Optimizing Latency and CPU Load in Packet Processing Systems

Paul Emmerich¹, Daniel Raumer¹, Alexander Beifuß², Lukas Erlacher¹, Florian Wohlfart¹, Torsten M. Runge², Sebastian Gallemüller¹, and Georg Carle¹

¹Technische Universität München, Department of Computer Science, Network Architectures and Services
Boltzmannstr. 3, 85748 Garching, Germany
{emmeric|raumer|erlacher|wohlfart|gallenmu|carle}@in.tum.de
²Universität Hamburg, Department of Computer Science, Telecommunications and Computer Networks
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{beifuss|runge}@informatik.uni-hamburg.de

Abstract—High-speed network cards supporting 10 or 40 GbE (Gigabit Ethernet) are available today. Software frameworks for high-speed packet reception and transmission were created to exhaust the performance of these cards. However, these frameworks are not applicable as general-purpose solution. Thus, it is necessary to revisit general purpose network IO software that was designed more than a decade ago. In standard Linux settings, connectivity between applications and physical networks happens via the New API (NAPI). This motivated us to investigate how underlying NIC drivers can be adapted to improve latency in combination with the Linux NAPI. Based on testbed measurements, we propose an optimized algorithm for the NIC driver to dynamically adapt the Interrupt Throttling Rate (ITR). We implemented the algorithm and evaluated it with latency and throughput measurements based on the Linux module of Open vSwitch that operates on top of the NAPI. Our measurements show that our new ITR algorithm improves the packet latency without affecting the CPU load as much as other solutions.

Keywords — Linux, packet processing, packet latency, NIC driver, ITR, NAPI, commodity hardware

I. INTRODUCTION

Specialized networking hardware, such as routers and switches, are optimized for high-speed packet processing and meet specified performance guarantees. Nonetheless, commodity hardware can be turned into routers, switches, firewalls, and other packet processing systems by using software implementations, which makes them both more cost-efficient and flexible while still being able to scale up to high-speed traffic [1]–[3]. In commodity networked systems, network interfaces (NIC) with data rates of 1 GbE and 10 GbE (Gigabit Ethernet) are ordinary. Even 40 GbE NICs are being introduced to the market [4]. These high-speed NICs make it necessary to revisit the packet processing software that was designed without the presence of these data rates, NIC capabilities, and CPU architectures.

Software frameworks for high-speed packet reception and transmission like netmap [5], PF_RING DNA [6], and Intel DPDK [7] were proposed to replace the existing networking APIs. These frameworks achieve significant performance improvements by melting driver, kernel, and even applications of the packet processing chain. In spite of this, they are only an alternative for specific scenarios because they achieve the performance at the expense of breaking with common design concepts like a standardized and easy to use API. In contrast, general-purpose networking mechanisms like the New API (NAPI) [8] in Linux or the Transport Device Interface (TDI) in Windows are applied in the broad field. However, we observed that the Linux NAPI does not interact optimally with NIC drivers like the widely-used ixgbe. As a consequence packets incur unnecessary latencies inside the packet processing system.

In this paper, we propose an optimized algorithm for the NIC driver to improve the packet latency in combination with the Linux NAPI. We achieve this through an improved calculation of the Interrupt Throttling Rate (ITR) by using packet counters and more suitable time measurements. We implemented our approach in the NIC driver. Based on that, we conduct testbed measurements to compare our proposal with the status quo.

The remainder of the paper is organized as follows. In Section II, we explain the necessary background regarding packet processing in Linux. Our methodology for the conducted testbed measurements is described in Section III. We characterize the status quo performance characteristics of the Linux NAPI in Section IV. Based on that, we propose an optimized algorithm for the NIC driver to determine the ITR. In Section V, we show the benefits of our new ITR algorithm based on testbed measurement. We discuss in Section VI the novelty of our contribution in comparison with previous work. Finally, we summarize our proposals in Section VII.

