A Survey of Trends in Fast Packet Processing

Konstantina Tsiamoura
Betreuer: Florian Wohlfart, Daniel G. Raumer
Seminar Future Internet SS2014
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: k.tsiamoura@tum.de

ABSTRACT
Network interfaces of 10 Gbit/s are used more and more. Despite their high performance, packet processing rates are not correspondingly higher. This happens due to the architecture of the network stacks and the design of device drivers. There are restrictive factors and overheads which need to be overcome, so as to enable fast packet processing with commodity hardware. In this paper, techniques towards this direction are discussed and existing frameworks which exploit combinations of them are presented and compared. The frameworks mentioned are divided into two main categories, depending on whether they involve GPU for the packet processing or not.

Keywords
fast packet processing, GPU packet processing, zero-copy techniques

1. INTRODUCTION
Network interfaces have increased their performance and 10 Gbit/s rates tend to be nowadays commonplace. However, the increased performance of these network interfaces cannot by itself guarantee packet processing at high speeds. There are overheads imposed by the network stacks' architectural design which was meant to be used with general-purpose hardware. It emphasizes on compatibility and overlooks performance [3]. Packet processing can become more efficient by alternating the network stack and without necessitating more powerful hardware [15]. This fact has led to surveys concerning how general-purpose hardware and fast network interfaces can be effective regarding packet processing. 10 Gbit/s entail a considerable high amount of packets per second [16], which demands intensive CPU usage, and the support of packet processing at high rates without packet losses.

In order to understand which techniques would be beneficial to making general-purpose hardware suitable for high speed network applications, an understanding of its current restrictive parameters is required. These factors refer to hardware resources limitations and the way that packet processing is organized based on the network stack. Afterwards, the presentation of widely suggested techniques which target to avoid or decrease the problems resulting from these factors is meaningful.

Two of the proposed techniques hold a special position. The first one is zero-copy technique, which eliminates the overhead of the data copy between kernel and user-space. It is implemented by the majority of the existing frameworks for fast packet processing. The second technique is more recent and emerging and refers to the exploitation of GPU to perform parts of the packet processing.

Frameworks aiming to enable fast packet processing are presented as well as the way the proposed techniques can be put into practice.

The rest of the paper is organized as follows: In Section 2, the state-of-the art New API (NAPI) and the Click modular router are presented. Section 3 provides an overview of the factors that restrict the performance of general-purpose hardware towards packet processing in combination with recommended solutions to them. Section 4 is about zero-copy technique and frameworks that make use of them in order to enable a faster packet processing. In Section 5, Netslice is presented; a framework that uses neither zero-copy technique, nor GPU for the packet processing. Frameworks that exploit GPU processing are presented in Section 6. Finally, in Section 7 a comparison between the frameworks, in terms of the techniques they use, takes place.

2. STATE OF THE ART

In this section the state-of-the art in packet processing is presented, by the description of the Linux IP stack and its NAPI, and the Click Modular Router. This section provides also some essential background for network stacks and the packet flow through them during packet processing.

2.1 LINUX NETWORK STACK AND THE NEW API (NAPI)
The Linux network stack is introduced by the description of the packet flow which takes place when a packet arrives. This packet flow is represented in figure 1.

The packet arrives at one of the circular receiving queues (RX) of the Network Interface Controller (NIC), which are also called rings. There, the packet is stored in a data structure, the receiver descriptor, which enables the copying of data between the NIC and the main memory. The data transfer is achieved via the Direct Memory Access (DMA) mechanism, which copies the data to the DMA-able memory region without the CPU. At this point a mechanism is needed to inform the system that a packet has been received so as to perform the data transfer between the

doi: 10.2313/NET-2014-08-1_06

DMA-able memory region and the packet buffer that Linux kernel allocates for each packet. For that purpose, in Linux network stack, an interrupt-based mechanism is used and an interrupt is raised at every packet's arrival. Then the packet has to be copied from the kernel to user-space [3].

The interrupt mechanism is not suitable for packets arrival at high rates, since it causes the receive or interrupt livelock phenomenon [2]. This phenomenon results in reducing the system's performance because all resources are devoted to interrupts handling and not to real packet processing related tasks [12, 5].

