Low Latency Packet Processing in Software Routers

Torsten Meyer[1], Daniel Raumer[2], Florian Wohlfart[2], Bernd E. Wolfinger[1], and Georg Carle[2]

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

***Abstract*—Commodity hardware can be used to build a software router that is capable of high-speed packet processing while being programmable and extensible. Therefore, software routers provide a cost-efficient alternative to expensive, special hardware routers. The efficiency of packet processing in resource-constrained nodes (e.g. software routers) can be strongly increased through parallel processing with commodity hardware based on multi-core processors. However, intra-node resource contention can have a strong negative impact on the corresponding network node. We describe how multi-core software routers can be optimized for low latency support by utilizing the technologies available in commodity PC hardware. For the analysis we used our approach for modeling of resource contention in resource-constrained nodes which is also implemented as the *resource-management* extension module for ns-3. Based on that, we derived a specific software router model which we used to optimize the performance. Our measurements show that the configuration of a software router has significant influence on the performance. The results can be used for parameter tuning in such systems.**

***Keywords* — parallel processing, resource contention, latency, delay, software router, modeling, simulation, ns-3**

I. Introduction

Commodity hardware is capable of high-speed packet processing [1] while being programmable and extensible. It can be used to build a software router. Software allows for rapid deployment of new features that require a more time-consuming and expensive development in hardware. Therefore, nowadays software routers become attractive as an alternative to existing special purpose hardware boxes that networks are built of [2]. Previous works have shown different bottlenecks that may limit the achievable throughput of a router in terms of bits or packets per second [1], [3], [4]. Some of these bottlenecks can be mitigated by efficient networking software [5]–[7]. Others require differentiated distribution of operations to hardware resources [8]. However, the new application of PCs as routers introduces challenges for packet treatment to PC-software development: The software should differentiate packets and consider that some packet flows are more critical in terms of Quality of Service (QoS) parameters like delay, packet drops, jitter, or connection establishment time [9].

In this paper we investigate differentiated packet treatment in software routers by making use of the low latency support by the underlying hardware. Therefore, we analyze the usage of specific Rx rings for low latency packets and selective packet drops in case of exhausted packet processing capacities. In Section II, we discuss the related work. Section III explains the hardware architecture and its implications for packet processing in a software router. Section IV derives a QoS-aware software router architecture. In Section V, we apply a general approach for modeling of resource contention in resource-constrained nodes to evaluate our concept. This software router model is used in a case study in Section VI to show how software router performance can be tuned. We summarize the paper and highlight our contributions in Section VII.

II. Related Work

With Netmap [5], PF_Ring [6], and Intel DPDK [7] three techniques exist that focus on the optimization of the software side of PC-based packet processing. They achieve a significant performance increase by melting driver, kernel, and even application parts of the processing chain. Mostly driven by hardware vendors similar efforts are made on the hardware side; e.g. DCA [10] has already developed into a standard technique in servers.

Know-how on measurement practice was described in 2005 by Tedesco et al. who published a technique to measure different parts of PC based packet processing systems with commodity hardware [11] based on a simple understanding of software router internal queueing: They measured 5 μs, 20 μs, and 5 μs for input queueing, processing, and output queueing. Carlsson et al. presented a delay measurement setup for IP routers as black boxes that follows the specifications of RFC 2679 [12]. The single hop hardware router delays that they measured were slightly higher than those obtained by Tedesco et al. They especially described a long tail distribution for packet delay. In [13], the authors utilized FPGAs for accurate software switch delay measurements. In 2007, Bolla and Bruschi presented a detailed study of a single core software router based on Linux kernel 2.6 and performed RFC 2544 conform tests on behalf of a special network device testing box [14]. The dedicated device testing box allowed to measure delay with microsecond accuracy. Depending on the type of software router, the configuration, and the packet size they measured delays from 14 μs to hundreds of μs. In scenarios where the CPU was the bottleneck, the delay increased to over 16 ms. In 2008, Bolla and Bruschi presented a study of architectural bottlenecks in software and hardware. They described and evaluated different uses of multiple Rx and Tx rings as these have been available [3]. A newer study of software router performance [1] and a study of performance based on different router workloads were published by Dobrescu et al [4]. The performance of software routers with a growing number of cores was analyzed in [15].

In [4], an analytical cache model for cache misses with multiple but well defined parallel packet flows on a multi-core software router was proposed. In the simulation community queueing systems with finite capacity are seen as more precise than those with infinite queues. These outperform analytical models in combination with traffic bursts. Chertov et al. presented a device-independent router model that takes into account the queue size and the number of servers inside a router [16]. With specific parameters, the model can be used for different router types. Bursty traffic, different packet sizes, and service times can be modeled on behalf of discrete event simulators like *ns-3*. In the *nsclick* project the *Click Modular Router* [17] was combined with ns-3 [18] providing the benefit of easy transfer of code from the simulation to software router deployments on real hardware. Kristiansen et al. [19] proposed a model for considering the packet processing overhead resulting from software. A model for resource-constrained network nodes which considers multi-core CPUs and other system internal components was presented in [20].

III. Packet Processing in Software Routers

The initial requirements to the Internet infrastructure have extremely grown with the increasing amount of use cases [21]. In the beginning, packets exchanged via the Internet Protocol (IP) contained text for asynchronous exchange of information. Today, Internet routers need to handle almost any type of traffic with diverse demands regarding available bandwidth, delay tolerance, or jitter. Some protocols require a certain percentage of dropped packets to adjust their higher-level communication channel (e.g. TCP), others are very sensitive to dropped packets (e.g. UDP). If the one-way delay in a telephone conference becomes greater than 150 ms the user experience is perceived as unacceptable [9], [22]; but the same delay for a file transfer is unproblematic. Especially traffic with real-time constraints (e.g. VoIP, online gaming) is very sensitive to delays but may be able to handle a certain percentage of packet drops due to failure correction mechanisms on higher layers.

