A Study of Networking Software Induced Latency

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

[1]Universität Hamburg, Department of Computer Science, Telecommunications and Computer Networks
{7beifuss|runge|wolfinger}@informatik.uni-hamburg.de
[2]Technische Universität München, Department of Computer Science, Network Architectures and Services
{raumer|emmericp|wohlfart|carle}@net.in.tum.de

***Abstract*—Most systems connected to the Internet are general purpose machines (except specialized routers and switches in the network core) that handle packet processing in software. Even in the network core, there is a trend towards packet processing in software, e.g. using OpenFlow or virtual switches. While packet processing in software is flexible and offers many capabilities, it also represents a challenge to evaluate, optimize, or predict the performance of such complex systems. This makes it hard to evaluate the networking performance of servers, end user hosts, or home routers. We present a study that investigates the packet latency caused by packet processing in the Linux network stack. We develop a simulation model in ns-3 for packet processing via the Linux network stack that helps understanding of its performance implications. We validate our simulation model based on measurements with nanosecond accuracy and software profiling.**

***Keywords*—latency, measurement, modeling, NAPI, ns-3**

I. Introduction and Related Work

Networking and Internet access is a core feature of any modern operating system (OS). Therefore, the OS needs to provide a network stack that is responsible for processing incoming and outgoing packets. The high complexity of an OS makes it hard to analyze and predict the packet processing performance and characterize performance guarantees. Specialized networking hardware, such as routers and switches, are optimized for high-speed packet processing and meet specified performance guarantees. Nevertheless, commodity hardware can be turned into routers, switches, firewalls, and other packet processing systems using software implementations, which makes them both more cost-efficient and flexible while still being able to scale up to high-speed traffic [1], [2].

As previous works have shown, the CPU is the performance bottleneck for packet processing systems [1]–[5]. This bottleneck can be mitigated by efficient packet processing software [6]–[8]. For example, interrupt moderation techniques which reduce the number of interrupts the CPU needs to handle [9], [10]. Other approaches use the massive parallel processing capabilities of dedicated graphics processing units (GPUs) [11].

As black box measurements [12], [13] only provide a rough understanding and white box measurements bare the risk of forged results due to the measurement side effects, previous work has addressed the challenges of measurements with a careful combination of both [1], [14]–[16]. Carlsson et al. [12] presented a latency measurement setup for black box measurements of routers according to RFC 2679. Rotsos et al. [13] utilized FPGAs for accurate software switch latency measurements. Bolla and Bruschi [14] presented a detailed study of a Linux kernel 2.6 based PC and performed RFC 2544 conform tests by means of a special network device testing box. The dedicated device testing box allowed to measure latency with microsecond accuracy. Dobrescu et al. [1] made analytical estimations for packet processing latency and combined them with the number of CPU cycles per packet that was measured with software profiling. Tedesco et al. [15] applied a queuing model to distribute measured latencies to different internal processing steps in a PC. Recently, Emmerich et al. [16] described practical know-how on throughput measurement of Open vSwitch in which they dealt with the problem of tampered results due to measuring.

Another approach to gain insights into the internals of complex PC-based packet processing systems is modeling and simulation. Therefore, Chertov et al. [17] presented a model of forwarding devices that can be configured to simulate the behavior of different devices. Kristiansen et al. [18] proposed a model for the packet processing overhead resulting from software. Meyer et al. [5] modeled resource contention in a server to analyze parallel packet processing with multiple cores. For our following analysis we distinguish three categories of tasks in network packet processing in PC systems: packet reception, application-specific processing, and packet transmission. We put our focus on the packet reception and transmission that is carried out by network stacks provided in OS and the NIC drivers.

In this paper, we measure and simulate how networking software like the network interface card (NIC) driver and the OS network stack influence the packet latency. We analyze NIC driver and OS mechanisms with respect to packet processing based on commodity hardware. We model these mechanisms and extended our previously proposed ns-3 resource management module [4] accordingly.

The rest of this paper is organized as follows. Section II reviews the developments in packet processing on PC hardware and introduces concepts to optimize the packet processing performance. Section III gives a step-by-step explanation of the Linux network stack and its interaction with the NIC driver. In Section IV we present the setup which was used for measurements in our testbed. We introduce our simulation model in Section V. Section VI contains the calibration and validation of our simulation model based on the results of our testbed measurements and simulations. We summarize our results in Section VII.

II. PC-Based Packet Processing

In the following we discuss hardware and software techniques that are relevant to mitigate different potential bottlenecks in PC-based packet processing systems.

A. Hardware

On the hardware side, the high network performance of PC systems can be attributed to two main developments: (1) Connections of hardware components and subcomponents in PC architectures and the related interaction processes are optimized, interactions are bundled and thus reduced. (2) The hardware underwent changes to cope with the growing number of cores and to shift workload from the general purpose CPU to dedicated components.

The first development results in a dedicated memory controller that provides direct memory access (DMA) for the NICs. A descriptor to the containing memory region is stored in a queue-like structure and via an interrupt the NIC informs the OS that a packet is ready to be processed. Even forehanded copying of data to CPU caches [19] is common today. Bus systems like PCIe allow for increased maximum data rates with each new version.