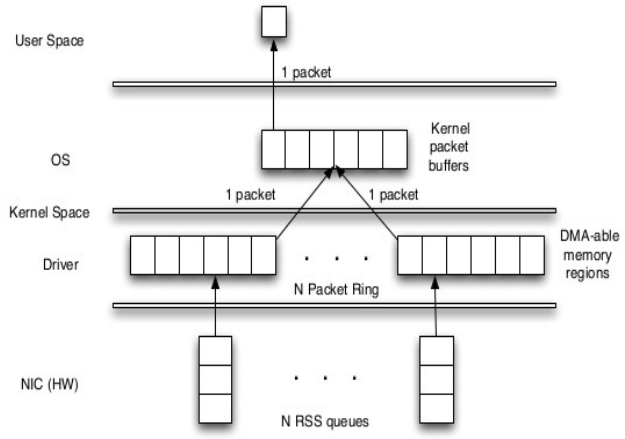

Figure 1: Linux Network Stack [3]

NAPI was introduced in the initial Linux network stack in order to overcome the receive livelock problem, by using a polling mechanism in addition to the interrupts, to exploit their benefits at high and low speeds of packet arrival respectively. NAPI can interchange these mechanisms dynamically. Interrupts are only enabled for the first packet of a batch. Then a polling mechanism gets periodically enabled, to inspect the devices that have received packets that want to forward to the network stack [19]. These packets are temporarily stored to the DMA-able memory region, waiting for the CPU to become available. The interrupts mechanism can be activated again on the condition that there are no more available packets in the DMA-able memory [3]. This hybrid mechanism ensures that when the traffic is low, there will be no increased latency due to the interrupts. In addition, when there is packet arrival at high rates the polling mechanism ensures: 1) reduced cache misses, 2) reduced I/O overhead, and 3) better CPU throughput [20].

2.2 Click Modular Router

Click modular router constitutes a framework for creating modifiable and flexible routers. Click, which can be used instead of the Linux IP stack [11], can be executed in kernel as well as in the user-space. Similar to NAPI, Click uses both interrupts and polling mechanism [1].

With Click, it is feasible to develop most of the packet processing software by putting together elements in a pipeline structure [23]. Click's simple architecture is enhanced due to the pull processing and the flow-based router context [13].

The components of the Click are called elements. They are individual units related to routing and forwarding processing, for instance filters and queues [11], which can incrementally implement a router configuration. A router configuration is defined as a graph, where nodes represent the elements, and edges represent the direction of the packets as they are moving through the elements via their input and output ports. The ports are responsible for establishing the connection between the elements.

The pull processing, as well as the push processing, are two functions that determine the way in which the elements communicate with each other and which entity causes a packet to travel from one element to another. According to the push processing, the sending element delivers a packet to a receiving element, while in the pull processing, it is the receiving element which triggers the move of a packet by asking it from the corresponding sending one. Sending and receiving elements are also referred to as upstream element, which are found at the top of the pipeline, and downstream element, which are found at the bottom of the pipeline structure respectively [23]. An example of a Click router configuration is depicted in figure 2. The black and the white ports are used to represent push and pull ports respectively. Queues are distinct entities in Click and are represented by a distinct element, the *Queue* element. Such an element is the second element of the router configuration.

Figure 2: Example of a Click router configuration [13]

The second important trait of the Click Modular Router is the flow-based router context. It is information available to an element, concerning the path that a packet needs to follow. With this information, the element can identify all the other elements which constitute the flow of the packet and not only these with which it has an immediate link.

3. FAST PACKET PROCESSING TECHNIQUES

Communication links have reached and exceeded the rate of 10 Gbit/s. This fact raises the question, how can general-purpose systems support the huge number of packets that 10 Gbit/s link entail [16]. Therefore, solutions that can achieve fast packet processing are needed.

3.1 Fast Packet Processing Impediments

In [20] the following parameters are considered to affect the performance of packet processing applications: 1) CPU speed and inadequate utilization, 2) interrupts overhead, 3) limited bus bandwidth in comparison to a fast processing unit, 4) memory latency, 5) I/O latency.

doi: 10.2313/NET-2014-08-1_06

In [3], there is a more detailed description of the factors related to the first parameter mentioned above, meaning the CPU, that restrict high speed packet processing. More specifically these overheads originate from: 1) performing memory allocation and deallocation for each packet (per-packet memory allocation), 2) overhead of copying data between kernel and user-space, 3) expensive cache misses, 4) per-packet system calls, because of the CPU-intensive context switch between kernel and user-space, and 5) the transformation of the parallelized processing of packets by the queues of multi-queue NICs to a serialized one. This happens because all packets have to converge to one single point, thus creating a bottleneck.