A. Packet Processing in Software Routers

Software routers can perform all packet processing steps in software, so they work on any general purpose machine. In contrast, hardware routers implement performance-critical packet forwarding functions in special purpose hardware, while other more complex tasks (e.g routing protocols) are also processed on a general purpose CPU. Usually these tasks are not required in line rate and in most cases are synthetically limited to avoid overloading of the hardware. This ensures that a router can still process an ICMP ping even if it would be overloaded with SNMP requests by limiting the number of SNMP messages per second.

Off-the-shelf hardware has received features to cope with the growing number of cores and network speed. The CPU received an integrated memory controller (IMC) which provides a Direct Memory Access (DMA) engine to the PCIe connected components. Even preemptive copying of data to the caches [10] is common today. The network interface cards (NICs) provide features like receive side scaling (RSS) and segmentation offloading to shift tasks from the CPU to the NIC controller to distribute the tasks efficiently among the CPU cores. Since Linux kernel 2.5 the New API (NAPI) defines how network packets are received and transmitted.

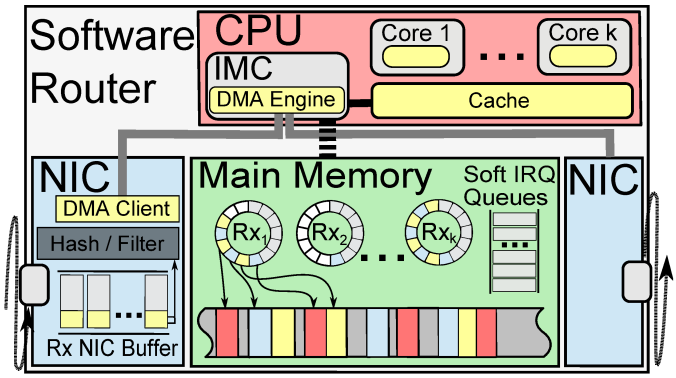

Fig. 1. Hardware resources in a software router

Fig. 1 shows resources that are relevant for software routers and their connections. Packets arriving at the ingoing interface are stored in the *Rx NIC Buffer*. NIC controllers like the Intel 82599 support programmable hardware filters and hash-based RSS [23]. These technologies enable the efficient distribution of network receive processing tasks across multiple CPU cores. Hardware filters allow to match header fields like IP/MAC addresses, ports, VLAN tags, or protocols to assign packets to a specific Rx ring, while standard RSS uses a static hash for the assignment. Modern NICs support configuration of numerous rings as destinations. Packets get transferred on behalf of the DMA engine via PCIe to the main memory. A DMA client can access a DMA provider (software) and the related DMA engine (hardware) to write and read data from DMA channels. This allows the NIC as DMA client to copy data to the main memory without involvement of a CPU core.

The DMA engine triggers a hardware interrupt for the core assigned to the Rx ring after the packet was copied to the Rx ring in the main memory. The Rx rings store pointers, which are called frame descriptors, to the actual packets that reside in an unordered manner in an extra memory area. Interrupts related to the different Rx rings can be assigned to certain CPU core interrupt queues. An interrupt triggered by the NIC due to an incoming packet is enqueued in the `net_rx_softirq` to avoid disruption of a busy CPU core. If the core is already processing the last batch of packets it may have deactivated interrupts and have switched to the polling mode. In polling mode the CPU cores fetch packets from the Rx rings. Polling mode ends when all tasks have been completed. To avoid blocking of a core polling mode also ends when a certain processing budget was reached. With the end of the polling mode the core is released and interrupts are reactivated. With interrupts activated the hardware interrupt entails different software interrupt routines of the kernel. These routines which process open tasks are stored in related SoftIRQs like the `net_rx_softirq`. The described process ensures that each packet of a specific flow is served by the same CPU core which avoids packet reordering and context switches between CPU cores. If more than one Rx ring is assigned to one core batches of packets are usually polled with a round robin strategy by the driver.

The CPU core processes packets according to the software

routines. Depending on the context of the application (user or kernel space) further copy overhead can be necessary for each context switch. In order to forward a packet, the router always needs to perform a lookup in the forwarding table, update the TTL (or hop count) field in the IP header, and trigger the sending process on the outgoing interface. If a packet is addressed to the router itself, such as routing protocol updates handled by XORP [24] or Quagga [25], the packet processing is more complex, however not as time-critical as in the case of packet forwarding. After the packet was processed the frame descriptor is placed in a Tx queue of the outgoing interface and the next processing steps are done by the egress NIC. The packet itself was not even copied once by the processing core but only pointers were copied and only relevant header information has been touched during this process.

B. QoS in the Linux Kernel

Linux kernel 3.3.0 introduced Byte Queue Limits (BQL), which allow to put a limit on the number of Bytes in an Rx ring, in addition to the existing packet count limit. Without BQL a full Rx ring would contain an unknown amount of data ranging from $Size_{RX} \times Pkt\_Size_{min}$ up to $Size_{RX} \times Pkt\_Size_{max}$, where Pkt_Size_{max} can be a multiple of the MTU due to the TSO/RSO mechanism which allows to offload splitting and merging to the NICs and therefore to send packets bigger than the MTU. By limiting the Bytes in the Rx rings the application of differentiated packet treatment is delayed to the different queueing disciplines (qdisc). In difference to the BQL the qdiscs are more powerful as they are implemented in the kernel only and do not require any support by the driver.

