A Look at Intel's Dataplane Development Kit

Dominik Scholz
Supervisors: Daniel Raumer, Florian Wohlfart
Seminar Innovative Internettechnologien und Mobilkommunikation SS 2014
Chair for Network Architectures and Services
Department of Informatics, Technische Universität München
Email: scholzd@in.tum.de

ABSTRACT

The increased performance and cost-efficiency of modern multi-core architectures allows for packet processing implemented in software instead of using dedicated hardware. Such solutions provide large flexibility as features can be added at any time. The drawback of such systems are bottlenecks which appear during the processing of packets at line rate up to 10 Gbit/s. Hence, specialized high-speed packet processing frameworks are needed to use the full potential of commodity hardware. Different research groups have identified several limiting factors and proposed their own solutions by implementing custom drivers and kernel modules. We provide a survey-like introduction to three different frameworks: the Intel Dataplane Development Kit, netmap and PF_RING DNA. To understand the required background of packet processing implemented in software, we explain the mechanisms of modern network adapters and network stacks by reference to Unix-based operating systems. In particular, we present and compare the different techniques that are used by those frameworks to make high-speed packet processing feasible.

Keywords

commodity hardware, highspeed packet processing, Linux network stack, Intel Dataplane Development Kit, netmap, PF_RING DNA

1. INTRODUCTION

Through steady development the performance of commodity hardware has been continuously increased in recent years. Multi-core architectures have therefore become more interesting to fulfil network tasks by using software packet processing systems instead of dedicated hardware. As an example, all Unix-based operating systems can accomplish network tasks, as a full protocol stack that handles the packet processing, is implemented. The advantages of high flexibility and lowered costs are contrasted with possible higher performance and reduced energy consumption achieved with specialized hardware [21, 16].
Unix-based Software routers are a good example of packet processing done using commodity hardware. Features can be rapidly deployed, whereas dedicated hardware would consume extensive development cycles before being able to be upgraded. The RouteBricks project has shown that software routers are in general capable of reaching the same performance that hardware routers deliver today [22].
In order to achieve the line rate of 1 Gbit/s and 10 Gbit/s network adapters, the problems of common packet processing software have to be identified. Various research groups [21, 22, 23] have therefore created models to determine those limitations by running tests for different use cases. They come to the conclusion that standard software is not capable of reaching the maximum performance for small packet sizes. Hence, several projects developed frameworks for high-speed packet processing using commodity hardware [24, 25, 26]. These frameworks replace or extend conventional concepts by implementing their own driver and kernel level improvements. Some of the concepts used by these groups are explained in this paper.
Other research projects created software solutions that specialize on specific tasks like bridging or routing. One example is Open vSwitch [27], an OpenFlow switch that can fully replace a hardware switch. An instance of the latter is Click modular router [28].
The remainder of this work is structured as follows: in chapter 2 we provide the background for packet processing using commodity hardware. In particular, we use the Linux network stack as an example, to highlight the arising problems with high-speed packet processing in common software implementations. Chapter 3 presents the Intel Dataplane Development Kit (DPDK) and explains the main concepts. Then, chapter 4 compares the Intel DPDK with two other packet processing frameworks, netmap and PF_RING DNA. Chapter 5 concludes this paper.

2. PACKET PROCESSING USING COMMODITY HARDWARE

In this chapter we give an overview of a common software implementation for packet processing using commodity hardware. For this purpose we use the network stack of Unix-based operating systems as an example. We will begin with the tasks of the network interface card (NIC) that receives incoming packets, followed by the Linux network stack that implements the necessary dataplane functionalities and conclude with the transmission of the network packet via a NIC. An abstract view of this process is illustrated in figure 1. The resulting performance limiting factors that can be identified during the packet processing of this implementation conclude this chapter.

2.1 Receiving Side NIC

With the arrival of the packet at the receiving NIC (1) two tasks have to be accomplished [2, 3]. Firstly, the packet has to be transferred to main memory (2). Therefore the Linux kernel uses the *sk_buff* structure for the internal representation of packets. The NIC has a buffer, usually a ring

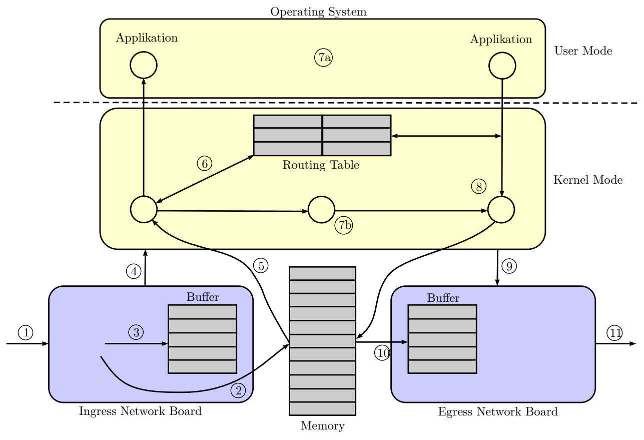


Figure 1: Abstract model of the packet processing steps in Unix-based software routers