3.2 Techniques against Fast Packet Processing Impediments

In [3], techniques that have been proposed as a countermeasure against the limitations and overheads of packet processing have been presented. Table 1 associates the problems discussed in Section 3.1 with their proposed solution. These techniques are widely adopted by the frameworks that are going to be presented in the following sections.

Table 1. Solutions for the CPU-related problems

Problem	Solution
Per-packet memory allocation	Memory pre-allocation
Copy of data between kernel and user-space	Zero-copy techniques
Cache misses	Memory affinity, prefetching
System calls	Batch packet processing to reduce system calls
Not parallel packet processing to the whole network stack	Direct paths from NICs to user-space that can support parallelism

4. FRAMEWORKS SUPPORTING ZERO-COPY TECHNIQUES

4.1 Zero-copy technique

As it has been already mentioned, one significant overhead of the packet processing is caused because of the data copies between kernel and user-space. To avoid data copies, a zero-copy technique needs to be applied. The zero-copy technique actually refers to a collection of techniques that reduce the copies between either kernel and devices or kernel and user-space [22].

4.2 Netmap

Netmap [16] is a framework that performs packet processing at high speeds by reducing the avoidable costs that slow the processing down. It enables the applications to gain fast access to the packets. Netmap is independent of specific devices and hardware, and is built upon existing operating system characteristics and applications [15].

In the figure 3, the Netmap architecture is presented. In particular the data structures that Netmap incorporates in its

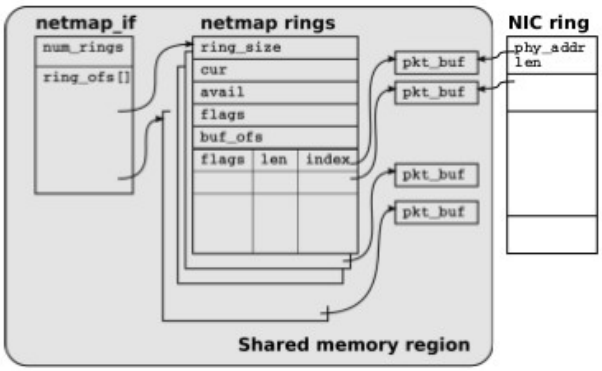

Figure 3: Netmap Data Structure [14]

architecture are depicted, namely packet-buffers, netmap rings and netmap_if. These data structures are all located to a shared memory region which can be accessed both by kernel and user-space applications, so as to avoid copying data. The shared memory region is mapped to all user processes address space via the system call *mmap()* [17]. It is important that due to the contents of the shared memory region, an application running on Netmap cannot lead to a kernel crash.

Packet-buffers: The packet buffers of Netmap have a predetermined size and are preallocated. This reduces the cost that per-packet allocations and deallocations entail. The size of the buffer is sufficient for storing packets without fragmentation. The packet buffers are used by both netmap and NICs' rings.

Netmap rings: They are circular queues that contain metadata related to the buffer, similar to NICs rings. They contain information like the count of data entries, *slots*, that the ring can store, the count of available buffers and an array of slots. Netmap decouples the NIC rings from the network stack, and it allows user-space applications to communicate with them separately and for different reasons through the netmap rings. More specifically, Netmap exploits the speed with which a NIC can transfer packets between the network and the memory. Synchronization can be achieved through operating system functions like *select()* and *poll()*.

Netmap_if: The netmap_if data structure includes information related to the interface, like how many rings there are.

The presented data structures offer the following advantages: 1) reduced per-packet cost, 2) efficient communication between interfaces and between NIC and network stack and 3) potential use of multi-queue NICs.

4.3 RF_Ring

PF_Ring [14] is a framework that provides fast packet capturing and processing. PF_Ring implements zero-copy by avoiding data copy between the kernel and the user-space. It achieves that with the use of packet buffers that are found in a memory region common to the kernel and the users' applications [16]. The

doi: 10.2313/NET-2014-08-1_06

packet buffers are preallocated. Due to this packet buffer schema, it avoids the cost of per-packet memory allocation and deallocation. The basic elements that compose PF_Ring's architecture are:

PF_Ring kernel module: It is responsible for copying the packets to the PF_Ring circular queues.