TABLE I. QDISC STRATEGIES IN LINUX

	classful	reordering	shaping
pfifo_fast	No	No	No
Token Bucket Filter (TBF)	No	No	No
Stochastic Fair Queueing (SFQ)	No	fair	No
Extended SFQ (ESFQ)	No	fair	No
Random Early Detection (RED)	No	No	dynamic
Hierarch. Token Bucket (HTB)	Yes	implicit	implicit
Hierarch. Fair Service Curve (HFSC)	Yes	fair	implicit
Priority scheduler (PRIO)	Yes	explicit	implicit
Class Based Queueing (CBQ)	Yes	implicit	implicit

Qdiscs are techniques for differentiated packet treatment in the Linux kernel. Filtering can be applied to ingress traffic of the Linux kernel and even more complex mechanisms which also include reordering to the egress or parts of the egress traffic of the Linux kernel. Table I shows existing queueing disciplines in Linux. All classful qdiscs can be applied to selected classes of traffic. Depending on the applied qdisc some packets are transmitted earlier as they would be transmitted with the standard first-come-first-served (FCFS) queueing behavior. Which techniques may change the order of packets when applied can be seen in Table I. Some algorithms cause packet reordering due to the goal of a fair bandwidth distribution to more competitors. The HTB and the CBQ qdisc implicitly reorder packets depending on the configuration as these are classful. PRIO explicitly reorders packets due to different prioritization. However, to have the best effect on the QoS of traffic passing a component each of these mechanisms must be applied before the bottleneck. Thus, with the described techniques, traffic shaping on a software router can avoid congestion of the outgoing Internet connection but not avoid service degradation due to an overloaded software router CPU.

IV. QOS AWARE SOFTWARE ROUTER ARCHITECTURE

Previous work described how software routers have to be configured to performantly utilize numerous Tx and Rx rings for efficient load balancing to different cores, but did not consider QoS differentiation [3]. Implementations of state-of-the-art QoS differentiation techniques in the Linux kernel (and other software routers) are well-suited for home routers and other scenarios where the link capacity is the bottleneck. However, we argue that these implementations do not work in scenarios where the software router is the bottleneck rather than the egress link. Previous work has demonstrated that the CPU is the main bottleneck in software routers [1], [4], [15], as other components such as the main memory and system buses usually handle significantly higher bandwidths than the CPU can process. In case the incoming traffic is overwhelming the CPU, such that it can not process all incoming packets, as soon as the Rx ring is filled some of the incoming packets are already dropped before being processed by the CPU. In this scenario, the approach to add traffic classification as just another step during the general packet processing does not work, because high-priority packets might have already been dropped before this processing step. This means, incoming packets need to be classified before being processed in the CPU, in order to provide QoS differentiation (e.g. upper bound for packet latency).

There are two possible approaches to solve this problem: First, one can dedicate one or more cores (as many cores as necessary to classify any type of incoming traffic at line speed) to receiving, classifying and forwarding the incoming traffic to the other cores for actual packet processing. This approach can be realized using PF_RING DNA clusters [26] (where the library *libzero* implements clusters to distribute incoming packets to multiple applications or threads) or Receive Packet Steering (RPS) [27] (which is a software implementation of RSS). The second and more efficient option is to offload traffic classification into a multiqueue NIC, which allocates different Rx rings for different priority classes. In this paper we opt for the latter approach, because it promises to be much more efficient and is easier to implement with today's soft- and hardware. For its implementation, we need to take care of two key points: We need to configure the NIC controller, that it recognizes and enqueues prioritized traffic into special Rx rings, and we need to find a strategy to assign and process these queues by CPU cores. Fig. 2 visualizes our approach. The receiving NIC (a) classifies incoming traffic into multiple Rx rings with different priorities (b) per core (c), which enqueues the processed packets into a Tx ring (d) of the transmitting NIC (e). An exclusive Tx ring for each combination of NIC and core allows to omit existing locking mechanisms.

A. Utilizing the Existing Hardware Capabilities

Fig. 2 shows a generic form of classifying traffic at the receiving NIC into multiple Rx rings with different priorities per core; $Rx_{n\text{-}RT,i}$ relates to non real-time (n-RT) rings and $Rx_{RT,i}$ to high priority real-time (RT) traffic rings with i denoting the core the ring is pinned to. The concept may also be used to configure more than two priority classes. In the following we refer to the specific features of the Intel 82599 Ethernet controller [23] as an example. This Ethernet controller offers more than one feature that can be used to implement