queue called 'ring buffer', to store pointers to *sk_buff* structures (3), one for each packet reception and transmission [3]. When receiving a packet, one of these descriptors will be used, assuming that it is ready. This means that the descriptor has to be initialized and allocated with an empty *sk_buff*. To achieve this the direct memory access (DMA) engine of the NIC is used, mapping the structure to kernel memory space. If this was successful the packet is stored in a *sk_buff* and is ready for further processing, otherwise it will be discarded by the NIC, because no packet descriptor was available [4].
The second step is to inform the kernel that a packet is available (4). The NIC schedules a hardware interrupt and eventually the CPU will respond by calling the interrupt handler of the driver [3]. Since the kernel version 2.4.20 the driver uses the New API (NAPI) [2]. The interrupt handler adds the NIC to a so called *poll_list* and schedules a soft interrupt. When this interrupt gets served the CPU polls each of the devices present in the list to get their available packets from the ring buffer. Then, for each of these packets a function is called to start the processing through the network stack [3].

2.2 Linux Network Stack

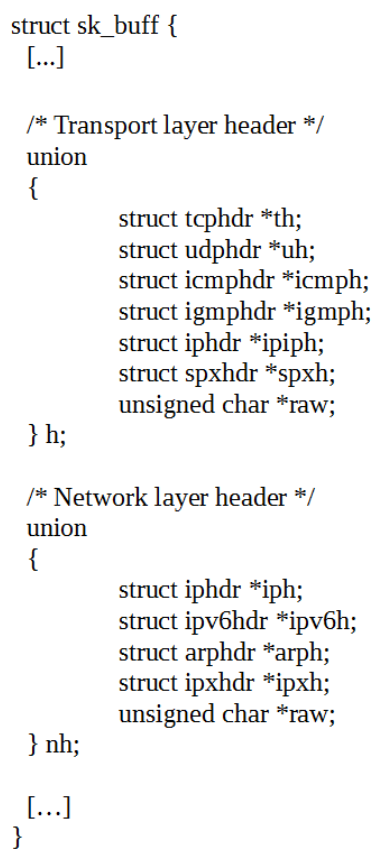
The network stack processes each packet layer by layer [1, 3, 4], starting with the network layer. At first, basic checks, including integrity-verification and application of firewall rules, are performed. If the packet is faulty it gets dropped, otherwise it gets passed to the routing subsystem (6) in order to make a decision whether the packet has to be delivered locally to a userspace application or forwarded to another host. This decision is being made based on the implemented routing algorithm that finds the best match for the destination IP address of the packet in the routing table [1].
If this address equals one of the hosts locally configured addresses the packet has to be delivered locally and is handed over to the transport layer. The Linux network stack provides a complete implementation of the TCP and UDP protocols. A more detailed description of this implementation is given in [3]. After passing the transport layer the packet can finally be delivered to the application. Using the socket API [19] the data of the *sk_buff* gets copied to userspace. Now the application has full access to the data of the received packet (7a). The same way an application in userspace can pass a packet down to the network stack in order to transmit it (8). Here, one of the main tasks is to determine the next hop and corresponding interface for an outgoing packet. This is accomplished by looking up the destination IP address in the routing table [3]. In the case of forwarding the packet (7b), no processing of layers beyond layer 3 is needed. On the network layer the main task is to decrement the TTL header field and, in case it reached zero, to drop the packet and send a corresponding ICMP message [1]. Furthermore, sanity checks, firewall rules and fragmentation can be applied. Afterwards the packet gets passed to the transmitting part of the network stack, too (8).
The layer 2 processing does not differ for forwarded and locally created packets. Based on the result of the routing lookup, the layer 2 address of the next hop has to be identified using for instance the Address Resolution Protocol. After meeting these tasks a function of the NIC is called, informing it to transmit the packet.

2.3 Transmitting NIC

To finally transmit a packet the transmitting NIC has to fulfil two tasks [3]. First of all the driver has to load the packet descriptor, which holds the location of the *sk_buff* in main memory, into the transmitting ring buffer (10). Afterwards he tells the NIC that there are packets available and ready to send (11). Secondly the NIC has to inform the CPU via an interrupt that the *sk_buff* structure can be deallocated.

2.4 Performance Limiting Factors

The Linux network stack is designed for general purpose networking. It is not only able to function as a router, but also supports various protocols on different layers like IPv4 and IPv6 on the network layer or TCP and UDP on the transport layer [3]. While this design choice is convenient for running common applications for "normal" users up to a rate of 1 Gbit/s, it rapidly reaches a limit when approaching the 10 Gbit/s at which the operating system can't handle more packets and thus starts dropping them. The following paragraphs list and explain some of the impediments which are being addressed by the frameworks presented in sections 3 and 4.3 [16].
The identified bottleneck is the CPU. A CPU can only operate a limited number of cycles per second. The more complex the processing of a packet is, the more CPU-cycles are consumed, which then can not be used for other packets. This limits the number of packets per second. Therefore, in order to achieve a higher throughput, the goal must be to reduce the per packet CPU-cycles.
Another limiting factor is the usage of the main memory, which can be further divided into three problems: per-packet allocation and deallocation, complex *sk_buff* data structure and multiple memory copies. The first one occurs right after the packet has been received by the NIC as described in section 2.1. For every packet a *sk_buff* has to be allocated and later on, when the packet has been passed to user-level or has been transmitted, deallocated again. This behaviour leads to excessive consumption of bus-cycles - that is, CPU-cycles spent transferring data from the main memory to the CPU - by the CPU, causing significant overhead. The second problem arises from the effort of the network stack to be compatible with as many protocols as possible. Therefore



```
struct sk_buff {
  [...]

  /* Transport layer header */
  union
  {
            struct tcphdr *th;
            struct udphdr *uh;
            struct icmphdr *icmph;
            struct igmphdr *igmph;
            struct iphdr *ipiph;
            struct spxhdr *spxh;
            unsigned char *raw;
  } h;

  /* Network layer header */
  union
  {
            struct iphdr *iph;
            struct ipv6hdr *ipv6h;
            struct arphdr *arph;
            struct ipxhdr *ipxh;
            unsigned char *raw;
  } nh;

  [...]
}
```