As a result of the second development the NICs (e.g. [20]) can distribute packets to different CPU cores via programmable hardware filters or static hash-based criteria. Even direct handover of specified flows to subsequent software processing steps by the NIC is common today. NICs also provide capabilities for packet segmentation, checksum calculation, and combination of shortly followed interrupts (interrupt moderation) to shift this workload of additional processing tasks and interrupt handling routines from the CPU.

B. Software

The actions for handling a packet after the NIC informed the OS are determined by software. While the driver coordinates the interaction with the NIC, the main functionality is provided by the OS which abstracts it via interfaces: in Linux this interface is called New API (NAPI), in Windows this is called Transport Device Interface (TDI). In the following we will focus on the Linux NAPI.

Legacy NIC drivers cause NICs to trigger an interrupt request (IRQ) for every incoming packet which implies IRQ storms in high load situations. As this overhead prevents the CPU from the actual packet processing, a new network interface was introduced in Linux kernel 2.6. It allows compliant NICs and drivers for IRQ mitigation to reduce the system load. Furthermore, the NAPI introduces a polling mechanism which enables the NIC driver to fetch multiple packets from an input queue while IRQs are disabled. Besides, it favors packet throttling in overload situations by early packet dropping directly in the NIC.

Modified drivers that do not conform NAPI and take polling to the extreme (e.g. busy-wait) like the Click Modular Router [21] achieve higher packet rates in comparison. However, they suffer from drawbacks like a permanent high CPU load due to active polling for packets even if no packets were received. Frameworks that fully rely on polling mitigate this by providing techniques to downscale the CPU frequency [8].

Depending on the purpose of the software on top of the OS network stack the relation of received traffic to the transmitted traffic can be of any type: 1:N (e.g. game server), 1:1 (e.g. firewall), N:0 (e.g. monitoring), 0:N (e.g. actor nodes), etc. The processing costs per packet vary from a constant number of CPU cycles per packet up to totally unpredictable per packet costs [22]. Processing is determined by software until a descriptor is placed in a Tx queue of the outgoing interface and the next processing steps are again performed by the egress NIC.

The applications can either run in kernel or in user space context. User space applications require additional copying which introduces extra overhead for each context switch between user and kernel space. Kernel space applications entail the risk of system crashes due to programming bugs. They require careful development and extensive testing due to the additional challenge of running in the kernel. Besides the Linux network stack, packet processing frameworks like DPDK [8], PF_RING [7], and netmap [6] exist, which replace the default network stack for certain purposes. These alternative network stacks can also be used in monitoring systems [22], web or game servers, (software) switches [23], [24], routers [24], (software) firewalls, or workstations.

There are currently four different approaches to optimize the performance of the network stack: (1) Avoiding the copy operation between kernel and user space processes by mapping buffer regions. (2) Preallocated packet buffers do not receive any supplementary adaptations and remain as initially configured to avoid any overhead. (3) The introduction of polling instead of interrupts. (4) Processing batches of packets with one API call on reception and sending to distribute the per-call overhead to a larger number of packets. Although these techniques allow high throughput rates they introduce drawbacks: (1) The lack of a standardized API via that applications can access the network increases the implementation complexity of applications. (2) Static buffer sizes prohibit adaptive reactions to filled buffers that introduce extra latencies in overload scenarios. (3) Continuous polling avoids sleeping of the CPU and counteracts power saving features that are desired in low load scenarios. The general purpose Linux network stack tries to satisfy any application by making tradeoffs between performance, usability, functionality, and power saving. The NAPI needs to perform well not only in providing high packet rates but also in low load scenarios, so specialized approaches outperform it in selected performance metrics, like the maximum packet rate. Nonetheless, the NAPI is widely-used today in end systems and servers due to its generality.

III. Linux-Based Packet Processing

In this section, we describe packet processing in Linux in detail. We describe the NAPI, explain the interaction between the NAPI and the NIC driver, and the interrupt throttling rate as an important configuration parameter for the packet processing.

A. NAPI

NAPI-based packet processing includes the following steps as depicted in Fig. 1:

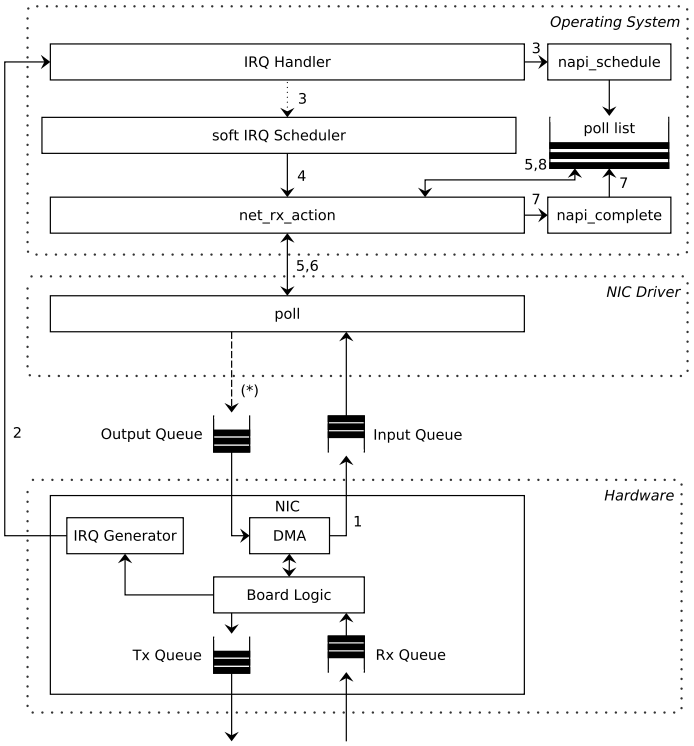


Fig. 1. Schematic view of the NAPI functionality

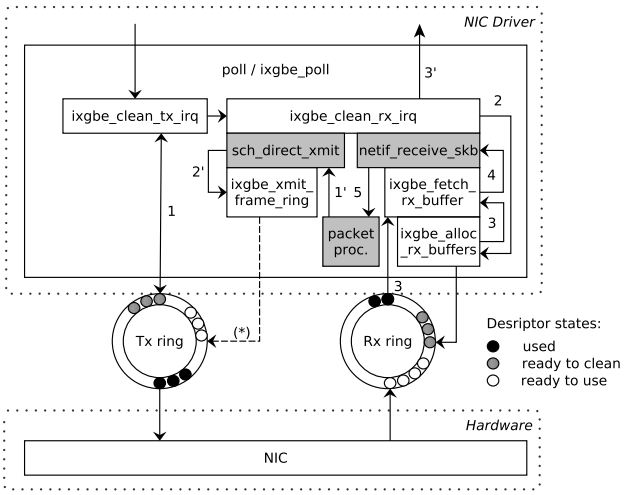


Fig. 2. NAPI in conjunction with NIC driver *ixgbe*; Shaded boxes represent functions which are part of the Linux kernel (not *ixgbe*)

1) The DMA engine copies a packet from the receiving NIC hardware *Rx Queue* to an *Input Queue* in the main memory.
2) The NIC triggers a hardware IRQ which is served by the assigned CPU core. The mapping between an IRQ and a CPU core that is intended to handle the IRQ can be statically assigned.
3) The *IRQ Handler* enqueues an entry referring to the *Input Queue* into the *poll list* (`napi_schedule()`) and raises a so called *soft IRQ* (`NET_RX_SOFTIRQ`). Each CPU core provides a dedicated *poll list* to schedule the queues that needs to be handled.
4) The *soft IRQ Scheduler* completes the soft IRQ and invokes the networking functionality (`net_rx_action()`).
5) `net_rx_action()` peeks the first entry of the *poll list* and initiates the poll (`poll()`). Since the implementation of `poll()` is driver-specific, we discuss its behavior in the next section.
6) A poll returns for two reasons:
  (a) The corresponding *Input Queue* is empty.
  (b) `poll()` yields after processing a certain quota (*poll size*) of packets to prevent other *Input Queues* from starving (Continue with step 8).
7) The respective entry is removed from the *poll list* (`napi_complete()`) and the poll finishes. (Continue with step 9).
8) The current poll is suspended although the *Input Queue* is not empty. The respective entry is re-enqueued into the poll list in a round-robin manner.
9) If the *poll list* still contains entries NAPI continues with step 5. Otherwise, the algorithm ends.

The handling of `net_rx_action()` is also limited to *budget* of OS processed packets as well as a timeout to share the CPU core with other competing devices or processes. If *budget* packets were processed or if the timeout expired then a `NET_RX_SOFTIRQ` soft IRQ is raised again and the CPU core is released.

B. NIC driver

For the description of the interaction between the NAPI and the NIC driver, we choose the Intel 10 GbE driver *ixgbe*. The *ixgbe* driver implements the *Input* and *Output Queues* as Rx and Tx rings which are continuously allocated memory blocks made of descriptors. These descriptors point to the actual packet buffers and are used by the DMA engine to copy packet data from the NIC to the main memory (Rx) and vice versa (Tx). In case no ready to use Rx descriptors are available to the NIC, new packets get dropped in hardware. This way an overwhelmed system is not bothered by additional incoming packets that cannot be handled anyway. If an Rx descriptor is available and a packet is received by the NIC, then it is stored in the NIC *Rx Queue*. The NIC's *Board Logic* fetches a Rx descriptor from the Rx ring and transfers the packet data via DMA into the associated buffer in the main memory. Afterwards, an IRQ is asserted and handled (cf. Sec. III-A, steps 1 - 6). A detailed view of the most important steps performed by the *ixgbe* driver's `poll()` function is provided in Fig. 2.

A feature of the *ixgbe* driver is that IRQs can be shared by Rx and Tx rings to mitigate the number of IRQs. This means an IRQ can indicate that a packet was received and has to be handled, or that a packet was transmitted and the Tx ring must be cleaned. For this reason the implementation of `poll()` is split up into the two following phases: `ixgbe_clean_tx_irq` and `ixgbe_clean_rx_irq`.

1) In the `ixgbe_clean_tx_irq()` phase, the driver cleans Tx descriptors from the Tx ring which are still associated to packets the NIC has already sent. The driver can clean up to 256 Tx descriptors consecutively (independent from the *poll size*).

