Latency optimization and analysis through the use of a high-speed packet IO framework for high-bandwidth data processing

Marcel Glübert
Deggendorf Institute of Technology
Autonomous Systems / Driver Assistance Systems
Deggendorf, Germany
Email: marcel@gluebert.de

***Abstract*—The increasing data rate in Ethernet networks requires increased performance to receive and process data. The conventional Linux network stack encounters problems when processing packets at line rate of 10Gps or higher. To counteract this, specialized high-speed packet processing frameworks are required to harness standard hardware's full potential. To understand the required background, the mechanisms of modern network adapters and network stacks are explained by reference to Unix-based operating systems. Furthermore, the implementation of high-speed frameworks for receiving and processing image data will be discussed. Finally, a measurement will be performed, which shows the latency optimization by using the Data Plane Development Kit.**

***Keywords*—** High-Speed Packet Processing, High-Bandwidth Data reception, High-Speed Packet IO framework, DPDK, Data Plane Development Kit, Linux Network Stack, Flow Bifurcation

I. Introduction

Autonomous driving is one of the key technologies for the future. Accordingly, more and more vehicles are equipped with driver assistance systems. A large number of sensors are required to enable these systems to precisely detect their panoramic view. Imaging sensors, such as cameras, are preferred for this purpose. These sensors generate a large amount of raw data, which must be received and processed by a control unit. Current analyses assume that future autonomous vehicles will perceive their panoramic view by using eight cameras. [1] A typical camera with a resolution of $2.5MP$, a color depth of $12Bit$, and a frame rate of $30fps$ generate a raw sensor data rate of $105.47MB/s$.

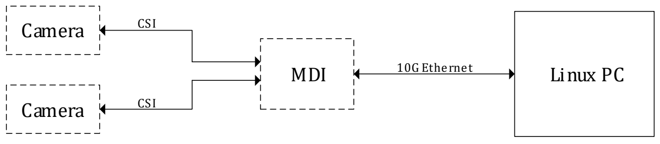

Fig. 1. Hardware Setup

One of the core tasks in the development of driver assistance systems is the acquisition of sensor data to develop and test these systems. The acquired sensor data are recorded in vehicle computers. The recordings are used to test new algorithms within a simulation environment before this is done for a vehicle on the road. Figure 1 shows a typical measurement setup. Data acquisition is considerably simplified if the data is transferred via a standardized interface. The Gigabit Multimedia Serial Link (GMSL-2) has become a standard for cameras. To record the sensor data, a Measurement Data Interface (MDI) is required, which receives the GMSL-2 data of the Image sensors, generates a timestamp and transmits the data by $10Gps$ Ethernet Interface.

The traditional Linux network stack has a decreasing overhead while receiving data. Within the Master Applied Research Project, a high-speed packet IO framework has to be implemented. The goal of the framework's implementation is the latency optimization of data reception, including the extraction of the image data. A comparison of the Linux network with the framework has to be realized, which shall clarify the optimization using a latency measurement.

II. Packet Processing

Ethernet packets have to be processed quickly to guarantee high data throughput for high-bandwidth data communication. The Ethernet standard (IEEE 802.3) specifies a maximum transfer unit (MTU) of 1500 bytes. The MTU can extend to 9100 bytes with so-called jumbo frames. Nevertheless, this is not specified in the Ethernet standard. Some switches reject these packets at store and forwarding due to the long processing time.

The processing time is particularly critical for small Ethernet frames. The following example shall clarify how fast the software has to work in order to process small packets on a $10Gbps$ interface. At a fully loaded Ethernet interface with 64 bytes of user data, 14.88 million packets arrive each second. Accordingly, a packet must be processed within $67.2ns$. With a processor clock of $1.8GHz$ one cycle corresponds to $0.55ns$, so one Ethernet frame has to be processed in 120 clock cycles.

This chapter gives an overview of typical software implementations for data reception with conventional hardware. In the beginning, the Linux network stack is examined in detail, and it is shown where problems with high data rates ($10Gbps$ and higher) occur. Afterward, two technologies are presented, which can accelerate packet processing.