Figure 2: Excerpt of the source-code of the *sk_buff* structure (image from [3])

the *sk_buff* structure (see figure 2) contains the metadata of several protocols, which may not be needed for the processing of the packet or the application. This complexity leads to an unnecessary large data structure, slowing down the processing. The latter problem arises from the fact that a packet has to traverse different stations until it reaches the application, resulting in at least two copy operations of the complete packet: after packet reception the data is copied from the DMA-able memory region to the *sk_buff* and later on it is copied from this buffer to the user-level application. In [16] is shown that 63% of the CPU usage during the processing of a 64 Byte large packet is *sk_buff*-related.
A third major problem is the context switching from user mode to kernel mode and vice versa. Every time the userlevel-application wants to receive or send a packet it has to make a system call, causing a context switch to kernel level (and later on back to user level). While this design improves the resistance to system crashes as the application in user space cannot use critical hardware functions, it consumes a huge part of the CPU-cycles [16].
When using multiple CPUs the packet processing can be slowed down because of so called spinlocks [17]. Those locks are the Linux implementation of active waiting-mutexes and appear on multiple occasions in the network stack. For example, in order to transmit a packet two locks have to be acquired, which protect the access to the NIC's transmitting queue [1], resulting in a bottleneck as possible parallel processing of frames is negated.

All of the mentioned design choices make the Linux network stack a good general purpose networking solution, but not for applications that require high-speed packet processing. Therefore, packet processing frameworks like the ones presented in the remaining chapters of this paper implement techniques that try to resolve most of the mentioned problems.

3. INTEL'S DATAPLANE DEVELOPMENT KIT

Chapter 2 showed that there is indeed need for frameworks that replace or extend the network stack to solve problems like memory allocation per packet or context switches in order to achieve the maximum performance. This chapter will present the Dataplane Development Kit from Intel (DPDK), first released in 2012. First an overview of the framework is given, afterwards the main implemented libraries are explained. This includes the libraries for queue management, memory management and packet buffers. This chapter concludes with an illustration of how to use the DPDK with user-land applications and what the inherited problems are.

3.1 Overview

The Intel Dataplane Development Kit is a framework that provides a set of software libraries and drivers for fast packet processing in dataplane applications on Intel architectures [5, 6, 8]. It supports Intel processors in 32-bit or 64-bit mode from Intel Atom to Intel Xeon generation, with or without non-uniform memory access (NUMA), with no limits to the number of cores or processors. The DPDK runs on Linux operating systems, but completely replaces the network stack, implementing a run-to-completion model: "all resources must be allocated prior to calling data plane applications, running as execution units on logical processing cores" [8]. Furthermore, scheduling is not supported as all devices are accessed via polling. This reduces the overhead produced through interrupt processing in high-speed scenarios [6]. Additionally a pipeline model may be used, which exchanges messages between different cores to perform work in stages.
As shown in figure 3, the DPDK provides the libraries, which are executed completely in userspace, by creating the Environment Abstraction Layer (EAL). The EAL hides the specifics of the available soft- and hardware and provides a interface of library functions that can now be used by ones application.

3.2 Libraries

The Intel DPDK provides several libraries that are optimized for high performance. They fulfil basic tasks, similar to what the Linux network stack does: allocating memory for network packets, buffering packet descriptors in ring-like structures and passing the packets from the NIC to the application (and vice versa). The libraries that are necessary for these assignments include memory, queue and buffer management, which are presented in the following sections. An excerpt of additional features supplied by the Intel DPDK is shown in section 3.2.4.

3.2.1 Queue Management

To manage any sort of queues the *librte_ring* library provides a ring structure called *rte_ring*, shown in figure 4. This struc-

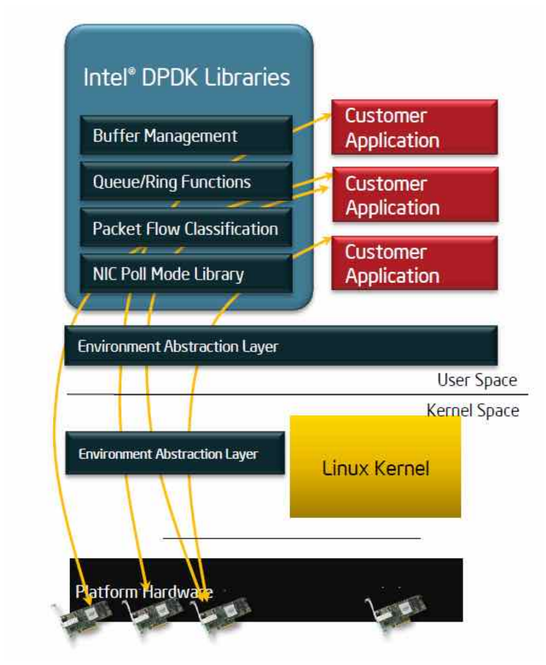


Figure 3: Overview of the Intel DPDK libraries and Environment Abstraction Layer (image from [6])