II. PACKET RECEPTION ON LINUX

Within a pure IRQ-driven (Interrupt Request) system, each received packet causes an IRQ in order to inform the system that there is a packet waiting for processing. However, the high priority of IRQs lead to a trouble with this approach in case of high traffic load, as the system will go to a state which is known as receive livelock [9]. When a system enters this state, it will spend its CPU resources on IRQ handling, whereby the actual packet processing as well as every other process will starve. A queue which backlogs packets that are transferred from the NIC to the main memory on IRQ handling begins to overflow and received packets are dropped without being
完全处理。在极端情况下，处理的速率下降，直到网络负载放松。

在为了解决这个问题，引入了新的数据接收机制，称为 NAPI 机制 [8]，该机制是 Linux 内核版本 2.5.7（后端到 2.4.20）。它的名称是 NAPI（新 API）并且它是一个机制，在管理硬件中断数据接收与处理时，确保不同任务的优先级得到处理，即在接收数据时屏蔽中断。

此外，NAPI 支持早期的硬件数据接收，旨在提高多个核心的处理能力，考虑了公平性，构成了良好的折中，即在吞吐量和延迟之间。

甚至，存在专门的基于 Linux 的应用程序，它们依赖于与其他网络 IO 框架（例如 [5]–[7]）中所使用的机制，以更好地满足广泛应用程序的通用要求。

因此，NAPI 是支持多种网络子系统中的重要部分，它已频繁地作为软件路由器或交换机 [1]、[3]、[10]–[12] 以及其存在的原因。

A. NAPI Workflow

在第一步，数据接收的机制，通过 NAPI，NIC 转换数据包到 DMA（Direct Memory Access）数据包，从相应的硬件输入队列中放入一个缓冲器，该缓冲器位于内存中，并使用一个特定中断和数据包接收中断。此中断导致 CPU 核心执行在已注册中断服务程序（Interrupt Service Routine）时，如果硬件没有定义中断，它在内存中执行。

第一个动作是 NAPI-守点（或更精确地说是中断线）如果硬件没有定义守点。然后，NAPI-守点为中断服务程序（Interrupt Service Routine）设置一个内核级中断，该中断是注册好并被中断服务程序（Interrupt Service Routine）执行的。在第一阶段，如果数据包没有被处理，那么 NAPI-守点将被删除。

The soft IRQ Scheduler (Softnet) 完成软中断服务程序（Interrupt Service Routine）。

soft IRQ function 被识别为虚拟中断服务程序（Interrupt Service Routine），并使用 NAPI-守点机制。

These poll functions 可以定义为一个 NAPI-守点函数，该函数负责将数据包从内存中取出，然后推送到上层。

For fairness, a poll function should not fetch more packets than defined by a quota (known as poll size). Thus, a poll function either returns if all packets were processed or due to an exceeded quota.

In the first case, the poll function managed to process all packets and there is no more work to do, so the corresponding entry is removed from the poll list before the IRQ is re-enabled. In the second case, the IRQ is not re-enabled since there are still packets waiting, but the corresponding entry is moved from the head of the poll list to the tail of the poll list.

The described mechanism makes the NAPI act like pure IRQ-driven mechanism and generate one IRQ per packet for low loads. With growing offered load, such a system will reach 100 percent of CPU utilization fast. However, if the system reaches full utilization, the offered load still increases, it does not drop packets. Instead, the system begins to behave like a poll-driven system and the IRQ rate decreases. Hence, the CPU share of IRQs lowers continuously with growing offered load and more packets can be processed. At a specific point the IRQ rate drops to zero. This happens if packets arrive as fast as they are processed. In this case the system does not manage to clean the input queues and the IRQ is not re-enabled. Therefore, the CPU resources are completely used for packet processing. If the offered load grows further, packets get dropped.

Moreover, the NAPI allows for packet drops in hardware if the system is overwhelmed. This happens when the buffers on the NIC fill up completely while the, now poll-driven system, cannot process them further. IRQs are disabled, so the NIC cannot trigger further IRQs and incoming packets are discarded by the NIC without affecting the CPU.

In few words, the NAPI is both simple and efficient, a low loaded system spends the CPU resources to improve the packet latency while mid and high loaded systems spend the CPU resources for packet processing in order to maximize the throughput.