A. Linux Network Stack

Figure 2 shows the structure of data reception in the Linux network stack. The network interface card (NIC) receives an Ethernet frame and stores it in its Rx buffer. After that, the DMA engine of the network card writes the frame to Random-Access Memory (RAM) via PCIe using the Memory Management Unit (MMU, part of the CPU) or to the CPU cache using Direct Cache Access (DCA). Provided the driver of the network card has reserved enough DMA/DCA descriptors (pointers to addresses in the RAM/cache) and that not all of them are used. After data is loaded into the RAM, the NIC generates an interrupt for the CPU. An event is stored in an event queue of the kernel (soft IRQ), which is processed immediately, but not directly. As soon as the soft IRQ is processed, the driver is called, rejecting the received packet and passing it to the network stack of the kernel (IP, TCP/UDP). The kernel processes the packet, and firewall rules may be applied. If the application is currently blocking or randomly polling for a packet, it is copied into the user space. There are several context changes between user space and kernel space. Interrupting and copying a packet from kernel space to user space creates an overhead. At high data rates and small packet sizes, packets can no longer be reliably processed and can be lost in the worst-case. [2]

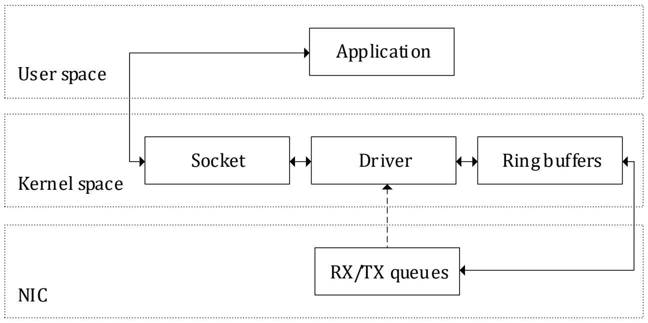

Fig. 2. Linux Network Stack structure

B. RDMA

Direct Memory Access (DMA) is an ability of a device to access host memory directly, without the intervention of the CPU. Remote Direct Memory Access (RDMA) is the ability of accessing (read, write) memory on a remote machine without interrupting the processing of the CPU(s) on that system. It has its origins in Infiniband (technology for data centers comparable to Ethernet), where it is part of the standard from the outset. In addition to Infiniband, RDMA can also be operated via Ethernet, for which there are two competing standards (iWRAP, RoCE). One of the RDMA Advantages is Zero-copy. Applications can perform data transfers without the involvement of the network software stack. Data is sent and received directly to the buffers without being copied between the network layers. An other advantage is bypassing the kernel. This implies that applications can perform data transfers directly from user-space without kernel involvement. All this is done with no CPU involvement. Applications can access remote memory without consuming any CPU time in the remote server. The remote memory server will be read without any intervention from the remote process (or processor). Moreover, the caches of the remote CPU will not be filled with the accessed memory content. [3]

The different RDMA protocols and the layers based on them are schematically summarized in Figure 3. RDMA is the link between the hardware and the application in the context of the kernel network stack. The network stack of the kernel exists parallel to the RDMA features, so regular Ethernet traffic and RDMA traffic can occur on the same interface. In the case of RDMA traffic, the DMA engine of the network card writes the data directly into the application's memory area, bypassing the kernel network stack and its overhead. RDMA requires hardware support, especially from the network card, and a suitable protocol. For this reason, this technology will not be considered further in this paper.

C. High-Speed Packet IO Framework

High-speed packet IO frameworks have been developed to make the processing of large amounts of packets more efficient. Among the most popular frameworks are DPDK, PFring and netmap. All three frameworks require modified drivers and use the same acceleration techniques. They bypass the standard network stack, i.e. the packets

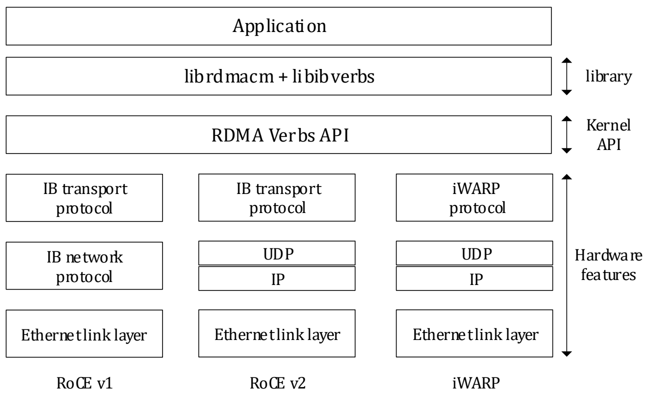

Fig. 3. RDMA protocols

are only processed by the framework and the applications. They also replace interrupts with polling when receiving data, which keeps a CPU core permanently busy polling the network card and thus avoids the overhead of an interrupt. They also avoid copying data between kernel space and user space because a packet is copied once from the NIC to user space via DMA.