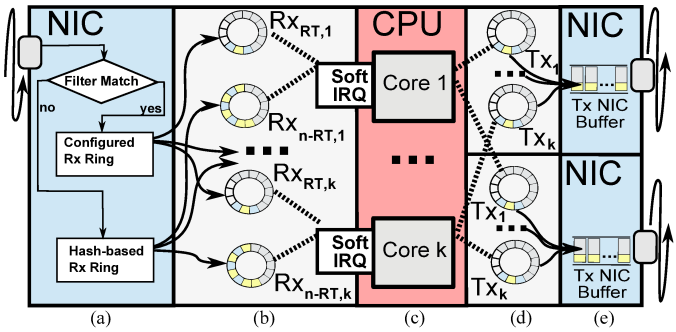

Fig. 2. Software router with support for differentiated packet treatment

our strategy, however each of them has a different application scenario. If the incoming traffic already comes with priority labels in their IEEE 802.1Q VLAN tags, a combination of the data center bridging (DCB) feature with receive side scaling (RSS) can automatically classify received packets into multiple Rx ring queues per core. If the incoming traffic does not carry priority tags, prioritized packets need to be identified using header information, such as IP addresses or port numbers. The Intel 82599 Ethernet controller supports various types of hardware filter rules, which can be used to match packet header fields and explicitly sort matching packets into a specified Rx ring. The packets that do not match any of these filter rules are put into the default Rx rings and thus can be distributed among all cores using RSS. For instance, the Intel 82599 Ethernet controller can match the following header fields: VLAN header, IP version (IPv4, IPv6), source and destination IP address, Transport layer protocol (UDP, TCP, SCTP), source and destination port number, or a flexible 2-Byte tuple within the first 64 Bytes of the packet.

B. Changes in the Software

Finally, we propose to extend the NIC driver, so that it supports different scheduling strategies to process packets from multiple Rx rings per core. By default the NIC driver applies a round robin strategy to poll multiple Rx rings, which means it iterates over all Rx rings, each time processing a batch of packets from an Rx ring. If a ring is empty it is skipped and the driver polls a batch of packets from the next Rx ring. In addition to this default scheduling behavior, we propose to modify the NIC driver and add other scheduling strategies, that prefer specific queues over others and meet QoS requirements. Without any significant computational overhead it is possible to implement a priority that only pulls packets from a ring if all higher prioritized rings are empty. Therefore, this approach is more flexible than dedicating one or more cores exclusively to prioritized traffic, which results in wasted clock cycles if the prioritized traffic does not fully utilize all dedicated cores. Other scheduling strategies, such as weighted fair queueing (WFQ) can also be implemented in the NIC driver with minimal overhead, so we do not expect a measurable decrease of performance from its implementation. The configured $weight$ guarantees a worst case share of high priority traffic in the maximum throughput TP_{max} of a software router of at least $\frac{TP_{max} \times weight}{\#cores \times \sum_{weights}}$ in case of skewed distribution of high priority traffic and $\frac{TP_{max} \times weight}{\sum_{weights}}$ if we assume high priority traffic that is evenly distributed to all cores.

V. Modeling Software Routers

In this section, we investigate how the performance of off-the-shelf software routers can be improved with respect to low latency traffic. Therefore, we introduce a model of a software router based on a standard Linux networking stack that is optimized for low latency packet treatment. This model is derived from our general modeling approach for resource management in resource-constrained nodes which was published in [20].

As it was already shown by us [15] and other researchers [1], the CPU cores represent the main performance bottleneck of a multi-core software router based on commodity hardware. Therefore, the cores' efficiency constitutes the main performance limiting factor and therefore has to be taken into account in great detail when evaluating such a system. Besides, there are usually one or multiple Rx rings per CPU core. In case of multiple Rx rings per core, the rings are served in a round robin manner. There is no support for prioritized packet treatment before reaching the CPU core bottleneck. However, this is important for software routers in high load situations (cf. Section III). Therefore, we extend our model of a standard software router based on our proposal for low latency traffic support. This is done by introducing dedicated Rx rings for low latency packets which will be served based on a specific resource management strategy (cf. Section V-C). For instance, this can be used to process packets faster with low latency constraints like real-time communications (e.g. VoIP, video conferencing, online gaming).

A. Model of a Software Router with Low Latency Support

According to the understandings of a real software router (cf. Section III), a model of an extended software router is derived which is depicted in Fig. 3. According to the general modeling approach [20], this specific software router model consists of three planes: the resource management plane, the resource plane, and the processing plane.

In our software router model, the resource pool RP_{Core} of the CPU cores contains k CPU core resources $C_1, \ldots, C_k$. The packet processing is modeled based on the task units $TU_{Mem}, TU_1, \ldots, TU_{2k}$ in the processing plane where TU_i $(1 \leq i \leq 2k)$ requires the resource CPU core $C_{i/2}$ if i is even (as in the case of non real-time traffic), or core $C_{(i+1)/2}$ if i is odd (real-time traffic). The TU_{Mem} abstracts NIC functionalities like RSS (cf. Section III) whereas the task units $TU_1, \ldots, TU_{2k}$ represent processes or threads of the operating system in the modeled system. The limited resources of the software router are modeled as resource objects which are located in specific resource pools in the resource plane. Furthermore, the Rx rings are represented as the task unit queues $Q_1 \ldots Q_{2k}$ which are located in the resource pool RP_{Mem}. The size of each task unit queue corresponds to the Rx ring size of the modeled system. Thus, a task unit queue can only store a limited number of packets. If not stated otherwise, we assume 4 cores and Rx ring sizes of 512 packets. These resource pools are administered by the (local) resource managers RM_{Core} and RM_{Mem}, whereas a global resource manager as introduced in [20] can be omitted because there are no dependencies between the two resource types.