PF_Ring user-space library: It is through this library that PF_Ring kernel module is exposed to user-space applications.

PF_Ring aware drivers: Although PF_Ring is NIC independent, which means that it is not based on specific NICs, it can be used with specialized drivers too and gain more in terms of performance.

4.3.1 PF_Ring DNA (Direct NIC Access)

The PF_Ring can be used with a specialized type of device driver, namely DNA, for achieving even faster processing without the intervention of CPU and the use of system calls [18]. PF_Ring provides a mapping between the NIC memory and the user-space memory and permits the explicit communication between applications and NICs. This can be seen in figure 4. The packets are transferred between the NIC and the user-space, without the intervention of either Linux kernel or PF_Ring module. In this case, there is only the data copying performed by the NIC Network Process Unit (NPU) via DMA, while the copy from the kernel packet buffer is omitted. This is referred to, as full zero-copy [3].

Besides the high speed performance that can be achieved by the PF_Ring DNA, this approach has two important disadvantages. First, it entails the risk of misusing memory addresses via the NIC's DMA engine and thus leading to a possible system crash [16]. Second, its use is limited to one application at a time. [14]

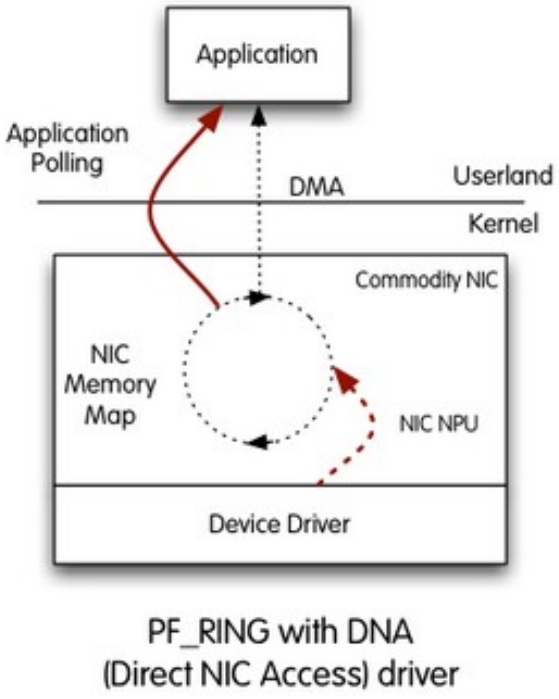

Figure 4: PF_Ring DNA Architecture [6]

4.4 INTEL Data Plane Development Kit

The Intel DPDK [7] is a framework that provides fast-packet processing to applications related to the data plane part, meaning to those applications that are responsible for forwarding packets to their applications [21].

The framework supports two models for processing the packets. It can implement either a run to completion model or a pipeline one. In the run to completion model, every processing unit is allocated to one packet, which it completely processes. All the processing units execute a common application code. Contrary to this, the processing units of the pipeline execute different application code, since they perform a distinct task of the packet processing. A packet is not entirely processed by a single thread, and it has to be transferred through every processing unit of the pipeline.

4.4.1 Intel DPDK Architecture

Environment Abstraction Layer (EAL): The framework provides libraries for particular environments. It also implements the Environment Abstraction Layer in order not to expose details concerning each environment to the libraries and to the data plane applications that will use them. Through the interface provided by the EAL, applications can gain access to hardware related resources.

More specifically, the EAL 1) allocates memory, 2) devotes cores to execution units, 3) communicates with the PCI bus, and 4) inspects the existence of interrupts.

doi: 10.2313/NET-2014-08-1_06

Core Components: The core components define the libraries offered by Intel DPDK for achieving fast packet processing.

Table 2. Intel DPDK's core components

library	Description
Memory Manager	Services for memory allocation
Ring Manager	Defines the Ring data structure for storing the packets
Memory Pool Manager	Allocates portions of memory of specific length and stores object by using rings
Network Packet Buffer	Manages packet buffers
Timer Manager	Services for scheduling functions

4.4.2 Intel DPDK Fast Packet Processing Techniques

The Intel DPDK manages to perform packet processing at high speeds due to the following characteristics:

- The ring data structure, which is implemented as queue, provides faster access and storage than a linked list. Besides, it diminishes the time needed for time-consuming bulk operations.