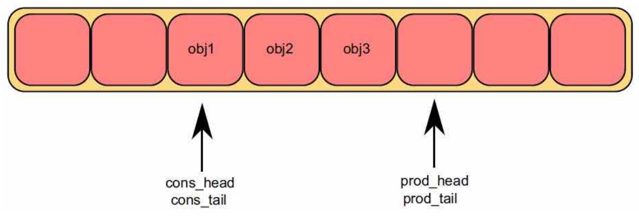


Figure 4: The *rte_ring* structure (image from [8])

ture has the following properties [8]: the ring uses a fixed-size, lockless implementation following the FIFO principle, supporting single and multi producer/consumer en- and dequeue scenarios. Furthermore, it is able to en-/dequeue a specified amount (bulk) or all (burst) available packets. The ring is implemented as a table of pointers to the stored objects and uses two head and tail couples of pointers, one for each the consumers and producers, to control the access. To fully understand the functionality of this structure and why it has the named properties one has to study several scenarios of producer/consumer interaction, therefore the details of a lockless ring-buffer are not shown in this paper (see [20] and [8] for further understanding).
Compared to a normal queue, implemented using a double-linked list of infinite size, the *rte_ring* has two big advantages. First of all it is much faster to write to this structure as the access is protected using lockless mechanisms [20]. Secondly, due to saving the pointer to data in a table, bulk and burst operations cause way less cache misses. Of course the disadvantage is the fixed size, which cannot be increased during runtime and also costs more memory than a linked-list as the ring always contains the maximum number of pointers.
The *rte_ring* can be used for communication, for example between applications, but also to manage the contents of memory pools (section 3.2.2) and to store buffers of network packets (section 3.2.3).

3.2.2 Memory Management

The EAL provides a mapping of physical memory [8]. It creates a table of descriptors which are called *rte_memseg* which each point to a contiguous portion of memory to circumvent the problem that available physical memory can have gaps. Those segments can then be divided into memory zones (figure 5). These zones are the basic memory unit

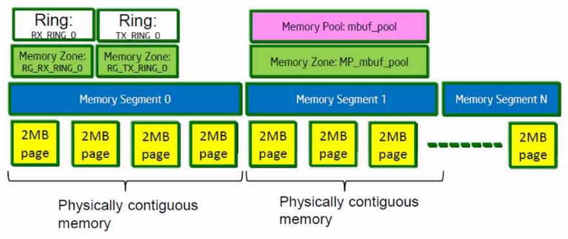

Ring: RX_RING_0
Ring: TX_RING_0
Memory Pool: mbuf_pool
Memory Zone: RG_RX_RING_0
Memory Zone: RG_TX_RING_0
Memory Zone: MP_mbuf_pool
Memory Segment 0
Memory Segment 1
Memory Segment N
2MB page
Physically contiguous memory
Physically contiguous memory

Figure 5: Distribution of physical memory into segments and zones for further use (image from [6])

used by any further object created from an application or other libraries.
One of these objects is the *rte_mempool* provided by the *librte_mempool* library. This structure (see figure 6) is a pool of fixed-sized objects that uses a *rte_ring* to store free objects and can be identified by a unique name [8].
To increase the performance, a specific padding can be

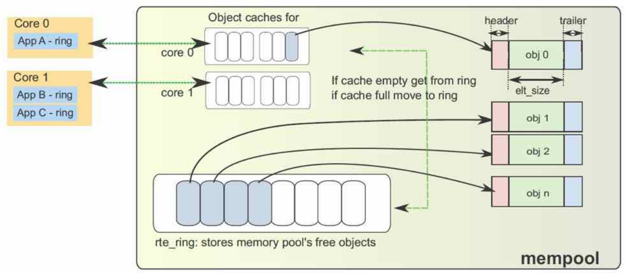


Figure 6: A *rte_mempool* with its associated *rte_ring* (image from [8])

added between objects. This ensures "that the beginning of each object starts on a different channel and rank in memory so that all channels are equally loaded" [8]. Examples, when to use this optimization, include layer 3 forwarding and flow classification of network packets, as only the first 64 bytes are necessary to make a decision [8].
Although a lockless ring is used to control the access to free objects, the costs may be high if multiple cores access the ring causing a compare-and-set operation each time. To avoid having the ring as a bottleneck, each core has an optional local cache in the *rte_mempool*. The core has full access to his cache of free objects, using a locking mechanism. When the cache reaches full capacity or is empty it uses a bulk operation to exchange free objects with the ring of the *rte_mempool*. This way the core has fast access to the

objects he potentially uses repeatedly and relieves the ring structure, which then can do operations with other cores [8].