III. Selection of Framework

In order to make a pre-selection of the framework, literature research was carried out. Performance, latency and detailed documentation of the framework are essential for this project. A research team of the TU Munich compared three different high-speed packet IO frameworks (PFRing, netmap and dpdk) in a paper [4]. Mainly the CPU clock speed, the number of processed packets per call and the cache usage were examined. Furthermore, the latency of packet forwarding was also examined. All measurements have been designed to ensure the comparability of the test results: They run on the same CPU, are equipped with the same $10Gps$ NICs and use packet forwarders that apply the same algorithm for each of the three frameworks to allow a fair comparison between them.

The result of the paper shows the strengths and weaknesses of the frameworks. If only performance and latency are considered, DPDK and PFRING seem to be superior to netmap. Though netmap has advantages. It uses well-known OS interfaces and modified system calls for packet IO, while keeping a certain degree of interface continuity and system robustness by performing checks on the user-provided packet buffers. DPDK and PFRING favour more radical approaches by breaking with these concepts, which leads to even higher performance gains. [4]

Comparing the documentation and the community behind the projects, DPDK stands out. For this reason, DPDK is used in this work.

A. DPDK

The Data Plane Development Kit (DPDK) is a high-speed packet IO framework initially developed by Intel. In the meantime, the project has been transferred to the Linux Foundation. This framework contains so-called poll-mode drivers, which are not executed in the kernel as usual. The pollmode drivers are executed in user space. The kernel is first instructed to load the UIO or VDIO driver for the affected network interface card. This is possible during operation and can also be undone. The drivers do nothing but prevent the kernel from initializing the hardware itself and instead allow a program in user space to control all features of the hardware itself. [5]

Figure 4 shows the schematic process of receiving data with DPDK. When a packet arrives at the network card, it is first stored in the RX queues, and then loaded directly into userspace via DMA

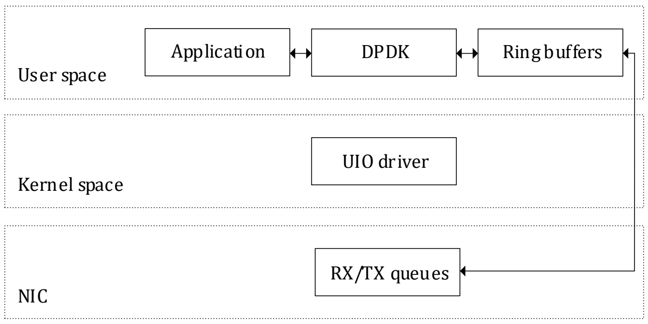

Fig. 4. DPDK structure

without passing through the kernel. For the network interface card it does not matter whether it accesses userspace or kernel memory. The physical addresses are legally accessed via DMA. To enable DPDK to derive DMA descriptors from it, the address space must be mapped. This is done via pagemaps, which translate the virtual addresses into physical addresses. After the data is stored in RAM the Poll Mode Driver of DPDK polls the network card and the main memory locations for new information and manages all functions of the network card. For polling, at least one CPU core must be used permanently, which is used at 100% capacity. By polling, the overhead of an interrupt is avoided, which means a reduction of latency. The DPDK now delivers the data available to the application via an API. Since a network card operated with DPDK, the NIC is no longer visible to the kernel. All packets must be processed within the DPDK application. [6]

IV. Implementation

The following chapter describes the development platform and the implementation and realization of the software. The aim is to implement a high-speed packet IO framework and extract image fragments from the Ethernet packets. The image fragments are then assembled to form an image that has to be transferred to an application for display.

A. Development Platform

An x86 computer with an Intel Core i7-7700K Quad-Core processor ($4.2Ghz$) and $32GB$ $2400MHz$ DDR4-RAM serves as a development platform. Ubuntu 18.04 is used as an operating system. For data reception, an Intel X550 network card is utilized, which provides two $10Gps$ ports. For the high-speed packet IO framework implementation, DPDK version 19.11 is used.