2) The `ixgbe_clean_rx_irq()` phase starts with the recycling of Rx descriptors, which is done before they are returned to the hardware (`ixgbe_alloc_rx_buffers()`).
3) The packet data is fetched from the Rx ring: A Rx descriptor is read from the ring and a socket buffer structure (SKB) is created that points to the respective buffer (`ixgbe_fetch_rx_buffer()`).
4) After several sanity checks, the processing of the SKB is initiated (`netif_receive_skb()`).
5) The *Ethertype* determines how the SKB is processed.

With Open vSwitch, the actual packet processing for IP packets is defined by (`ovs_vport_receive()`). Open vSwitch determines the outgoing interface and the output queue. At this point, the packet transmission based on NAPI and *ixgbe* starts.

1') In the end of the processing, the SKB containing the packet gets scheduled for transmission (`sch_direct_xmit`).
2') A Tx descriptor is prepared in the Tx ring (`ixgbe_xmit_frame_ring()`). If more packets are available on the Rx ring and if the *poll size* is not reached, the algorithm continues with step 2.
3') In case the Tx and Rx rings were cleaned, the respective IRQ is re-enabled. If the dynamic interrupt throttling rate (ITR) is enabled, the ITR is recalculated to reprogram the NIC. Then, the poll returns to the NAPI (cf. section III-A, step 7).

C. Interrupt Throttling Rate

NAPI-based packet processing can be configured by several parameters. In case of using the *ixgbe* driver one of the most important parameters is the ITR. The ITR defines an upper bound of IRQs per second for a set of Tx and Rx rings. The ITR relies on a ITR timer which is set to $\frac{1}{ITR}$ after an IRQ was asserted. Until the ITR timer is expired, no further IRQs can be generated. If packet transmission or reception happened before the ITR timer expired, the IRQ is fired on timer expiration. Otherwise the next reception or transmission event immediately causes an IRQ. The ITR can be configured as static, dynamic, or disabled.

Disabling the ITR results in short packet latencies but has a negative impact on the maximum throughput, especially in high traffic load situations, where the CPU is often occupied with IRQ handling. Using a static ITR is suitable for manually setting the upper bound of IRQs per second which is then independent of the offered load. The increase of the ITR lowers the latency but increases the CPU utilization and may lower the maximum throughput. Hence, the appropriate configuration of the ITR is a trade-off between latency and maximum throughput.

With a dynamic ITR, the ITR is adopted according to the current traffic load. When a poll finishes, a new ITR is recalculated. The three ITR states *lowest*, *low* (initial state) and *bulk* are defined where each ITR state is associated to a specific ITR value in thousand interrupts per second (kips) as depicted in Fig. 3.

The current ITR state s and the throughput determine the transition to a new ITR state s'. The new ITR r' is calculated

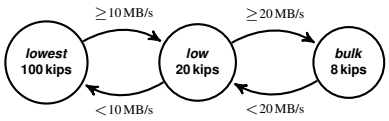


Fig. 3. Interrupt throttling rate states of the *ixgbe* NIC driver

on basis of the current ITR r and the ITR value of the new ITR state s' according to Eq. (1).

$$r' = \frac{10 \cdot s' \cdot r}{9 \cdot s' + r} \tag{1}$$

For instance, if the current offered load is low, and thus the throughput is low, then the ITR becomes high and vice versa.

IV. Latency Evaluation with Measurements

For the measurement of the NAPI performance a network stack is required that utilizes the NAPI. This network stack must not introduce any unpredictable effects into the measured data to avoid corruption of our packet reception and transmission measurements. In the best case it only utilizes a constant additional share of the CPU and adds a constant additional latency per packet to the measurements. Therefore, we decided to use Open vSwitch [25]–[27] as a representative NAPI-based in-kernel packet forwarding application. Open vSwitch is part of Linux and is able to operate in layer 2 of the ISO OSI stack but also in higher layers. Previously we have shown that Open vSwitch has a predictable average per packet processing cost in terms of CPU cycles [16].

A. Measurement Setup

Our test setup is based on recommendations by RFC 2544 [28]. The device under test (DuT) is connected to a device which runs a load generator and a packet counter in order to measure the achieved throughput. For profiling of software the DuT runs the Linux tool `perf` to gather statistics like the interrupt rate. Profiling measurements were run for five minutes per test to get accurate results. Our tests indicate that running this utility on the DuT introduces an overhead that reduces the maximum throughput by 1 %.

The DuT uses an Intel X540-T2 dual 10 GbE NIC and is equipped with a 3.3 GHz Intel Xeon E3-1230 V2 CPU. We disabled Hyper-Threading, Turbo Boost, and power saving features that scale the frequency with the CPU load because we observed measurement artefacts with these features.

The DuT runs the Debian-based live Linux distribution Grml with a 3.7 kernel, the *ixgbe* 3.14.5 NIC driver with interrupts statically assigned to CPU cores. Open vSwitch is used in version 2.0.0 with manually created OpenFlow rules to match the traffic.

B. Load Generation

Our load generator is based on the high-performance packet processing framework DPDK [8]. This packet generator can reliably generate constant bit rate (CBR) traffic by utilizing rate control hardware features of a X540-based NIC [20].