3.2.3 Network Packet Buffer Management

For any application to be able to transport network packets, the *librte_mbuf* library provides buffers called *rte_mbuf*. These buffers are created before the actual runtime of the application and are stored in a *mempool*. During runtime the user can now use a *rte_mbuf* by specifying a *mempool* from which it wants to take an available one. Freeing a *rte_mbuf* simply means returning it back to the *mempool* it originated from [8].

The size of a *rte_mbuf* is kept as small as possible to fit

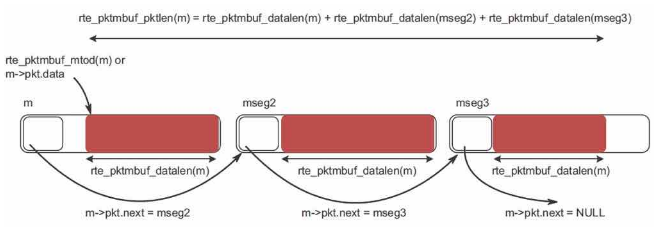

rte_pktmbuf_pktlen(m) = rte_pktmbuf_datalen(m) + rte_pktmbuf_datalen(mseg2) + rte_pktmbuf_datalen(mseg3)
rte_pktmbuf_mtod(m) or m->pkt.data
m
mseg2
mseg3
rte_pktmbuf_datalen(m)
rte_pktmbuf_datalen(m)
rte_pktmbuf_datalen(m)
m->pkt.next = mseg2
m->pkt.next = mseg3
m->pkt.next = NULL

Figure 7: One packet as chain of *rte_mbuf*s. Every *rte_mbuf* contains the same metadata, a pointer to the next buffer and a part of the packet data (image from [8])

in one cache line. In order to hold the information of a larger packet, multiple *rte_mbuf*s can be chained together. As illustrated in figure 7, one *rte_mbuf* does always contain metadata, which is "information [...] retrieved by the network driver [...] to make processing easier" [8], and a part of the data of the packet itself, including header information. It also contains a pointer to another *rte_mbuf* to allow buffer chaining.

This structure can easily be extended to contain other types of information, like serving as a control buffer or for logging reasons.

3.2.4 Further Libraries and Features

Aside from the already illustrated libraries which are needed to allow basic networking operations, the Intel DPDK also has further libraries and drivers that provide specialized functionalities.

The LPM library implements a table that allows longest prefix matching (LPM), which is an algorithm used to forward packets depending on their IPv4 address [1]. The main functions are to add or delete a rule to or from the table and lookup an IP address using the LPM algorithm. The best fitting entry is then returned. Based on this library an application can be written to fulfil IPv4 forwarding tasks. The same functionalities but for the IPv6 protocol are provided by the LPM6 library [8].

Another library offers similar functionalities by providing hash functions. The hash library allows to do efficient and fast lookups in a big set of entries using a unique key[8]. This can then be used for flow classification, that is, mapping a packet to the connection, called flow, it belongs to. This is useful to match for instance TCP packets to their connection and process all the frames together in the context of this flow. A common procedure to achieve this matching is to create a hash value of the 5-tuple of the IP- and transport-layer headers (source and destination address, protocol, source and destination port) [8].

The timer library provides the ability for functions to be executed asynchronously, which can be used for example for garbage collectors. The timer can be executed with high precision only once or periodically. It uses an interface of the EAL or the CPU's Time Stamp Counter to receive a precise time reference [8].

The DPDK also includes poll mode drivers for 1 Gbit/s and 10 Gbit/s network cards and several other emulated frameworks like virtio or VMXNET3 adapter [8]. These provide APIs to directly configure the device and access the receiving and transmitting queues without the need for interrupts, comparable to the NAPI drivers.

Several more specialised libraries are also provided. See [8] for a full list and detailed descriptions.

3.3 Using the Dataplane Development Kit

The Intel DPDK provides all its libraries by creating the Environment Abstraction Layer (EAL) [5, 7]. The EAL has access to low-level resources including hardware and memory space and provides a generic programming interface that is optimized for the hardware of the system, but hides the environment specifics from the applications and libraries. The supplied services include the loading and launching of applications, reserving memory space in form of memory segments as described in section 3.2.2 and trace and debugging functions for instance for the logging of events [8].

Before being able to compile an application the Intel DPDK target environment directory, which contains all libraries and header files, has to be created. To compile an application two variables must be exported: *RTE_SDK* pointing to the installation directory of the DPDK and *RTE_TARGET* pointing to the environment directory. The compiled program can now be run by linking it with the EAL [7].

One has to keep in mind that the Intel DPDK only provides a framework for basic operations that allow packet processing at high speed. It does not provide any functionalities like layer three IP forwarding, the use of firewalls or any protocols like TCP or UDP that are fully supported by the Linux network stack. It is the programmers responsibility to build all the needed functionalities using the provided libraries. Thus, porting an application to the Intel DPDK can be time consuming.

However, the development of applications is eased as many example programs, illustrating the use of different libraries or features, are provided. Furthermore, the *dpdk.org*-project tracks the Intel DPDK by not only listing recent patches, but also contributing major enhancements provided through the community [5]. The DPDK vSwitch [9], an improved version of OpenvSwitch, is an example of an application that is based on the Intel DPDK and achieves significant performance boosts [10].

