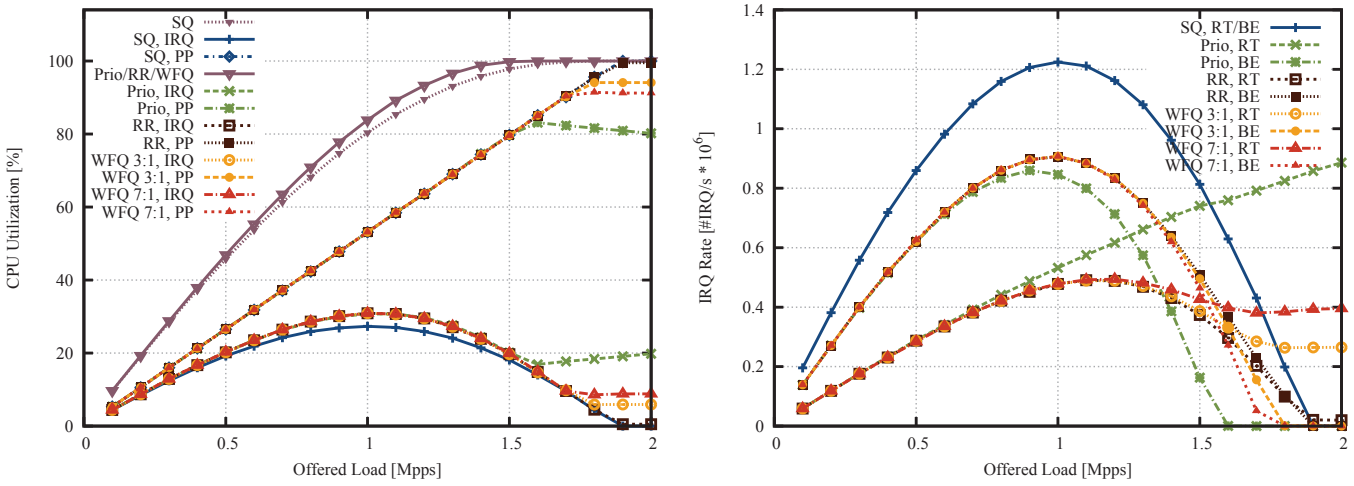
For offered loads higher than 1 Mpps, the IRQ rate decreases. This also decreases the IRQ-induced CPU utilization



(a) Mean Packet Latency

(b) Maximum Loss-free Forwarding Rate

Fig. 4. Mean packet latency and maximum loss-free forwarding rate as functions of real-time percentage for different scheduling strategies



(a) CPU Utilization

(b) IRQ Rate

Fig. 5. CPU utilization and IRQ rate as functions of offered load for different scheduling strategies

which reaches its maximum at 30 % for all strategies. Although the IRQ-induced CPU utilization decreases, the IRQ rate further increases with Prio because the NAPI often empties the RT Rx ring which reenables the IRQ of this ring. The corresponding packets have to wait until the CPU core becomes available which increases the packet latency. Thus, the mean packet latency of BE packets increases exponentially if the offered load is raised to 1.88 Mpps.

For offered loads higher than 1.88 Mpps and especially for BE traffic, the NAPI cannot empty the BE Rx ring and the IRQs remain disabled to save CPU cycles for the actual packet processing. In case of Prio, BE packets must occasionally be dropped at offered loads higher than 1.6 Mpps due to the additional IRQ overhead caused by the RT Rx ring. Thus, the maximum packet processing rate is reached earlier, particularly in comparison to SQ and RR. Nonetheless, the mean packet latency of SQ and RR for RT packets is not acceptable at over 100 μs especially at high offered loads. Therefore, RR is not suitable for low latency packet processing. In contrast to Prio, the mean packet latency of best effort packets exponentially increases whereas it only linearly rises to 1.5 μs for real-time packets. This even holds for high offered loads and is beneficial to satisfy low latency constraints. In case of WFQ, if the offered load increases then the higher the priority of the real-time traffic is, and thus the corresponding poll size, the less is the mean packet latency of the real-time packets. The mean packet latency of RT traffic only linearly increases but is obviously worse than Prio.