- It uses pre-allocated buffers. In fact, there is only one buffer for keeping both metadata information and packet data and so there is only one memory allocation per-packet.

- It enables the cores to have their own cache memory and limits their accesses to the shared ring of a memory pool, which can be CPU-intensive.

- It reduces interrupts overhead with the use of the Poll Mode Drivers.

5. NETSLICE

In this section, NetSlice [10] is presented. Contrary to the solutions of fast packet processing presented in the previous section, NetSlice does not take advantage of zero-copy techniques.

NetSlice is an operating system abstraction that attempts to provide high-speed packet processing and runs in the user-space. More specifically, it aims to reconcile the benefits of the packet processing applications that run in the user-space, i.e fault isolation and programmability, with the performance of 10 Gbit/s of current NICs. NetSlice is based on the high coupling of software and hardware components that are related to packet processing. Moreover, it allows these components to be managed by the applications and reduces the contention between CPUs.

NetSlice is based on spatial instead of temporal partitioning of the software and hardware components that are relevant to packet processing, i.e of the memory, the CPU cores and the NICs. By performing such partitioning, NetSlice minimizes the contention of the shared resources.

The elementary notion of the NetSlice is the execution context, named *NetSlice*. In fact, there is an array of such execution contexts, in order to provide high parallel execution, as well as keeping contention rates low.

In figure 5, an array of NetSlices is depicted. As mentioned above, NetSlice takes advantage of multi-core and multi-queue NICs. In order to support parallel execution at the multiple cores, the multi-queue NICs maintain more than one queue for transmission and reception. The NetSlice execution context contains implicitly partitioned resources and the CPU cores and NICs which are also referred to as explicitly partitioned resources. More specifically, each NetSlice must have at least two CPU cores. The CPU cores of a NetSlice execution context are classified into *k-peer*, and *u-peer*, where the in-kernel and user-mode tasks are executed correspondingly. Only one of these CPU cores, which has been specified to receive the interrupts of the queues of the context, is the k-peer CPU and is assigned to execute the in-kernel network stack. The user-mode task is executed in parallel on the u-peer CPU. The k-peer and the u-peer CPU cores of an execution context are defined as tandem CPU cores, and it is the number of tandem CPU cores that determines the count of the NetSlices.

NetSlice also determines the path that packets have to follow from NICs to applications and vice versa. Packets are slightly processed on the k-peer CPU core, and then they are directed to the user-space application, where they are processed in a pipeline.

NetSlice is based on the conventional socket API. More specifically, it uses the operations write, read and poll for the distinct data flows of the different NetSlices. NetSlice extends the API via the *ioctl* mechanism. One difference of the NetSlice extended API is the batched system calls that it can support. Due to the batching, NetSlice achieves a reduced number of system calls, and thus minimizes the delays of system calls. Batching results also in a reduction of the overheads that per packet processing poses.

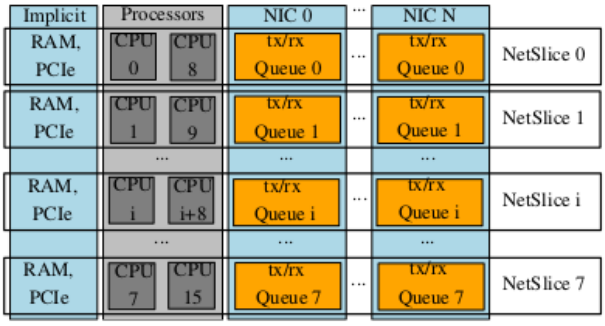

Figure 5: Array of NetSlices [10]

NetSlice does not exploit the advantages of zero-copy techniques, and it does copy the packets from kernel to the user-space. It is estimated that zero-copy technique could indeed further

doi: 10.2313/NET-2014-08-1_06

ameliorate the performance of NetSlice. However, it is not implemented in the framework, since it would restrict portability.