4. OTHER FRAMEWORKS

As discussed in chapter 3 the Intel DPDK is a framework to speed up packet processing by completely replacing the network stack on Unix-based operating systems and implementing different techniques like buffer pre-allocation or a custom packet buffer. Other frameworks with similar purposes have been developed over the last few years, sharing some of the approaches used in the Intel DPDK, but also using different ideas. This chapter will present two selected projects by shortly explaining the main concepts and techniques, without getting into the details of their respective implementations. Firstly, the netmap framework for fast packet I/O will be introduced, followed by an engine called PF_RING DNA. This chapter will conclude with a comparison of the three presented packet processing frameworks.

4.1 netmap

Netmap is a framework developed at the University of Pisa by Luigi Rizzo within "The netmap project" [11]. It is designed for fast packet I/O in a safe manner, implemented as a kernel module, thus, running completely in kernel mode. Netmap implements techniques to resolve three problems of packet processing: the dynamic allocation of memory per packet, overhead produced through system calls and multiple memory copies [12].
The data structures of the netmap architecture reside in the same memory region (illustrated in figure 8) which is shared by all user processes. This makes the forwarding of packets

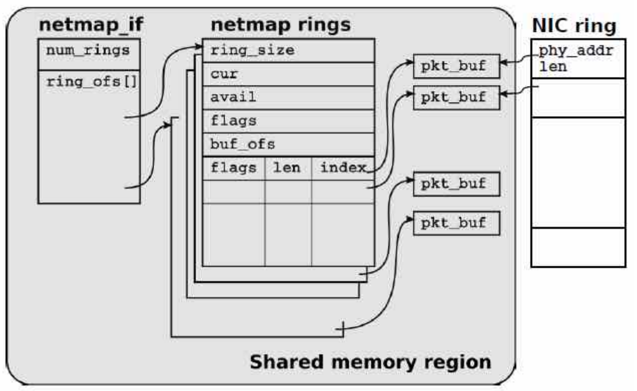


Figure 8: The structures of the netmap architecture (image from [12])

between different interfaces possible without a single copy operation. Furthermore, netmap uses its own lightweight representation of packet buffers [12]. Those *pkt_buf*-structures are fixed-sized and are preallocated before the actual packet processing begins. Each packet buffer is referenced by the ring of the corresponding NIC. Additionally, *netmap rings* are used to keep references to the buffers. This is a circular queue that uses an array to store the indices and other information of the packets. The last data structure, called *netmap_if*, contains important information, like the number of rings, about an interface. To reduce per-packet processing costs, netmap provides batching techniques. Also, a method called parallel direct paths is utilized to solve the problem of serialization during the access of traffic [16]. Basically, a direct path from a receiving or transmitting queue of the NIC to the application in user space is constructed and being assigned to only one core. This supports Receive Side Scaling (RSS) [18], a technique to distribute incoming network traffic to multiple CPUs. RSS is supported by many network cards as it "relieve[s] bottlenecks in receive interrupt processing caused by overloading a single CPU" [18].
Despite using shared memory regions and keeping the structures in kernel context, a misbehaving user application cannot cause the kernel to crash [12]. However, processes can corrupt each others netmap rings and packet buffers. A possible solution for this problem is to distribute each ring to a separate memory region.
The netmap framework supports a wide range of soft- and hardware. Initially it was developed for FreeBSD operating systems, but a Linux-version has been completed [11]. With regard to NICs, 10 Gbit/s adapters using ixgbe-driver or 1 Gbit/s adapters from Intel, RealTek and nVidia are supported. We found no documented restrictions for the usage of CPUs from different manufacturers [16].
Netmap uses standard system calls like `poll()` or `select()` for synchronization and other purposes [12]. While this reduces the probability of the system to crash, the performance may be reduced by additional context switches as described in section 2.4.
To ease the employment of netmap, a library that maps libpcap calls into netmap calls is provided. Furthermore, some functionalities like IP-forwarding are already implemented [16]. Notable examples of applications that added support for netmap are Click [28] and VALE [29].

4.2 PF_RING (DNA)

PF_RING is a network socket for capturing, filtering and analysing packets at high-speed, developed by ntop [14]. It is a kernel module that has the structure shown in figure 9. PF_RING uses the Linux NAPI drivers to poll packets from

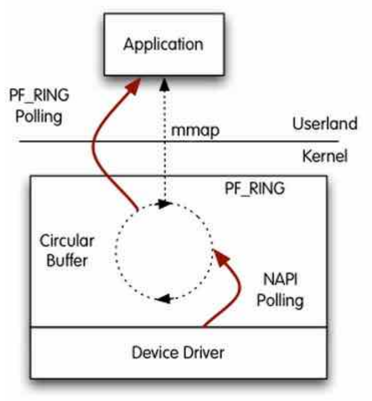


Figure 9: Vanilla PF_RING (image from [14])

the NIC and copy them into the PF_RING circular buffer. Using another polling mechanism the application in user-mode can read the packets from this ring. While two polling operations lead to increased CPU cycles spend per packet, the advantage lies in that "PF_RING can distribute packets to multiple rings, hence multiple applications, simultaneously" [14]. Furthermore, PF_RING implements memory pre-allocation for packet buffers, creation of parallel direct paths and memory mapping techniques [16].
A special feature comes with the PF_RING Direct NIC Access module [15]. This is a special driver to directly map NIC memory and registers to userspace, resulting in no copy operations besides the initial DMA transfer that is done by the NIC at packet arrival. Therefore, no interaction of the kernel takes place, apart from the polling operation of the application that gets the packets from the NIC memory and stores them into preallocated packet buffers in