B. The Role of the NIC driver

By default the NAPI reduces the IRQ rate in case of a fully utilized CPU. Thus, the receiving process rather behaves like a polling process instead of a pure IRQ-driven process with a bad IRQ to packet ratio and the maximum throughput increases at the cost of latency (cf. Section II-A).

However, there are cases where it is desirable to manually influence the IRQ rate in certain ways. For example, a high CPU load at low packet rates is undesired. NICs like the investigated Intel NICs therefore support IC (Interrupt Coalescing) schemes in order to mitigate the number of IRQs that can be generated per second [15].

IC techniques are typically based on counters (e.g. packet counter and/or timeout counters) which are often offloaded to NICs and which are configured by the NIC driver (pure software solutions are also conceivable).

An example for such an IC feature is the ITR (Interrupt Throttling Rate) which is implemented for Intel’s 10 GbE
adapters [16]. NICs of this class have IRQ dedicated timeouts. These timeouts are configured accordingly by the driver (ixgbe) in the case that the poll function manages to process all backlogged packets that are associated to an IRQ. Until the timeout the correspondingIRQ is disabled (independent of the NAPI). If a packet arrives at the NIC within the timeout interval, then the IRQ is directly generated on the timeout, otherwise, the first packet arrival after the timeout will cause the generation of an IRQ. Figure 1 visualizes two packet arrivals and their respective IRQ.

Intel’s ITR algorithm has three different modes:

1) **Disabled**: The ITR is disabled. The IRQ rate is only influenced by the NAPI. This scheme provides best latencies, but it consumes a lot of CPU resources for the ISR.

2) **Static**: The timeout is configured with a static values (IRQ rate), which allows for specifying an upper bound of IRQs per seconds (per input queue) regardless of the current traffic situation.

3) **Dynamic/Adaptive** (default): In the adaptive mode, the driver adjusts the timeout according to the observed load (throughput). This scheme represents a trade-off between latency and CPU load.

Statically limiting the IRQ rate is not sufficient as higher IRQ rates at low packet rates are desired. The Intel ixgbe driver therefore implements dynamic adaption of the ITR with the load [16] on which we focus in the following.

The dynamic mode can be explained with the help of a state machine (cf. Figure 2). The state machine has three states, each state represents the current configuration of a timeout. The transitions between the three states are defined by thresholds which relate to the throughput \( \rho \). A ITR is typically set for the corresponding entry if it is removed from the poll list (cf. Section II-A). The IRQ rates and the thresholds cannot be configured by the end-user — unless they modify the ixgbe driver code. For a more detailed description of the Linux NAPI and the ixgbe driver the interested reader is referred to [17].

### C. The Trade-off between Latency and CPU Load

IRQ processing is an expensive task as it poses an additional overhead. Processing an IRQ for each single packet adds this overhead to the processing cost of each packet. Throttling the IRQ rate increases the number of packets that are processes per IRQ. The cost of an IRQ is distributed across multiple packets and the averaged clock cycles per packet are reduced. The downside is that low IRQ rates may have negative impact on the packet latency (especially in case of low to mid packet rates).

Both, the NAPI as well as the ITR feature, reduce the IRQ rate but they do it in different ways and with different goals. The NAPI is greedy for CPU resources and throttles the IRQ rate only if the available CPU resources are not sufficient to cope with the offered load. Hence, the NAPI focuses best packet latencies with respect to the traffic situation.

In contrast to the NAPI, the ITR feature specifies an upper bound in order to reduce the IRQ rate. However, CPU resources which are saved by the ITR feature typically cannot be spent to improve the throughput, as otherwise the NAPI would have throttled the IRQ rate too — the NAPI cannot be disabled and the ITR feature works in conjunction with the NAPI. Therefore, the ITR feature saves IRQs on the expense of packet latency and the saved CPU resources can either be used by other processes or for allowing the CPU to go into a lower power state in order to reduce power consumption.

In summary, this means the NAPI provides low packet latency at the cost of a high CPU load, while the ITR works complementary and gives better CPU load at the increase of the packet latency. Therefore, our objective is to improve the ITR algorithm in a way, that allows to reduce the CPU load but still provides low packet latencies. Therefore, our goal is to optimize the ITR implementation in a way that the ITR provides an optimal trade-off between the latency and CPU utilization.