We generate minimally sized Ethernet frames (64 B) because we have shown in previous work that the packet size does not affect the throughput [16]. Using the minimal packet size allows for a challenging load on the DuT without reaching the 10 GbE line rate.

C. Measurement Accuracy

Relying on data measured in software introduces uncertainty. Therefore, we rely on the hardware counters of our X540-based NIC [20], which we periodically snapshot for our measurements.

Utilizing the IEEE 1588 hardware timestamping feature allows for an accuracy less than 25 ns resulting. This results from the per interface accuracy of the clocks and the inaccuracy of synchronization between the clocks of two different interfaces. The synchronization is even required for two interfaces on a dual port NIC that share one chip because the two ports are completely independent and do not share a clock for timestamping. We record timestamps for randomly sampled packets at a rate of ca. 1000 packets per second to get statistically independent timestamps.

V. Latency Evaluation with Simulations

Simulations are a cost-effective approach to design, validate, and analyze proposed protocols and algorithms in a controlled and reproducible manner. Our proposed simulation model is able to imitate the packet processing software by means of NAPI and NIC driver behavior in a Linux system (cf. Section III). Besides the prediction of throughput and CPU utilization, our simulation model aims for the prediction of latencies introduced by the packet processing software for any offered load.

In order to simulate the software induced packet latencies of real systems, we model the scheduling of polls defined by NAPI and the dispatching of packets according to *ixgbe* as described in Section III.

A. Integration into ns-3 Resource Management

We implement our simulation model within the widely-used discrete event network simulator *ns-3* [29]. In our previous work, we presented a modeling approach for resource-constrained network nodes [4] which we applied to show the linear scaling of multi-core software routers [5]. The OS is modeled by the resource manager and the actual packet processing is modeled by task units. We implemented this modeling approach as the ns-3 resource management module. Now we extend this modeling approach with respect to the NAPI and NIC driver behavior.

B. Simulation Model

Fig. 4 illustrates our simulation model derived from a real Linux-based system with NAPI behavior (cf. Figs. 1 and 4). The resource manager can be seen as the abstraction of the NAPI functionality and a task unit represents the functionality of the NIC driver.

The *Resource Manager* is responsible for handling IRQs and managing the *poll lists*. In real systems the interrupt service routine is a high priority task that causes other processes

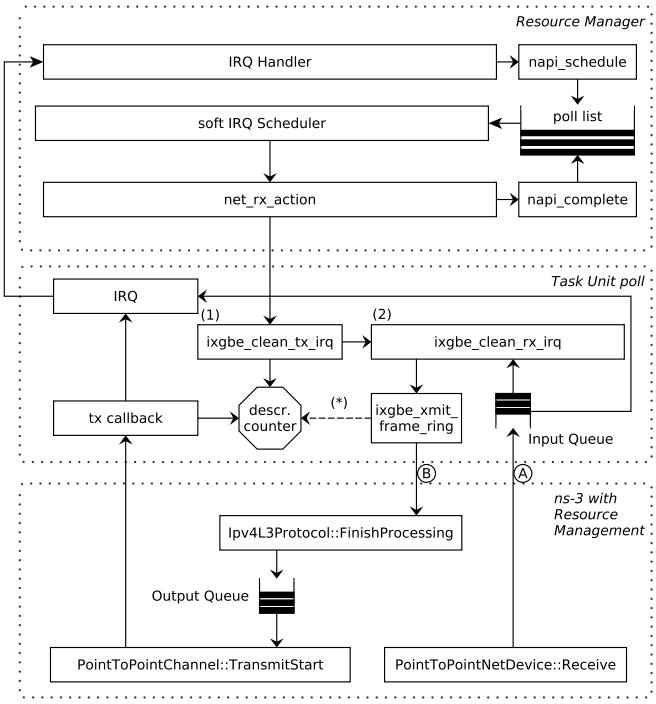


Fig. 4. Schematic view of the simulation model

to be suspended during IRQ handling. Our simulation model respects this and consumes a specific amount of simulated time t_{irq} for handling IRQs. The driver-specific behavior of the *poll()* function is modeled by a *Task Unit*. As described before *ixgbe*'s *poll* function splits up into two cyclic phases. Although in real systems numerous functions are involved in each of these phases, the successive performed steps are always the same. Thus, our model simplifies and considers each of both phases as a loop of a single step. The phases are:

Phase 1: Cleaning of the the *Tx ring* (`ixgbe_clean_tx_irq`). The total simulated time consumed in this phase depends on the number of packets that have been transmitted using the Tx ring. Each transmitted packet causes the simulated time to advance for a specific amount t_{clean}. In order to keep track of the transmitted packets, each *Task Unit* provides counters (*dscr. counter*) that are altered when a packet is scheduled for transmission (`ixgbe_xmit_frame_ring()`), a packet has been transmitted (`tx callback()`), or the *Tx ring* is cleaned.