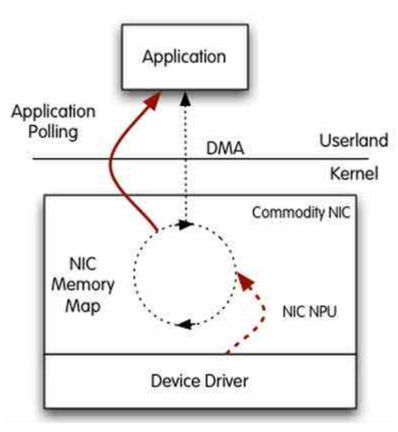


Figure 10: PF_RING Direct NIC Access driver (image from [14])

user space [13]. This behaviour is demonstrated in figure 10. While this allows for pure speed as the CPU only has to do one polling operation, it has the drawback that only one application at a time can access the DMA ring, thus, read packets from the ring. Furthermore, if the application misbehaves while accessing the memory and registers of the NIC, the whole system might crash [16].
PF_RING and all its modules are available for Linux kernels 2.6.32 and newer and support various 1 Gbit/s and 10 Gbit/s NICs from Intel. A wrapping of the PF_RING API that is similar to the libpcap library is also provided to allow flexibility and ease of use. On top of that, various sample programs for traffic monitoring, capturing and other purposes exist. One example is *pfsend*, a high-speed traffic generator.

4.3 Comparison: Intel DPDK, netmap and PF_RING DNA

The three presented frameworks show that different approaches to high speed packet processing can be considered. This begins with the decision whether to fully shift the processing to user space like it the Intel DPDK implements or use kernel modules instead. However, all frameworks tend to implement similar techniques in order to solve certain core problems which can be identified for example by having a closer look at the Linux network stack. A summary of those is given in table 1. All of the presented frameworks use their own structure to store the network packets. However, unlike the Linux network stack, they don't differentiate between packet data and several layers of headers, hence, reducing overhead caused by additional metadata. Packet buffers are always preallocated during start-up of the runtime and later on re-used, whereas the access to them is controlled using a ring structure. To reduce copy operations, memory mapping techniques are used, whereby PF_RING DNA brings it a step further in granting the application full access to the NIC's memory and registers. Unfortunately this comes at the cost of being prone to errors which may result in system crashes. As high reliability can be of great importance, one has to be particularly careful in the development of an application. Receive Side Scaling is supported by all three frameworks by creating parallel direct paths, allowing to distribute the load over all cores. One disadvantage that is difficult to get around, is the inability of PF_RING DNA

	DPDK	netmap	PF_RING DNA
Memory pre-allocation	✓	✓	✓
Custom network buffer structure	✓	✓	✓
Memory mapping	✓	✓	✓
Batch processing	✓	✓	✗
Parallel direct paths	✓	✓	✓
Not prone to architecture-related system crashes	✓	✓	✗
Use of standard system calls	✗	✓	✗

Table 1: Implemented techniques and features of different frameworks (table from [16], modified)

for batch processing of packets. If this feature is vital for an application to achieve high performance, one is advised to use another framework instead. The same holds true for netmap, as common system calls are used, that may lead to reduced performance caused by excessive context switching. One general problem of packet processing frameworks is that they are often limited to specific hardware and operating systems. Only if the required CPUs and NICs are present, specialised frameworks like the Intel DPDK or PF_RING (DNA) can be used, whereas netmap supports a broader range. Furthermore, all frameworks provide a different, yet libpcap-like, programming interface. Also, one has to keep in mind that only basic packet processing functions are delivered, whereas the user has to port any application to the chosen framework first. Example programs or libraries that provide additional functionalities are either provided by the developers themselves or have been created by communities over the past few years. Hence, one should look out for similar programs before writing an application from scratch.
As we found no work that tested the Intel DPDK, netmap and PF_RING DNA under same conditions, it is hard to compare the three frameworks regarding their achieved performance. However, tests from Intel using the Intel DPDK [6], [12] using netmap and [16] using PF_RING DNA (and several other packet processing frameworks) show that they are able to achieve the line rate of 10 Gbit/s even with 64 byte packets. Of course, this depends on the used hardware and can be different for other applications and use cases. To determine which framework is suited best for which application, generalized tests under equivalent conditions (for instance equal hardware) have to be made.