### III. TEST SETUP AND METHODOLOGY

Our measurements were conducted on two servers with Intel X520 NICs that are connected via a direct 10 GbE fiber link. One server acts as a load generator and packet sink, the other server is the device under test (DuT).

#### A. Test Setup

- **a) Hardware**: The DuT runs a 3.3 GHz Intel Xeon E3-1230 v2 CPU. All features that scale the CPU frequency with the load (i.e. Intel Turbo-Boost and SpeedStep) were disabled to avoid measurement artifacts. The NIC is an Intel X520-T2 dual 10 GbE adapter which is based on the Intel 82599 chip [15].

- **b) Software**: The DuT runs Grml Debian live Linux with kernel 3.7, ixgbe 3.14.5, and Open vSwitch 2.0.0 as representative NAPI-based forwarding application on a 3.3 GHz Intel Xeon E3-1230 v2 CPU.

The CPU load was measured by reading the CPU’s idle cycle counter with the tool perf to obtain reliable measurements of CPU load caused by interrupts.1

Open vSwitch was statically configured to forward packets back to the load generator. We chose Open vSwitch because we are arguing about improvements at the lowest software layer, so the overhead of the forwarding application must be as low as possible. We have experienced in previous work [12] that Open vSwitch provides fast, stable, and constant-time inner kernel forwarding and therefore the best choice. Note that a forwarding application is necessary to send the packets back to the load generator to measure their latencies accurately.

1Standard tools like top or mpstat are not sufficient for such a CPU load measurement as the Linux kernel does not account CPU cycles consumed by hardware interrupts precisely enough by default [12].
c) Load Generator and Sink: We use our software load generator MoonGen [18] to generate constant bit-rate traffic for all measurements from a second server. It also measures the throughput by counting the incoming packets.

The used hardware timestamping technique allows for latency measurements with sub-microsecond accuracy and precision [18].

B. Test Methodology

We apply an increasing load of 0.02 Mpps to 2.5 Mpps (million packets per second) of constant bit-rate traffic on the DuT for each experiment. All packets are minimally sized as we have shown in previous work that only the packet rate and not their size matters for forwarding applications [12]. The DuT forwards the packets back to the load generator. Each test runs for at least 60 seconds and at least 60,000 packets are timestamped for each measurement point.

All tests were restricted to a single CPU core by configuring the NIC with only one queue and pinning its interrupt to a core. In previous work [12], we have shown that the maximum throughput scales linearly with the number of CPU cores. Restricting the DuT to a single core therefore simplifies our experimental setup without affecting the validity of the results for multi-core systems.

We define system overload as the point at which the DuT starts dropping packets.

IV. NAPI Performance

To discuss and evaluate optimizations of NAPI based packet processing, we first give an overview about packet processing latency with unmodified Linux systems.

A. Quantitative NAPI Performance Characteristic

As mentioned in Section II, there are two different algorithms controlling the IRQ rate: the NAPI and the ITR of the driver. We disable the ITR in the driver initially to acquire a baseline performance measurement.

Figure 3 shows the latency, CPU utilization, and IRQ rate of the DuT under increasing utilization. The CPU utilization increases linearly with the number of packets per second until it hits 100%. Latency decreases slightly with the number of packets at the beginning. This is likely an effect of powersaving idle states in the CPU. We only disabled frequency scaling, this does not affect the sleep states from which the CPU needs to wake up in order to process IRQs.

Once the system hits 100% CPU utilization, the latency is at its lowest point. Note that increasing the packet rate further does not cause an overload condition. Instead, the NAPI adopts and polls more often, this is visible in the decreasing IRQ rate. This mechanism is completely independent from the IRQ rate throttling found in drivers. Before the system becomes overloaded at 2.1 Mpps, the latency increases only marginally. Overloading the system causes all buffers to be filled completely causing a sudden jump to a large latency that is now dominated by the system’s buffer size. The latency under overload for our DuT is 2300 µs and omitted from this, and the following, graphs to improve the readability.

This experiment shows the best-case for the latency as there is no throttling. However, this is also the worst-case for CPU utilization due to the overhead of interrupt processing.