Phase 2: Packets from the *Rx ring* are handled (`ixgbe_clean_rx_irq`). The number of successively handled packets is specified by the *poll size*. The model does not consider real packet processing but packet transmission. Thus, the simulation model has to schedule the subsequent transmission events *ixgbe_xmit_frame_ring* for up to *poll size* successive packets from the *Input Queue*. The simulated time between two consecutive transmission events is t_{proc} .

Additionally, each *Task Unit* needs to generate IRQs in order to indicate the *Resource Manager* that a queue state has changed. IRQs can be generated on *ITR* events. An *ITR* event schedules the next *ITR* event according to the ITR (cf. Sec. III-C).

As our simulation model focuses on the processing time consumed by software. We determine the packet latency, as the difference between the time the NIC passed the packet data to the *Input Queue* (cf. Fig. 4 marker (A)), and the time the software inserts the packet into the respective *Output Queue* (cf. Fig. 4 marker (B)).

Our model neither considers realistic DMA, modulation and demodulation latency in the NICs nor other latencies caused by hardware. Therefore, we introduce a constant offset value t_{off} which is added to each determined latency in order to estimate the total intra-node latency. Furthermore, our model neglects concurrent processes which might influence the packet processing. However, due to the strict separation of hardware, OS, and NIC driver in our model, it can easily be extended by adequate software or hardware models.

VI. Model Calibration and Validation

In this section we discuss our measurement and simulation results. We measured the per-packet processing latency on behalf of Open vSwitch. The real measurements were conducted in a testbed (cf. Sec. IV) and the simulations were made with our ns-3 resource management extension (cf. Sec. V).

For the calibration and validation of our model we simulate the system and the setting that we use in our real world measurements (cf. Sec. IV-A). This reflects the standard setup for device benchmarking [2], [28]: the DuT is charged by traffic that flows from a load generator to a sink. Both are directly connected to the DuT. In both, our simulation and our measurement setup we configured the DuT with the default Tx and Rx ring size of 512. The *poll size* is directly programmed in the driver code. For the `ixgbe` driver the *poll size* is 64 packets The ITR was set to dynamic ITR scheme. We load the system with traffic of different constant bit rates (CBRs).

A. Model Calibration

Model calibration is the procedure of setting the model parameters in the simulation model with respect to the modeled real system. We calibrate the model according to measurement and profiling results of the DuT in the testbed.

Based on our throughput measurements we obtain a maximum packet forwarding rate of 1.87 Mpps achieved by the DuT with one CPU core. With a CPU frequency of 3.3 GHz this relates to a processing time of 567 ns per packet. Profiling indicates that 99.5 % of these 567 ns are consumed by *ixgbe_poll* which is composed as follows: 15% (85 ns) correspond to *ixgbe_clean_tx_irq* (t_{clean}) and 85% (479 ns) correspond to *ixgbe_clean_rx_irq* (t_{proc}). Profiling revealed that handling an IRQ from the NIC takes 202 ns (t_{irq} ca. 667 CPU cycles).

From our latency measurement, we observe a minimum latency of approximately 8000 ns. The measurement was performed with low offered load (i.e. 0.0446 Mpps), so each packet received by the ingoing interface immediately generates an IRQ and the packet is processed directly (the Tx descriptor rings of the outgoing interface has been cleaned due to a separate IRQ for the prior packet transmission). Since a single packet takes 479.5 ns to be processed, we have to add an additional latency (t_{off}) of 8000 ns - 479 ns = 7521 ns to each processed packet.

t_{clean}	85 ns
t_{proc}	479 ns
t_{irq}	202 ns
t_{off}	7521 ns

TABLE I. Model calibration parameters

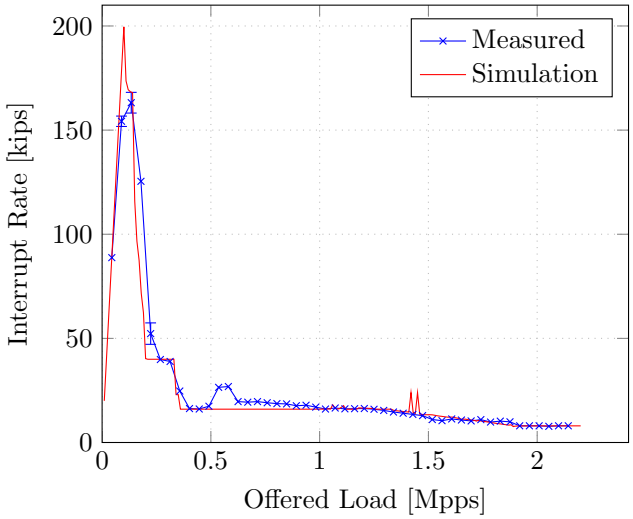


Fig. 6. Number of interrupts in dependence to offered load

B. Model Validation

Fig. 6 shows the measured and the simulated interrupt rate in dependence on the offered load in million packets per second (Mpps). The simulated interrupt rate reveals two abnormalities: (1) The peak at ca. 0.5 Mpps is missing. (2) Two peaks are visible at ca. 1.4 Mpps. (3) The interrupt rate decreases faster and less smooth than in the real system. We assign these discrepancies to effects not considered in our simulation model such as realistic DMA or hardware buffers.