In virtual memory management, which is needed for DPDK, the kernel maintains a table in which it has a mapping of the virtual memory address to a physical address. For every page transaction, the kernel needs to load related mapping. When using small size pages, more page numbers must be loaded, which reduces performance. [7] When using huge pages, the number of mapping tables to be loaded by the kernel is reduced. For this reason, four huge pages are configured with a page size of $1GB$ instead of the standard page size of $4KB$.

Since the Intel X550 network card does not support the UIO driver, the VFIO driver is used, which means that the IOMMU must be activated. The huge pages and the IOMMU configuration are passed as kernel arguments.

B. Flow Bifurcation

By integrating the UIO or VFIO driver, the network interface card is no longer visible in the system. The Flow Bifurcation splits the incoming data traffic to user space applications (such as DPDK applications) and/or kernel space programs (such as the Linux kernel stack). It can direct some traffic, for example UDP traffic, to DPDK, while directing some other traffic, for example ARP requests, to the traditional Linux networking stack.

There are a number of technical options to achieve this. A typical example is to combine the technology of SR-IOV and packet classification filtering. SR-IOV is a PCI standard that allows the same physical adapter to be split as multiple virtual functions. Each virtual function (VF) has separated queues with physical functions (PF) see figure 5. The network adapter will direct traffic to a virtual function with a matching destination MAC address. In a sense, SR-IOV has the capability for queue division. Packet classification filtering is a hardware capability available on most network adapters. Filters can be configured to direct specific flows to a given receive queue by hardware. In this way the Linux networking stack can receive specific traffic through the kernel driver while a DPDK application can receive specific traffic bypassing the Linux kernel by using drivers like VFIO or the DPDK UIO module. [8]

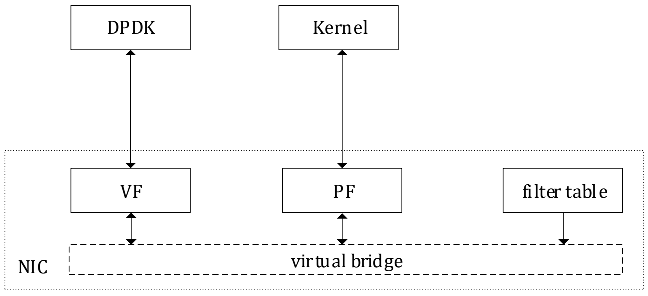

Fig. 5. flow Bifurcation structure

C. Software Structure

Figure 6 shows the structure of the previous implementation. The ImageApp is responsible for displaying the image or video data. This application uses a library called RxAPI, which gets the extracted payload of the Ethernet packets from the Linux network stack. Within the RxAPI, the transferred user data is collected and compiled into an image. The compiled image is then transferred as a structure to the ImageApp, which displays the image on a screen.

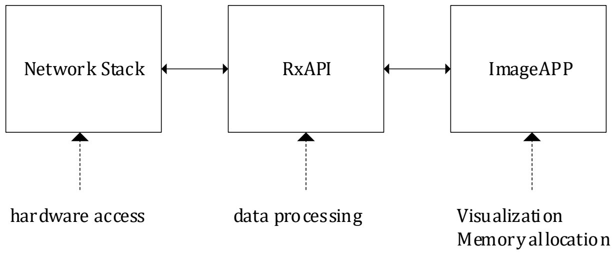

Fig. 6. previous implementation structure with RxAPI

The newly implemented variant is shown in figure 7. Compared to the previous implementation, the Linux network stack has been replaced by the DPDK, which now receives the packets. The received packets are now available in ring buffers. The parsing of the image fragments and the compilation of the image data is realized in the HbcLIB. The HbcLIB has the same interface as the RxAPI, so that the ImageApp can be used for the newly implemented variant without any adaptation. Within the HbcLIB, thread handling is also implemented, which efficiently allocates resources to the CPU cores.

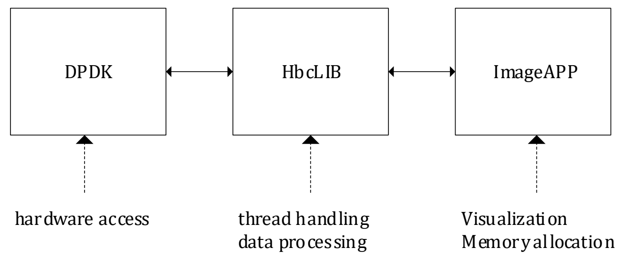

Fig. 7. new implementation structure with HbcLIB

V. Measurement

In this Chapter, the latency of the Linux network stack and the DPDK implementation will be investigated. Only the time needed for packet processing will be considered.