6. PACKET PROCESSING WITH GPUs

6.1 GPU

The architecture of the Graphical Processing Units with the hundreds or thousands of cores, have resulted in their use in other applications too, except their primary use in graphics rendering [8, 4]. Highly parallel applications can take advantage of the ample computation cycles and the ample memory bandwidth of GPUs, higher than that of CPUs, which enables the access to different data sets [8]. Among the applications that can be benefited from the highly parallel computing of GPUs are applications related to packet processing tasks.

6.1.1 GPU Overview

Due to the numerous cores that they possess, modern GPUs have gained a massively parallel processing power [23]. A GPU has a device memory, which is explicitly accessed by the GPU. Implicit access of the host memory can be achieved through DMA or through the Peripheral Component Interconnect Express (PCIe) bus. The PCIe bus is also necessary for the communication between GPU and the corresponding host CPU. GPUs are based on a SIMT (Single Instruction, Multiple Thread) execution model. According to this, many cores have a common program counter and threads are executed in parallel.

6.1.2 GPU Benefits in packet processing

GPUs provide high thread-level parallelism, while a CPU is organized in a way that increases the instruction-level parallelism. As a consequence, a GPU, contrary to a CPU achieves to minimize the memory access latency by executing a considerable number of threads.
The significant memory bandwidth of the GPU, can be used for memory-intensive tasks. This can be very beneficial particularly in cases where the CPU' s memory bandwidth is used for packet I/O. Similarly, the processing power of the GPU can be used for computing-intensive operations which run on software routers and thus overcome the bottleneck posed by the CPU.

6.2 PacketShader

PacketShader [4] constitutes a user-space framework for fast-packet processing that takes advantage of the GPU characteristics. What increases the speed of the PacketShader is, except the GPU computing power, the optimized packet I/O engine which achieves the performance of 10Gbit/s due to pipelining and batching.

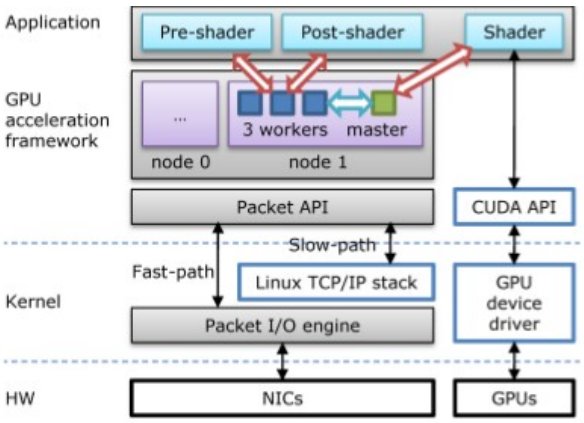

Figure 6: PacketShader Architecture [4]

6.2.1 PacketShader Architecture

The architecture of the PacketShader is depicted in the figure 6. Its more important components are analyzed below:

Packet I/O Engine: PacketShader does not use the per-packet buffer allocation scheme; it develops a new one, namely the huge packet buffer. According to the huge packet buffer scheme, the device drivers allocate one huge buffer for the metadata and another one for the real packet data. This optimized buffer allocation scheme offers two advantages. First, it does not result in CPU overhead as the per-packet buffer allocation does. Second, it reduces per-packet DMA mapping.

GPU Acceleration framework: The nodes in the GPU acceleration framework are NUMA (Non-uniform memory access) nodes and are used to increase parallelization. According to the NUMA model, the time needed to access the memory, is related to the position of the physical memory. The nodes contain CPUs with four cores. Three cores are allocated for worker threads and one for master thread.

Packet processing application: The packet processing application works with batches of packets instead of individual packets and is divided into three tasks, which are performed by the methods Pre-Shader, Post-Shader and Shader.

The Pre-Shader method is executed by a worker thread. It is responsible for packet classification and for forwarding data to the master thread. While waiting for the master thread to finish with the shading, the worker thread is processing the pre-shading for the next batch of packets.

Then the Shader method is performed. The master thread copies the data between the GPU device memory and the host memory, and launches the kernel, i.e the code that is executed by the GPU. The data copying and the execution of GPU kernel run concurrently in order to increase GPU performance. This technique is known as *concurrent copy and execution*. Afterwards, it returns the data to the worker thread where the Post-Shader method takes place. Another technique used in this step, for better performance, is the gather/scatter, which enables

doi: 10.2313/NET-2014-08-1_06

the system to process multiple batches of packets. The master thread gathers data from the worker threads and after the shading processing, scatters them in the corresponding worker threads.