B. Interrupt Throttling in the ixgbe NIC Driver

As mentioned in Section II-B, the ixgbe driver measures the number of bytes processed between two IRQs and reduces the IRQ rate as the load increases. Figure 4 illustrates the idea of saving IRQs with this dynamic adaption. Point (A) represents the point at which the IRQ rate would peak without ITR (i.e. about 0.5 Mpps as seen in Figure 3). (B) is the point at which the system is in pure polling mode. The ITR sets in
V. IMPROVING THE DYNAMIC ITR

As described in Section II-B, the ixgbe NIC driver in the current version (3.23.2) offers the choice between dynamic adaptation based on data rate, a statically configured ITR, or no ITR at all [16]. We believe that the currently implemented dynamic ITR algorithm exhibits flaws that can be fixed. There are three points which can be improved: counting packets instead of bytes, measuring elapsed time properly, and using a different mapping between packet rate and ITR.

A. Counting Packets Instead of Bytes

Packet processing performance is usually limited by the number of packets processed, not by the bytes in contained in these packets. This is due to the inherent cost of processing a packet which dominates over the processing on the payload [5], [12].

Therefore, we use the metric packets per second instead of bytes per second as basis for the calculation of the ITR.

B. Measuring Elapsed Time

The ixgbe NIC driver uses the ITR to approximate the time since the last IRQ, i.e. it always assumes that the NIC is firing interrupts at exactly the specified maximum rate. However, this assumption is wrong. The NAPI disables IRQs during processing (cf. Section II-A), so the specified rate may not be reached. Therefore, the driver estimates the passed time as too short and the byte rate as too high.

Therefore, we replace this measurement with a proper time measurement provided by the kernel’s `getrawmonotonic` function.

C. Calculating the ITR

ixgbe currently only supports three different throttle rates in the dynamic adaption algorithm (cf. Section II-B).

We propose to replace this state machine with a continuous function illustrated in Figure 6.

We used our simulation model of the NAPI and the driver [17] to quickly test different algorithms. Based on these simulations, we propose the following formula to calculate the IRQ rate $r$:

$$ r = r_{\text{max}} - \rho \cdot \left( r_{\text{max}} - r_{\text{min}} \right) / \rho_{\text{max}} $$

Fig. 6. Schematic view of the IRQ throttling with our improved ITR algorithm
where $r_{\text{max}}$ is the maximum desired rate and $r_{\text{min}}$ is the minimum. We set these values to 100,000 and 8,000, respectively, as these two values were also used in the original implementation. $\rho$ is the current packet rate and $\rho_{\text{max}}$ the maximum packet rate before the system becomes overloaded.

$\rho_{\text{max}}$ is system-specific (2.1 Mpps here) and may need to be adopted for optimal performance. This parameter could be exposed through the configuration interface of ixgbe. Note that this would be an improvement over the existing algorithm, which uses the constants 10 and 20 MBytes/s that cannot be changed (cf. Figure 2).

D. Implementation

Implementing the algorithm requires modifying the function `ixgbe_update_itr` in the ixgbe driver. The packet statistics information that we use to replace the bytes statistics is already available to the function. We achieve better measurements of the passed time by using the Linux kernel’s function `getrawmonotonic` instead of the currently used inaccurate approximation. Changing the calculation is done by replacing the state machine with our formula.

Our patch for ixgbe 3.14.5, which was used for this evaluation, is publicly available at [19]. The latest version of ixgbe at present is 3.23.2. The ITR algorithm was not updated between these two versions. Our patch for this later version is available at [20].

E. Evaluation

Figure 8 compares the latency, CPU load, and IRQ rate of our improved algorithm with Intel’s default implementation. Both show the same behavior at low rates below 0.1 Mpps as there is effectively no throttling. Our algorithm then exhibits a higher CPU utilization while maintaining a latency that is almost as low as seen in the first measurement with no ITR in Section IV-A.