A. Measurement setup

Before the latency measurement can be performed, the test setup must first be determined. For the measurement, a 2.5MP camera is used, which is connected to an MDI via a GMSL-2 interface. The MDI is connected to a Linux PC via the 10Gps Ethernet interface. As measurement software, XTSS from b-plus is used, synchronizing the Ethernet interface of the MDI and the Linux system. On the Linux system, two separate measurements with 100,000 received Ethernet packets each are performed.

First, the conventional implementation with the Linux network stack is examined. For this purpose, the ImageApp is started with the RxAPI. Via XTSS the time stamp is recorded within the network stack when the config packet arrives from the MDI. The RxAPI has been adapted so that an additional timestamp is recorded when the config packet is processed. The processing of the config packet within the RxAPI happens directly before the image data is transferred to the ImageApp and thus offers a comparable basis to the DPDK implementation. The latency is then determined from the difference between the two timestamps.

The second measurement examines the implementation of the DPDK. First, the VFIO Driver must be loaded for the Ethernet interface so that the DPDK can process the packets. Afterward, the ImageApp is started with the HbcLib. Via XTSS the time stamp within the network card is recorded again as soon as the config packet arrives from the MDI. Within the HbcLIB the second timestamp was recorded at the same position as in the RxAPI before the transfer of the image data to the ImageApp. The latency can now be determined again from the difference between the two timestamps.

B. Analysis

With the XTSS tool, the measurement deviation can be determined directly. In both measurements, the inaccuracy was $10\mu s$, which is almost negligible about the measured values.

Table I shows the latency measurement results of the Linux network stack. The high divergence of the min and max values of 13.04ms is particularly noticeable. The reason for this is the implementation of the Linux network stack. On the one hand, when an Ethernet packet is received, a soft interrupt is executed, which is not necessarily processed directly but immediately afterward. On the other hand, the receipt of the packets by the application depends on the load of the operating system, which can increase the deviation of the min and max values.

Table II shows the latency measurement results of the DPDK implementation. The low divergence of 0.36ms is particularly noticeable when compared to the Linux network stack implementation. The measurement deviation of 3% is much higher than in the first measurement, but it is negligible. The convergence of the measurement results can be explained on the one hand by the polling, which can lead to a time offset in the processing of the packets. On the other hand, creating batches of two packets also leads to a time offset.

Comparing the average values of the Linux network stack implementation with the DPDK variant, it is noticeable that the packets within the DPDK implementation are 11.9 times faster than the Linux network stack. This shows that the overhead caused by the interrupt and the copying of data from kernel space to user space leads to a significant increase in latency.

TABLE I
Latency results of Linux Network Stack

Description	measured value
min.	5.89 ms
avg.	12.14 ms
max.	18.93 ms

TABLE II
Latency results of DPDK

Description	measured value
min.	0.53 ms
avg.	0.78 ms
max.	0.89 ms

VI. Conclusion

The implementation of the high-speed packet IO framework DPDK shows a significant improvement in latency. Compared to the classic Linux network stack, however, the setup and implementation of the DPDK represent a considerable development effort, since any protocol must be processed within the framework or the subsequent application.

References

[1] T. Limbrunner, Vorlesung: *Grundlagen der Fahrerassistenzsysteme*, Vorlesung 08 Teil 1 S4, 2020.
[2] *Linux Project Page*, https://www.linux.org/threads/linux-network-stack.9065/ last visited: June 2020
[3] *Mellanox Community*, https://community.mellanox.com/s/article/what-is-rdma-x last visited: June 2020
[4] S. Gallenmüller, P. Emmerich, *Comparison of Frameworks for High-Performance Packet IO*, Munich, Germany: IEEE, 2015.
[5] Linux Foundation DPDK, *Getting Started Guide*, January 2014
[6] D. Scholz, *A Look at Intel's Dataplane Development Kit*, Munich, Germany: Seminars FI / IITM, 2014.
[7] *Kerneltalks*, https://kerneltalks.com/services/what-is-huge-pages-in-linux/ last visited: June 2020
[8] Linux Foundation DPDK, *Programmers Guide*, January 2014