During the post-shading, the packets can get modified, duplicated or discarded and then they are ready for transmission.

6.3 Snap

Snap [23] is another framework that enables fast packet processing due to both the processing power of GPU and the modular and flexible architecture of the Click modular router.

6.3.1 Snap Architecture

Snap exploits the extended parallelism that GPU offers. In the Snap framework, not all elements are offloaded to GPUs. Only elements that result in bottlenecks are implemented in GPUs. The other elements are still supported by the CPU as in the Click architecture. Besides, not all tasks of packet processing are suitable for highly parallel processing and therefore GPU is not appropriate for them.

Snap supports two types of elements. The first is the serial element and refers to Click's conventional elements, meaning those elements that fit in Snap's architecture without having undergone any change. The second type of elements is the GPU-based parallel element that Snap introduces in order to execute them in the GPU. This new type has, except the GPU code, a CPU part, which is responsible for receiving batches of packets and launching the kernel. For the communication of these elements, which process single packets and batches of packets respectively, a Batcher and a Debatcher element are needed. These elements act as adapters. The Batcher gathers individual packets from a serial element and inputs them in form of batches in a parallel element. Its opposite element, the Debatcher, receives batches of packets from a parallel element and transfers the packets separately to a serial element. By attributing the adapter functionality to specific elements, Snap allows the developer to monitor batching and offloading.

Batching is necessary for achieving high performance through GPU processing. Batching in Snap, as well as in PacketShader, differs from batching in Click, in that the latter refers to batches of elements that run on one individual packet, while Snap and PacketShader's batching signifies that a batch of packets is processed by the same element.

In order to be able to support this kind of batching, Snap introduces the functions *bpull()* and *bpush()* that perform the functionality of pull() and push() methods of Click for batches of packets. These modified functions can be performed by GPU-based parallel elements.

Similarly to the huge buffer allocation scheme that PachetShader uses, Snap uses huge buffers too for the batches of packets. In this way, it reduces per-packet DMA mapping.

As has been mentioned, GPU cores communicate only directly with the device memory. Therefore, copying packets between GPU and host memory through the PCIe bus is unavoidable. Snap diminishes the times that such a copying is necessary and consequently the memory overhead by implementing two types of elements, the *HostToDeviceMemcpy* and the *DeviceToHostMemcpy* element and putting them at the edges of a sequence of parallel elements.

Snap manages to overcome the problem of packet reordering which has been observed in parallel packet processing. This happens because it requires that all threads of a parallel element that processes a batch of packets must have finished before the batch of packets can be processed by the next parallel element. For avoiding a potential packet reordering because of the asynchronous GPU scheduling, a GPUCompletionQueue element is implemented and placed between the DeviceToHostMemcpy and the Debatcher.

7. COMPARISON OF THE FRAMEWORKS

In this section, the solutions presented above are compared. More specifically a comparative table showing which of the techniques discussed in the previous sections are implemented by each framework, is presented.

	Netmap	PF_Ring DNA	Intel DPDK	NetSlice	Snap	PacketShader
Memory preallocated, re-used	yes	yes	yes			yes
Parallel direct paths				yes		yes
Memory mapping	yes	yes	yes			yes
Zero-copy	yes	yes	yes			
Batch processing	yes			yes	yes	yes
CPU affinity	yes		yes	yes		yes
Memory affinity			yes	yes		yes
Aggressive prefetching						yes
GPU processing					yes	yes

Figure 7: Techniques per framework

8. CONCLUSION

Commodity network interface cards can achieve the speed of 10 Gbit/s. As a consequence, the need for performing packet processing at a corresponding rate with general-purpose hardware arose. In this paper, a survey of current techniques for high-speed packet processing was conducted. The survey was mostly focused on existing frameworks developed for serving fast packet processing. This survey described and compared several frameworks which provide fast packet processing. These frameworks actually manage to do so, by applying proposed techniques to the network stack and to the way that packet processing is performed. A newly proposed and emerging technique, that some recent frameworks have adopted, involves the incorporation of GPU for processing parts of packet processing.

9. REFERENCES

[1] Bianco, A., Birke, R., Bolognesi, D., Finochietto, J., Galante, G., Mellia, M., Prashant M., and Neri F. 2005.

doi: 10.2313/NET-2014-08-1_06

Click vs. Linux: Two Efficient Open-Source IP Network Stacks for Software Routers. IEEE High Performance Switching and Routing Workshop, Hong Kong, May 2005.