The distribution of the packet latency is not normally distributed (cf. Fig. 7). Thus, we omitted mean values and confidence intervals. Instead, we denote the latency distributions (for measurements and simulation) by percentiles.

The observed latency distribution as indicated by the percentiles results from randomized sampling in measurements (cf. Sec. IV-A) and simulations. Fig. 5(a) shows the measured 25th, 50th, 75th, and 99th percentile for the observed latency

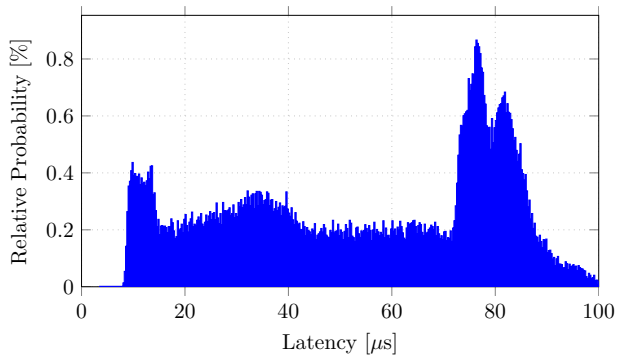
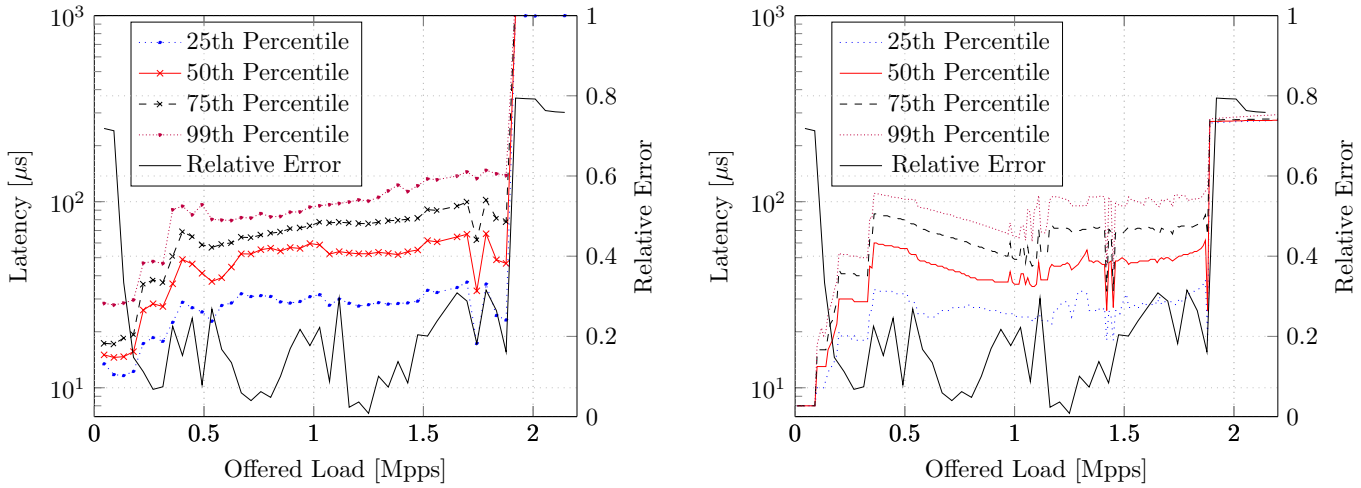


Fig. 7. Example of the non-trivial distribution of packet latencies: histogram of measured latencies at 1.03 Mpps, bin width: 250 ns



(a) Measured

(b) Simulated

Fig. 5. Distribution of latencies and the relative simulation error with varying offered loads

distribution in relation to the offered load. Fig. 5(b) illustrates the percentiles for the latency distribution predicted by our simulation model in relation to the offered load. An Xth percentile refers to the minimal value which is higher than X percent of the measured or simulated latencies.

The comparison of Fig. 5(a) with Fig. 5(b) shows: (1) For low offered load the simulated latency is constant while the measured latency increases. This error arises from the constant latency offset t_{off}, which we use to estimate the additional latency introduced by non-software parts of the system. (2) From 0.5 to 1.0 Mpps the measured latency is almost constant, cf.(a), while the simulated latency decreases, cf.(b). (a) For offered loads from 0.5 to 1.0 Mpps we see the measured interrupt rate is higher than the 16 kips we expect due to the ITR algorithm. According to the offered load the throughput on the Tx ring as well as on the Rx ring is above 20 MB/s. Therefore, we expect 8 kips at maximum for each ring but we observe up to 27 kips for both rings. Thus, we assume the ITR is oscillating. This effect occurs if a poll starts with many packets backlogged in the Rx ring while the current ITR has a high value. In this case the time between the poll finishes (all packets from the Rx ring are served) and the expiration of the ITR timer is short. In worst case the poll finishes shortly before the ITR timer expires and causes an IRQ. Since the observed throughput was high, the ITR is decreased. For the successive poll which starts immediately after the previous poll due to the IRQ, there are just few packets backlogged. In this case the time between the poll finishes and the remaining time until the ITR timer expires is long. Hence, many packets are backlogged for the successive poll. Furthermore, the ITR is increased due to the low throughput observed and the procedure starts over. An oscillating ITR potentially influences the distribution of latencies, because on the one hand the large backlogs introduce high latencies and on the other hand small backlogs introduce low latencies. In a real system it is possible that concurrent processes interrupt the packet processing and exceptionally more packets get backlogged. (b) If the interrupt rate remains constant and the offered load increases, then the mean latency decreases because more packets arrive during an active poll. Instead of being backlogged in the Rx ring and served by the successive poll, such packets are served directly by the poll they arrived in. The time these packets are backlogged in the Rx ring before being served is therefore short. This positive effect is a consequence of growing offered load and reaches its maximum when the packets arrive approximately as fast as they are served. This is why we observe a drop off in the measured as well as the simulated latencies right before overload situation (1.87 Mpps and above). (3) In overload situations the measured latency is four times higher than the simulated latency. In this case the latency is predominantly defined by the service time of the bottleneck (in this case the CPU) and the accumulated queue sizes in front of the bottleneck. Therefore, we assume this effect is related to additional hardware buffers which are not considered in our model (e.g. the Rx buffer in the NIC).