For better comparison we constitute a metric that takes the trade-off quality between CPU utilization and latency into account. The load-latency product $P_{\text{ll}}$ is defined as follows.

\[ P_{\text{ll}} = c_{\text{Pkt}} \cdot t \]

\[ c_{\text{Pkt}} := \frac{l \cdot f_{\text{CPU}}}{r} \]

$t$ is the mean latency and $c_{\text{Pkt}}$ is the number of CPU cycles required to process a single packet that is calculated from the CPU load $l$, its frequency $f_{\text{CPU}}$, and the packet rate $r$. The goal of a dynamic ITR algorithm is to reduce this product $P_{\text{ll}}$ which represents the trade-off between latency and CPU load. A high number of IRQs leads to a large $c_{\text{Pkt}}$ due to inherent costs of IRQ processing and small batch sizes. Larger batch sizes, however, lead to a large $t$.

Figure 7 shows the load-latency products of Intel’s algorithm and our algorithm and the difference at an offered load from 0 to 2.1 Mpps. Our algorithm shows a significant improvement, especially for medium packet loads. There is no trade-off decision to be made at low and high loads as
low loads are determined by IRQ-driven packet processing and high loads by polling.

VI. RELATED WORK

As described in Section II the NAPI allows for switching between polling and interrupt-driven packet processing. Salim described the processes in NAPI based packet processing in 2005 [21]. Switching between NAPI polling and interrupt-triggered packet processing was already addressed before interrupt throttling in drivers like Intel’s 10GbE NIC driver ixgbe was introduced. Dating back to that time, but published in 2009, Salah and Qahtan addressed the problem of switching the NAPI to polling mode on the basis of a packet rate based estimator [22]. This work is different from our approach as it addresses the problem in the NAPI by switching between interrupt-driven and poll-driven mode. We believe that the NIC driver is the right point at which the interrupt rate should be optimized. Salah and Qahtan did not evaluate their system behavior under high packet rates (only up to 0.25 Mpps).

In fact related work dates back even more to the past. Already in 1997, and before the introduction of the NAPI, Mogul et al. described an optimization that avoids pure interrupt-driven packet processing [9] as this may lead to receive live locks (cf. Section II). Later, several implementations following this idea where published by Maquelin et al. [23], Dovrolis et al. [24], Chang et al. [25], and Salah and Qahtan [22]. Getting the information that is necessary for the switch between polling and more interrupt-driven modes can be based on different information like packet inter-arrival times [24], the time a packet remains in the buffer [23], the buffer filling level [25], or the packet rate [22]. Especially gathering of the sojourn time a packet incurs in the buffer and the inter-arrival times entail high additional CPU costs and are, therefore, not practical.

Unfortunately older works cannot be directly compared due to significant improvements in hardware (PCIe, DMA, etc.) and software (e.g. introduction of the NAPI) architectures and its performance (e.g. the move to 10 Gigabit Ethernet and multi-core CPUs). These improvements changed the implications of polling and interrupt triggered packet processing concerning latency: while interrupt-driven packet processing was considered as exhibiting the lower latency [9], more recent work have shown the opposite [26] and a feature called low latency device polling has become part of the Linux kernel [27], [28].

However, pure polling approaches come with a significant overhead at low packet rates as the system needs to inquire about newly arrived packets instead of being informed about them by the NIC. This is a concern for the power usage as the CPU load is too high at low packet rates [29].

VII. SUMMARY

The ixgbe driver already includes an adoption algorithm for the ITR that provides an optimization of latency of Linux based packet processing. Although the ixgbe driver already interacts in an optimized way with the NAPI when compared to other drivers we have shown in this paper that this algorithm is still not optimal for the latency.

Adaptation happens in two discrete stages although traffic increases continuously. Based on a detailed analysis, we developed a linear adaptation of the ITIM which uses a traffic estimator based on the packet rate instead of the byte rate.

Our patch for the ixgbe driver that implements our proposed novel throttling algorithm is available publicly on GitHub [19], [20]. We invite you to try out our patch and reproduce the results from this paper to verify our work. We will also submit this patch to Intel for inclusion into the ixgbe driver.

ACKNOWLEDGMENTS

This research has been supported by the German Research Foundation (DFG) as part of the MEMPHIS project. We also would like to thank our colleagues Prof. Bernd E. Wolfinger, Dominik Scholz, and Sebastian Gallemüller.

REFERENCES