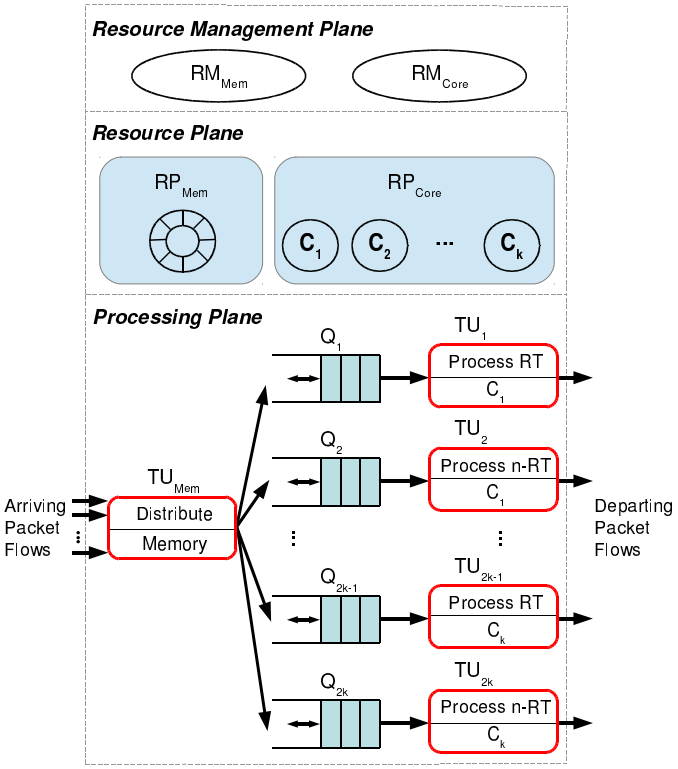

Fig. 3. Intra-node resource management model of a software router with low latency support

When a new packet is received, the TU_{Mem} distributes the incoming packet into a specific task unit queue depending on its packet attributes. Each packet belongs to a specific packet flow which is characterized by a source IP address, a destination IP address, a source port, and a destination port. Based on that, the TU_{Mem} maps each flow to a specific incoming queue Q_i of a task unit TU_i $(1 \leq i \leq 2k)$. In consequence, every packet of a specific flow is served by the same task unit corresponding to a specific CPU core. Furthermore, we distinguish between the packet attributes real-time (RT) and non real-time (n-RT). Based on that, the packets are enqueued in the corresponding task unit queues of the task units which process real-time or non real-time packets. For instance, a task unit that processes real-time packets may have a higher task unit priority than a task unit which processes non real-time packets. Thus, the processing of specific packets can be prioritized. The replicated task units $TU_1, \ldots, TU_{2k}$ model the actual parallel packet processing (e.g. IP table lookup, firewall) in multi-core software routers. To process a packet, a task unit requires a resource of the type CPU core. Therefore, it has to request to the RM_{Core} for allocating a core resource.

To give an example, the task unit TU_1 (which processes real-time packets) could require a CPU core from the RP_{Core}, namely C_1. However, it is possible that this core resource is currently not available because it could be allocated to TU_2 (which processes non real-time packets). Hence, resource contention occurs. Thus, the RM_{Core} has to schedule the resource allocation of the core resource(s) between the task units TU_1 and TU_2. After having been allocated the shared core resource, TU_1 is able to process packets from its incoming queue Q_1. The task unit functionality *Process RT* respectively *Process n-RT* consumes simulation time corresponding to the required service time depending on the packet size and type of packet processing (e.g. IP routing, IPsec).

B. Model Calibration

The packet latency represents the delay of a packet during its traversal through the software router. It consists of waiting and services times in several system internal components, where it is dominated by the waiting and service time at the bottleneck component, i.e. here the set of CPU cores. The waiting time of a packet depends on the number of packets prior to that packet in the task unit queue where the service time depends on the type of packet processing (e.g. IPsec, Routing, Firewall) and its packet attributes (e.g. packet size S, real-time or non real-time). For the service time x, we assume a variable part a per Byte and a constant part b, according to $x = a \cdot S + b$. We model the most practice-relevant type of packet processing, namely IP routing. Here, the packet is relayed from the incoming port to the outgoing port which is packet size dependent where $a \approx 2\,\frac{\text{ns}}{\text{B}}$ and $b \approx 272\,\text{ns}$. Furthermore, every packet is subjected to IP routing including routing table lookup, checksum calculation, etc. The effort for updating the IP header is equal for small and large packet sizes [28]. Thus, the effort for IP routing is represented with a constant overhead of $c \approx 225\,\text{ns}$. Therefore, the service time x for IP routing of a packet can be modeled as $x = a \cdot S + b + c$.

We also have to estimate the additional latencies from other system internal components. In modern NIC drivers, NIC batching reduces the overhead of interrupt handling through processing packets in a "bulk" from and to the NIC. According to [1], NIC batching and DMA transfer times lead to an increase of the packet latency. Based on that, we estimate that NIC batching introduces a delay for up to 16 packets before DMA transmission which implies 8 packets on average (if we assume uniformly distributed load). Besides, the processing of a packet in total requires four DMA transfers: Two transfers from the NIC to the memory (one for the packet and one for its descriptor) and vice versa. We estimate a DMA transfer at $T_{DMA} = 2.56\,\mu\text{s}$. Based on [1], we assume that NIC-driven batching from and to the NIC adds $T_{NIC} = 2 \times 8 \times x$ where x represents the service time in the core. Thus, we estimate an additional packet latency from other non-bottleneck components with $T^{+} = 4 \times T_{DMA} + T_{NIC}$.