5. CONCLUSION

In this paper we described the problems of high-speed packet processing using modern commodity hardware. Using the Linux network stack as an example, several performance limiting factors have been identified. The basic functionalities of the Intel Dataplane Development Kit have been explained in detail. Furthermore, this framework has been compared with two other projects, netmap and PF_RING DNA, to illustrate some techniques that solve the described problems.
Commodity hardware is a valid alternative to packet processing using dedicated hardware as the software is always configurable to suit the current problem in the best possi-

ble way. Several solutions have been created over the past few years to achieve the best performance that the hardware allows. While the presented frameworks differ in their respective implementations, they all try to resolve the core problems of existing, widely distributed packet processing solutions: per-packet allocation and deallocation, multiple memory copies, restrictions to parallel processing through locking mechanisms and context switching.
One has to keep in mind, that such frameworks are not designed to fulfil everyday general purpose networking tasks. Widely distributed and established solutions like the Linux network stack will retain the upper hand in those scenarios, as they support for instance a broad range of protocols. When it comes to high performance packet processing, the Intel DPDK, netmap and PF_RING DNA have the clear advantage. They are all capable of reaching the line rate of 10 Gbit/s for a specific task. The first applications like the DPDK vSwitch or VALE are becoming more and more popular as they function reliable at high speed.
However, packet processing frameworks like the Intel Dataplane Development Kit or PF_RING only support selected hard- and software. Furthermore, new problems arise for the end-user that has to write an application, as all frameworks provide different APIs. As some of those projects have dedicated communities like *dpdk.org*, the problem of porting applications to a specific framework can be eased significantly. Therefore, packet processing frameworks and commodity hardware are the way to go for the future, because hardware is constantly evolving, allowing for increased performance. The same trend must be maintained by the respective sophisticated software to become widely distributed and, thus, the de facto standard for packet processing.

6. REFERENCES

[1] C. Benvenuti: *Understanding Linux Network Internals*, O'Reilly Media, Inc., Volume 1, 2005

[2] R. Rosen: *Linux Kernel Networking: Implementation and Theory*, Apress, Volume 1, 2013

[3] M. Rio et al.: *A Map of the Networking Code in Linux Kernel 2.4.20*, in: Technical Report DataTAG-2004-1, March 2004

[4] W. Wu et al.: *The Performance Analysis of Linux Networking - Packet Receiving*, in: International Journal of Computer Communications, 30(5), pages 1044-1057, 2007

[5] *Intel DPDK: Data Plane Development Kit Project Page*, `http://www.dpdk.org`, last visited: June 2014

[6] Intel DPDK: *Packet Processing on Intel Architecture*, Presentation slides, 2012

[7] Intel DPDK: *Getting Started Guide*, January 2014

[8] Intel DPDK: *Programmers Guide*, January 2014

[9] Intel Open Source Technology Center: *Packet Processing*, `https://01.org/packet-processing`, last visited: July 2014

[10] Intel Open Source Technology Center: *Packet Processing - Intel DPDK vSwitch*, `https://01.org/packet-processing/intel%C2%AE-onp-servers`, last visited: July 2014

[11] *The netmap Project*, `http://info.iet.unipi.it/~luigi/netmap/`, last visited: July 2014

[12] L. Rizzo: *netmap:a novel framework for fast packet I/O*, in: Proceedings of the 2012 USENIX Annual Technical Conference, pages 101-112, 2012

[13] L. Deri: *Modern packet capture and analysis: Multi-core, multi-gigabit, and beyond*, in: the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM), 2009

[14] ntop: *PF_RING Project Page*, `http://www.ntop.org/products/pf_ring/`, last visited 10.06.2014

[15] ntop: *PF_RING DNA Project Page*, `http://www.ntop.org/products/pf_ring/dna/`, last visited: June 2014

[16] J. García-Dorado et al.: *High-Performance Network Traffic Processing Systems Using Commodity Hardware*, in: Data Traffic Monitoring and Analysis, Springer Verlag, pages 3-27, 2013

[17] *Wikipedia: spin-lock*, `http://goo.gl/f1Dqhx`, last visited: June 2014

[18] Red Hat Enterprise Linux 6: *Performance Tuning Guide*, Chapter 8.6 Receive-Side Scaling (RSS), `http://goo.gl/qhGQYT`, last visited July 2014

[19] *Linux man page: socket*, `http://linux.die.net/man/7/socket`, last visited: July 2014

[20] lwn.net *A lockless ring-buffer*, `http://lwn.net/Articles/340400/`, last visited: June 2014

[21] G. Carle et al.: *Measurement and Simulation of High-Performance Packet Processing in Software Routers*, in: Proceedings of Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und Verteilten Systemen, 7. GI/ITG-Workshop MMBnet 2013, Hamburg, Germany, September 2013

[22] M. Dobrescu at al.: *RouteBricks: Exploiting Parallelism To Scale Software Routers*, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 15-28, October 2009

[23] R. Bolla et al.: *Linux Software Router: Data Plane Optimization and Performance Evaluation*, in: Journal of Networks, 2(3), pages 6-17, 2007

[24] S. Han et al.: *PacketShader: a GPU-accelerated software router*, in: ACM SIGCOMM Computer Communication Review, 40(4), pages 195-206, 2010

[25] F. Fusco: *High speed network traffic analysis with commodity multi-core systems*, in: Proceedings of ACM Internet Measurement Conference, pages 218-224, 2010

[26] N. Bonelli: *On multi-gigabit packet capturing with multi-core commodity hardware*, in: Proceedings of Passive and Active Measurement Conference, pages 64-73, 2012

[27] *Open vSwitch*, `http://openvswitch.org/`, last visited: June 2014

[28] R. Morris et al.: *The Click modular router*, in: ACM Transactions on Computer Systems (TOCS), 18(3), pages 263-297, 2000

[29] L. Rizzo and G. Lettieri: *VALE, a switched ethernet for virtual machines*, in: CoNEXT, pages 61-72, 2012.