In summary, the improvement of the mean packet latency for the real-time traffic is achieved at the expense of the best effort traffic and the MLFR. For instance, with Prio and 10 % real-time traffic, the MLFR slightly decreases to 1.75 Mpps which corresponds to 1.88 Mpps (SQ) resp. 1.87 Mpps (RR). However, the RT packets only experience a latency of 1 μs

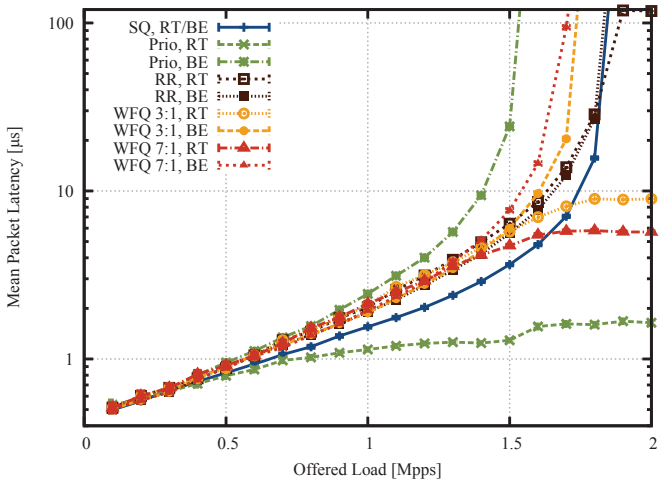


Fig. 6. Mean packet latency as function of offered load

which is significantly better than SQ resp. RR with 4 μs resp. 8 μs. At this point, the mean latency of BE packets is only increased to 8 μs with Prio. In case of RR, the BE packets are even served faster (6 μs) than the RT packets. This clearly shows that SQ and RR (as the state of the art of the current Linux NAPI) are not suitable for the packet processing of latency-sensitive applications. Thus, we propose to extend the Linux NAPI that the Prio scheduling strategy can be applied. Finally, this positive effect is strengthened at high offered loads where the real-time packets more strongly benefit from the NAPI extension.

V. Conclusion

In this paper, we described how the packet processing based on commodity hardware can be optimized for latency-sensitive applications. We proposed to extend the Linux NAPI, so that each Rx ring possesses a priority which is served according to a scheduling strategy. Thus, latency-sensitive traffic can be prioritized before reaching the CPU bottleneck. Our approach stands in contrast to traditional QoS techniques like DiffServ or IntServ which assume that the outgoing link is the bottleneck. Based on a detailed model of the Linux NAPI, we analyzed and investigated optimizations for the packet processing of latency-sensitive applications. Our case studies showed that the Prio scheduling strategy strongly improves the packet processing of latency-sensitive applications at relatively low cost in terms of the achievable maximum throughput. Therefore, we argue that our approach constitutes a fundamental enhancement of the state of the art.

In future, we plan to further refine our model. This includes a detailed modeling of the transmission path (qdisc) as well as the impact on the latency caused by other system components (e.g. DMA, PCIe). Besides, we will conduct testbed measurements with a prototype of our proposed low latency enhancements for the Linux NAPI.

Acknowledgments

This research has been supported by the *Deutsche Forschungsgemeinschaft* (DFG; German Research Foundation) as part of the *MEMPHIS* project (GZ: WO 722/6-1). We would like to acknowledge the valuable contributions through numerous in-depth discussions from our colleagues Prof. Georg Carle, Dr. Klaus-Dieter Heidtmann, Andrey Kolesnikov, Paul Emmerich, Daniel Raumer, and Florian Wohlfart.

References

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

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

[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] 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.

[5] 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.

[6] 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.

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

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

[9] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network Simulations with the ns-3 Simulator," *ACM SIGCOMM Demonstration*, August 2008.

[10] R. Chertov, S. Fahmy, and N. Shroff, "A Device-Independent Router Model," in *IEEE Conference on Computer Communications (INFOCOM)*, April 2008.

[11] S. Kristiansen, T. Plagemann, and V. Goebel, "Modeling Communication Software Execution for Accurate Simulation of Distributed Systems," in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, May 2013, pp. 67–78.

[12] T. Meyer, B. E. Wolfinger, S. Heckmüller, and A. Abdollahpouri, "Extensible and Realistic Modeling of Resource Contention in Resource-Constrained Nodes," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, July 2013, Best Paper Award.

[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 *5th Annual Linux Showcase & Conference*, vol. 5, 2001, pp. 18–18.

[15] 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.

[16] *Intel 82599 10 Gigabit Ethernet Controller Datasheet Rev. 2.9*, Intel Corporation, January 2014.

[17] Microsoft Corporation, "Eliminating the Receive Processing Bottleneck – Introducing RSS," in *Windows Hardware Engineering Community (WinHEC)*, April 2004.

[18] 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.

[19] T. Meyer, D. Raumer, F. Wohlfart, B. E. Wolfinger, and G. Carle, "Low Latency Packet Processing in Software Routers," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, July 2014, Best Paper Award.