For the simulated and the measured 99th percentile of the latency distribution we calculated the absolute mean error $\overline{E}_{abs}$ = 16.049 μs and the relative mean error $\overline{E}_{rel}$ = 15.20 % that are defined as follows:

$$\overline{E}_{abs} = \frac{(\sum_{i\in n} |T^i_{sim} - T^i_{meas}|)}{n}, \overline{E}_{rel} = \frac{\sum_{i\in n} |\frac{T^i_{sim} - T^i_{meas}}{T^i_{meas}}|)}{n}$$

Where T^i_{meas} (resp. T^i_{sim}) is the observed latency of the ith measurement (resp. simulation) and n is the total number of sampled latencies.

The error plots in Fig. 5 show the relative error as a function of the offered load. The relative error increases drastically in cases of very low offered load and overload situations (due to unconsidered effects). Therefore, the (relative) mean error was calculated based on the measurements with an offered load between 0.2 Mpps and 1.87 Mpps. The confidence intervals were omitted because the results only show insignificant variances as they are based on CBR traffic. It is conspicuous that the error plot is unsteady. On closer inspection we noticed that the peaks for relative errors are in coincidence with disparities in the measured and simulated interrupt rates (except in overload situation). Therefore, if we manage to eliminate these discrepancies (e.g. by implementing

a proper DMA model), we expect the prediction of intra-node packet latency becomes more precise. Despite of the discussed deficits, the general shapes of the interrupt rate and latency plots indicate that our approach is basically correct. Thus, our model is suitable to predict latency related effects caused by NAPI and *ixgbe*.

VII. Conclusion

In this study, we investigated the latency which a packet incurs due to the packet processing software in PC systems based on Linux. We analyzed the interactions between the NIC driver and NAPI as part of the operating system. On the one hand side, we carried out testbed measurements to determine the distribution of packet latency with nanoseconds accuracy. On the other hand side, we modeled and simulated NIC driver and NAPI mechanisms by extending our ns-3 resource management module. Based on the testbed measurements, we calibrated and validated our simulation model with respect to packet latency. The simulation results show in comparison with the results of the measurements a rather accurate prediction of the packet latency. In future work, we will evaluate new algorithms for low latency packet processing (e.g. real-time support) with our validated model. Based on that, we will recommend optimizations for NIC drivers and the OS.

Acknowledgments

This research has been supported by the DFG (German Research Foundation) as part of the MEMPHIS project (CA 595/5-2), the EU as part of KIC EIT ICT Labs on SDN, and the BMBF (German Federal Ministry of Education and Research) under EUREKA-Project SASER (01BP12300A).

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] 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.

[3] 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.

[4] 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.

[5] 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. 37.2, pp. 93–107, 2014.

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

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

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

[9] J. H. Salim, "When NAPI comes to town," in *Linux Conference*, 2005.

[10] C. Benvenuti, *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2006.

[11] 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.

[12] P. Carlsson, D. Constantinescu, A. Popescu, M. Fiedler, and A. Nilsson, "Delay Performance in IP Routers," in *Performance Modelling and Evaluation of Heterogeneous Networks*, July 2004.

[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] 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.

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

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

[18] 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.

[19] 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.

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

[21] 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.

[22] L. Braun, C. Diekmann, N. Kammenhuber, and G. Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware," in *IFIP Networking*, May 2013.

[23] "Intel DPDK vSwitch," https://github.com/01org/dpdk-ovs, Intel Corporation, August 2014.

[24] L. Rizzo, M. Carbone, and G. Catalli, "Transparent Acceleration of Software Packet Forwarding Using Netmap," in *IEEE INFOCOM*. IEEE, 2012, pp. 2471–2479.

[25] "Open vSwitch," http://openvswitch.org/, September 2014.

[26] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," in *Workshop on Hot Topics in Networks (HotNets)*, 2009.

[27] J. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby, "Virtual Switching in an Era of Advanced Edges," in *Workshop on Data Center Converged and Virtual Ethernet Switching (DC-CAVES)*, September 2011.

[28] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices," RFC 2544 (Informational), Internet Engineering Task Force, March 1999.

[29] 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.