[2] Deri, L. 2004. Improving Passive Packet Capture: Beyond Device Polling. Proceedings of the 4th International System Administration and Network Engineering Conference (SANE), 2004.

[3] Garcıa-Dorado, J., Mata, F., Ramos, J., Santiago del Rio, P., Moreno, V., and Aracil, J. 2013. High-Performance Network Traffic Processing Systems Using Commodity Hardware, in Biersack, E., Data Traffic Monitoring and Analysis, (Berlin: Springer, 2013).

[4] Han, S., Jang, K., Park, K., and Moon, S., Packetshader: a gpu-accelerated software router. ACM SIGCOMM Computer Communication Review 40, 4 (2010), 195–206.

[5] Indiresan, A., Mehra, A., and Shin, K. 1998. Receive Livelock Elimination via Intelligent Interface Backoff. TCL Technical Report (Michigan: University of Michigan, 1998).

[6] Ine, J. 2014. PF_RING User Guide, Version 5.6.2, Jan 2014, ntop.org

[7] Intel. 2014. Intel® Data Plane Development Kit (Intel® DPDK), Programmer's Guide.

[8] Jang, K., Han, S., Han, Se., Moon, S., and Park K. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors. USENIX NSDI, April 2011.

[9] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. 2000. The Click modular router. ACM Trans. Comput.Syst., 18(3):263–297, Aug. 2000.

[10] Marian, T., Lee, K., and Weatherspoon, H. 2012. Netslices: Scalable Multi-Core Packet Processing in User-Space, ANCS'12, October 29–30, 2012, Austin.

[11] Meyer, T., Wohlfahrt, F., Raumer, D., Wolfinger, B., and Carle, G. 2013. Measurement and Simulation of High-Performance Packet Processing in Software Routers. 7th GI/ITG-Workshop MMBnet, Hamburg, September 2013.

[12] Mogul, J., and Ramakrishnan , K. 1997. Eliminating receive livelock in an interrupt-driven kernel. ACM Transactions on Computer Systems (TOCS) 15, 3 (1997), 217–252.

[13] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. 1999. The Click modular router. Operating Systems Review 34(5):217–231, December 1999

[14] NTOP. http://www.ntop.org/products/pf_ring/dna

[15] Rizzo, L. 2012. Revisiting Network I/O APIs: The Netmap Framework. Communications of the ACM, vol. 55 no. 3, March 2012.

[16] Rizzo, L. 2012. Netmap: a novel framework for fast packet I/O. Proceedings of the 2012 USENIX conference on Annual Technical Conference, USENIX ATC'12, Berkeley, CA, 2012.

[17] Rizzo, L., Carbone, M., and Catalli, G. 2012. Transparent acceleration of software packet forwarding using netmap. INFOCOM'12, Orlando, FL, March 2012, http://info.iet.unipi.it/∼luigi/netmap/.

[18] Rizzo, L., Deri, L., and Cardigliano, A. 2012. 10 Gbit/s Line Rate PacketProcessing Using Commodity Hardware: Survey and new Proposals. Online: http://luca.ntop.org/10g.pdf (2012).

[19] Salim, J., Olsson, R., and Kuznetsov, A. 2001. Beyond Softnet. Proceedings of the 5th Annual Linux Showcase & Conference - USENIX Association, (Oakland,CA, 2001).

[20] Salim, J. 2005. When NAPI Comes to Town. Linux 2005 Conference.

[21] Schuchard, M., Vasserman, E., Mohaisen A., Kune, D., Hopper, N., and Kim, Y. 2011. Losing Control of the Internet: Using the Data Plane to Attack the Control Plane, ISOC Network & Distributed System Security Symposium (NDSS 2011).

[22] Song, J., and Alves-Foss, F. 2012. Performance Review of Zero Copy Techniques. International Journal of Computer Science and Security (IJCSS), Volume (6) : Issue (4) : 2012.

[23] Sun, W., and Ricci, R. 2013. Fast and Flexible: Parallel Packet Processing with GPUs and Click. Proceedings of the 9th ACM/IEEE symposium on Architectures for networking and communications systems, 2013.