For instance, in the case of routing a 1518 B packet, the service time is $x = 4.04\,\mu\text{s}$. Additionally, we assume that NIC-driven batching from and to the NIC adds 64.64 μs ($2 \times 8 \times 4.04\,\mu\text{s}$) on average. Based on that, we estimate an additional packet latency from other non-bottleneck components at 74.88 μs ($4 \times 2.56\,\mu\text{s}$ + 64.64 μs). This means that at offered loads below the maximum throughput of ca. 1 Mpps (0.25 Mpps/core) the packet latency refers to the sojourn time at the core bottleneck plus 74.88 μs to take into account the additional latency resulting from other system internal components. All used calibration parameters are derived from real testbed measurements (cf. Table 3 of [1]).

C. Resource Management Strategies

Each task unit possesses a *task unit priority* TUP_i, $i \in \{1, 2, ..., n\}$ which is used by a resource manager to arbiter between task units which compete for the same shared resource(s). Corresponding to the resource manager strategy, the resource manager prefers a task unit with a higher priority where TUP_1 is the highest priority. Therefore, in case of resource contention, the resource manager may revoke a shared resource from a task unit based on a specific resource management strategy because another task unit is requesting the resource at the same time. We model the following resource management strategies.

- **Priority (Prio)**: The task unit with the highest task unit priority gets the resource immediately. This strategy is non-preemptive which implies that a low priority task unit is not interrupted during the processing of the current packet.
- **Round Robin (RR)**: The task unit gets the resource(s) for processing for a time slice. The time slice length Δt is equal for all task units.
- **Prioritized Weighted Fair Queueing (PrioWFQ)**: The task unit with the highest task unit priority gets the resource immediately if there was previously no resource contention. In case of contention, a task unit gets the resource(s) for processing for a specific time slice corresponding to the task unit priority which means that the time slice Δt_{RT} of the high-priority task unit is longer than the time slice $\Delta t_{n\text{-}RT}$ of the low-priority task unit.

VI. Case Study: Low Latency Packet Processing in Software Routers

In this section, we evaluate and optimize the packet processing performance of an off-the-shelf quad-core software router with respect to low packet latency. Therefore, we aim to find optimal resource management strategies for software routers based on the software router model of Section V.

A. Simulation Scenario

The ns-3 simulation scenario consists of multiple load generators and sinks acting as end systems and a router serving as device under test (Fig. 4). The load generators and the sinks have no resource constraints, but the router possesses limited resources, namely 4 CPU cores, and a limited size of Rx rings of 512 packets. The ns-3 resource management extension [20] is applied to model the router under test.

The load generators and the sinks are connected via 10 Gbps point-to-point links to the router which are consciously chosen as 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. All of the applied traffic is a composition of RT and non real-time packet flows corresponding to the mixing proportion (e.g. 30 % RT traffic). The RT and n-RT traffic is separated by the software router based on the TOS (Type of Service) field of the IP header. The packet flows are uni-directional traffic from the load generators to the sinks. A specific packet flow is modeled as a video-conferencing flow which requires

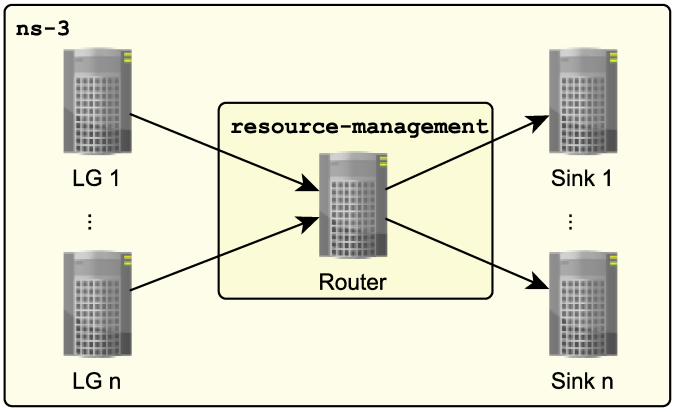

Fig. 4. Case study simulation scenario

ca. 5 Mbit/s respectively ca. 410 packets/s at a frame size of 1518 B. The frame size is constant for all traffic corresponding to the maximum transmission unit (MTU) of the Ethernet protocol. We apply a basic load of background traffic based on a Poisson arrival process which causes ca. 50 % utilization in the software router. Additionally, we inject packet flows which cause traffic bursts at normally distributed interarrival times. These bursts overload the software router for a short period of time. However, these bursts are short enough that no packet loss occurs due to an overflow of the Rx rings. Thus, the bursts lead to an increase of the number of packets in the corresponding Rx rings.

B. Simulation Results

We analyzed the mean packet latency for RT and n-RT traffic of the modeled software router with respect to different resource management strategies. As a point of reference to the state of the art, the single queue (SQ) and round robin (RR) configurations represent the mean packet latency behavior when using a software router based on the standard Linux networking stack. In the case of SQ, only one Rx ring is mapped to a specific core. Thus, no resource management strategy is applied. All incoming packets are enqueued in the same Rx ring and served according to FCFS service discipline without any prioritization of RT packets. In contrast to the state of the art, within the RR configuration a dedicated Rx ring for the RT packets and another one for the n-RT packets are used per core as it is illustrated by Fig. 3.

Besides, the resource management strategy of prioritization (Prio) represents borderline cases with respect to the mean packet latency for all resource management strategies. It represents a lower-bound for the RT packets whereas it depicts an upper-bound for the n-RT packets.

1) Real-Time Percentage: The real-time percentage is a mixing proportion between RT and n-RT traffic. For instance, a value of 10 % means that on average every tenth packet is a RT packet and all other packets are n-RT packets. Fig. 5 shows the percentage of RT 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 which are too small to be visible in the graphs. Each of the lines represents a different resource management strategy with respect to RT or n-RT traffic. The router utilization of 80 % as well as the time

slice sizes for RR ($\Delta t = 1.5\,\mu$s) and PrioWFQ ($\Delta t_{RT} = 6\,\mu$s, $\Delta t_{n\text{-}RT} = 1.5\,\mu$s) are kept constant in all experiments.

In the case of SQ, the RT and n-RT packets incur the same mean latency because no resource management strategy is used. When applying the Prio strategy, then an incoming high-priority RT packet is always served before a n-RT packet. Thus, with the Prio strategy RT packets are served faster at the expense of the n-RT packets. This effect is stronger the less are the percentages of RT traffic. With higher percentages of RT traffic, there are many RT packets in the same Rx ring which are served in a FCFS manner. Thus, the mean packet latency of the RT packets increases. The mean packet latency of RT respectively n-RT packets equals the SQ case, if the RT percentage is 100 % respectively 0 %. With the RR strategy, the mean packet latency of RT respectively n-RT traffic is smaller the less the percentage of the corresponding traffic is because the time slice sizes for both types of traffic are equal. When the percentage of RT traffic is 50 %, the RT and n-RT traffic suffer the same mean packet latency. On the one hand, the PrioWFQ strategy shows similar behavior as the Prio strategy at low RT percentages because in most cases no resource contention occurs. This implies that when a RT packet is received the high-prio RT task unit immediately gets the resource (after the processing of the current packet of the low-prio n-RT task unit). On the other hand, PrioWFQ becomes more similar to RR at high RT percentages because it is more likely that resource contention occurs and each task gets its corresponding time slice.

In general, even at a moderate CPU utilization of 80 %, the RT packets strongly benefit from the introduction of a resource management strategy like Prio. This effect is strengthened at higher values of utilization.

2) Utilization: The utilization is a metric for the degree of the resource occupation. It is defined as the relationship between the busy time of the resource and the total time of observation. It refers to the bottleneck resource which is here the set of CPU cores. Fig. 6 shows the utilization of the software router on the x-axis and the mean packet latency in microseconds on the y-axis. The mean packet latency is stated with 95 % confidence intervals which are again too small to be visible. The RT percentage of ca. 30 % as well as the time slice sizes for RR ($\Delta t = 1.5\,\mu$s) and PrioWFQ ($\Delta t_{RT} = 6\,\mu$s, $\Delta t_{n\text{-}RT} = 1.5\,\mu$s) are kept constant in all experiments.

At values less than 50 % utilization, the resource CPU core is often idle. Here, the mean packet latency is nearly equal for all resource management strategies. However, in case of a high-speed software router, we assume utilization values of 50 % and above. Thus, we focus on these cases. When the CPU core is busy, the corresponding task unit (and also the packets) has to wait until the resource becomes available which leads to an increase of the packet latency. If the utilization increases up to 100 % then the mean packet latency increases exponentially up to a maximum which is determined by the Rx ring size. The mean packet latency is no longer well-defined because arriving packets often come up with a full queue (aka. Rx ring) and must be dropped. Hence, the stated mean packet latency refers only to the successfully served packets.

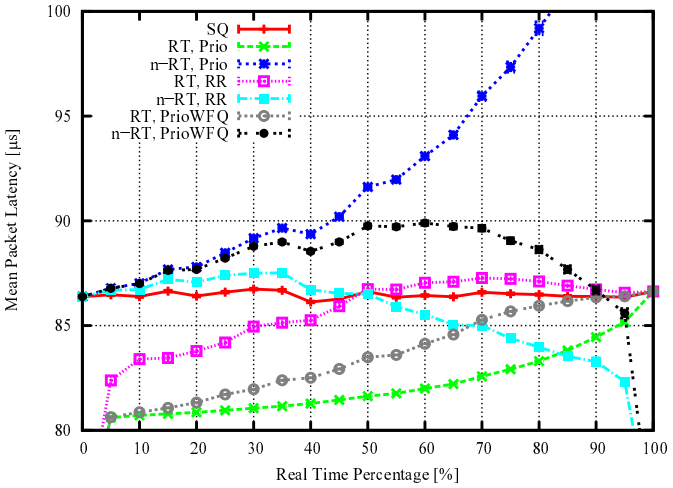

Fig. 5. Mean packet latency in dependence on real-time percentage for different resource management strategies

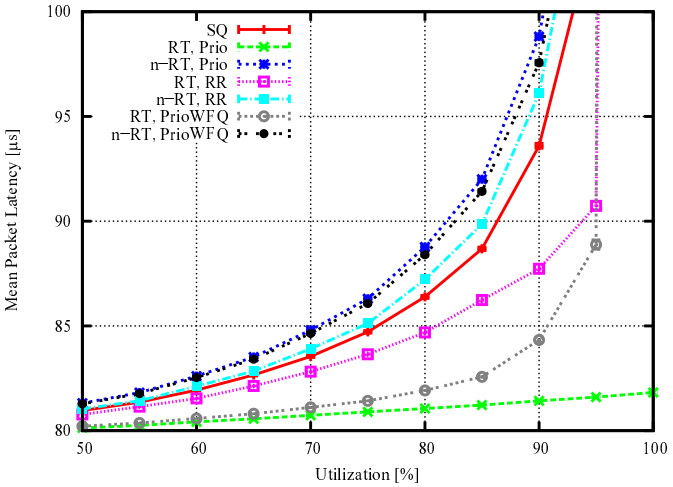

Fig. 6. Mean packet latency in dependence on utilization for different resource management strategies

The SQ case shows the default behavior as a reference to the state of the art when no resource management strategy is applied. In the case of RR, the mean packet latency of RT traffic is always smaller because the time slice sizes are equal for RT and n-RT packets but only 30 % of all packets are RT packets. In case of PrioWFQ, the mean packet latency of RT traffic is close to the Prio strategy for low values of utilization because it is likely that no resource contention occurs. However, if the utilization increases then in most cases there is resource contention which causes a rise of the mean packet latency of RT traffic similar to the RR strategy. When applying the Prio strategy, the mean packet latency of n-RT packets exponentially increases whereas it only linearly rises for the RT packets because the RT traffic is always served prior to the n-RT traffic. Thus, the RT packets show a significantly lower mean packet latency in comparison to the n-RT packets, even at high values of utilization. This is very helpful for a resource-constrained node to satisfy low latency constraints.

VII. Summary and Outlook

In this paper we proposed dedicated Rx rings to achieve low packet latency in packet processing systems. Thus, packets (e.g. with real-time constraints) can be prioritized before reaching the CPU core bottleneck. This enhancement has the focus on but is not restricted to software routers. We described how PC-based multi-core packet processing systems can be optimized to provide best effort for other parts of traffic in case of an overloaded software router just by utilizing technology that is already available in commodity servers. We used our approach for modeling of resource contention in resource-constrained nodes which is also implemented as the *resource-management* extension module for ns-3. Based on that, we derived a specific software router model which we used to optimize the performance of a software router. Our simulation studies showed that the configuration of a software router has significant influence on the performance. Therefore our results can be used for parameter tuning in such systems.

In future research, we plan to carry out more fine-grained testbed measurements to refine our resource-constrained software router model in terms of further performance-relevant details. For instance, a more accurate modeling of the effects on the intra-node latency resulting from other system internal components (e.g. driver) will be one of the next steps. Moreover, we will investigate further resource management strategies with the help of our ns-3 extension. Additionally, we want to look into the routing software using code inspection and profiling. Finally, we hope to be able to identify further performance-limiting factors and bottlenecks of existing software routers as well as to predict effects caused by changes and optimizations in the router software.

Acknowledgments

This research has been supported by the German Research Foundation (DFG) as part of the *MEMPHIS* project and the German Federal Ministry of Education and Research (BMBF) under EUREKA project SASER. We also would like to acknowledge the valuable contributions through numerous in-depth discussions from our colleagues Dr. Klaus-Dieter Heidtmann, Andrey Kolesnikov, Alexander Beifuß, Paul Emmerich, and Paul Lindt.

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] B. Munch, "Hype Cycle for Networking and Communications," Gartner, Report, July 2013.

[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, 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.

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

[6] F. Fusco and L. Deri, "High Speed Network Traffic Analysis with Commodity Multi-core Systems," in *Internet Measurement Conference*, November 2010, pp. 218–224.

[7] Intel, "Data Plane Development Kit: Programmer's Guide, Rev. 6," January 2014.

[8] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-Accelerated Software Router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, August 2011.

[9] M. Hassan, A. Nayandoro, and M. Atiquzzaman, "Internet Telephony: Services, Technical Challenges, and Products," *IEEE Communications Magazine*, vol. 38, no. 4, pp. 96–103, August 2000.

[10] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 50–59, May 2005.

[11] A. Tedesco, G. Ventre, L. Angrisani, and L. Peluso, "Measurement of Processing and Queuing Delays Introduced by a Software Router in a Single-Hop Network," in *IEEE Instrumentation and Measurement Technology Conference*, May 2005, pp. 1797–1802.

[12] G. Almes, S. Kalidindi, and M. Zekauskas, "A One-way Delay Metric for IPPM," RFC 2679, IETF, September 1999.

[13] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation," in *Passive and Active Measurement*. Springer, March 2012, pp. 85–95.

[14] R. Bolla and R. Bruschi, "Linux Software Router: Data Plane Optimization and Performance Evaluation," *Journal of Networks*, vol. 2, no. 3, pp. 6–17, June 2007.

[15] 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. 2, pp. 1–12, 2014.

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

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

[18] 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.

[19] S. Kristiansen, T. Plagemann, and V. Goebel, "Extending Network Simulators with Communication Software Execution Models," in *International Conference on Communication Systems and Networks (COMSNETS)*, January 2013.

[20] 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.

[21] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff, "A Brief History of the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 5, pp. 22–31, October 2009.

[22] S. Abbas, M. Mosbah, and A. Zemmari, "ITU-T Recommendation G.114, One Way Transmission Time," *Lect. Notes in Comp. Sciences*, May 2007.

[23] *Intel 82599 10 Gigabit Ethernet Controller Datasheet Rev. 2.76*, October 2012.

[24] "eXtensible Open Router Platform," http://www.xorp.org/, April 2014.

[25] "Quagga Routing Suite," http://www.nongnu.org/quagga/, April 2014.

[26] "PF_RING High-speed packet capture, filtering and analysis," http://www.ntop.org/products/pf_ring/libzero-for-dna, April 2014.

[27] T. Herbert and W. de Bruijn, "Scaling in the Linux Networking Stack," https://www.kernel.org/doc/Documentation/networking/scaling.txt, April 2014.

[28] R. Ramaswamy, N. Weng, and T. Wolf, "Analysis of Network Processing Workloads," *Journal of Systems Architecture*, vol. 55, no. 10, pp. 421–433, December 2009.