Commodity Packet Capture Engines: tutorial, cookbook and applicability

Victor Moreno, Javier Ramos, José Luis García-Dorado, Pedro M. Santiago del Río, Francisco J. Gomez-Arribas, Javier Aracil

Abstract

Users' demands have dramatically increased due to widespread availability of broadband access and new Internet avenues for accessing, sharing and working with information. In response, operators have upgraded their infrastructures to survive in a market as mature as the current Internet. This has meant that most network processing tasks (e.g., routing, anomaly detection, monitoring) must deal with challenging rates, challenges traditionally accomplished by specialized hardware —e.g., FPGA. However, such approaches lack either flexibility or extensibility —or both. As an alternative, the research community has proposed the utilization of commodity hardware providing flexible and extensible cost-aware solutions, thus entailing lower operational and capital expenditure investments. In this scenario, we explain how the arrival of commodity packet engines has revolutionized the development of traffic processing tasks. Thanks to the optimization of both NIC drivers and standard network stacks and by exploiting concepts such as parallelism and memory affinity, impressive packet capture rates can be achieved in hardware valued at a few thousand dollars. This tutorial explains the foundation of this new paradigm, i.e., the knowledge required to capture packets at multi-Gb/s rates on commodity hardware. Furthermore, we thoroughly explain and empirically compare current proposals, and importantly explain how apply such proposals with a number of code examples. Finally, we review successful use cases of applications developed over these novel engines.

Keywords: Commodity hardware; packet capture engine; high-performance networking; network traffic monitoring.

*Authors are with High Performance Computing and Networking research group, Universidad Autónoma de Madrid, Spain. Email: {victor.moreno, javier.ramos, jl.garcia, ivan.gonzalez, francisco.gomez, javier.aracil}@uam.es

1 Introduction

Users' demands and the capacity of both backbone and access links have played a daily game of cat and mouse almost since the Internet was born. On the one hand, the widespread availability of the Internet is a fact. Moreover, users tend to use the Net more intensively as new applications gain significant popularity in a matter of weeks [1]. In line with such an increased demand, users' quality of service expectations have also strengthened turning the Internet into a truly competitive and mature market.

On the other hand, the operators' answer has been higher investments in terms of both capital and operational expenditures (CAPEX and OPEX). That is, backbone links witness continuous upgrades, and probes have been deployed across operators' infrastructures to allow them to perform measurement activities and careful monitoring, which helps satisfy quality demands from users but also entails costly management.

Needless to say, it is not only operators who face the challenge of handling and monitoring high-speed networks, but also other entities such as banking institutions, content providers, and network application developers [2]. In these cases, the task may not only be to deliver bytes but also to dig into applications' behavior, achieve the best performance, or inspect what traffic looks like. As an example, in a bank network where security is a key element, network managers must collect and study huge sets of data often with aggregated rates of several Gb/s to identify anomalous behavior and patterns. Indeed, things may be even worse as malicious traffic often shows bursty profiles [3].

In short, the increment of the user's demands and subsequent link capacities have forced the different players in the Internet arena to deal with multi-Gb/s rates. To put this into perspective, we note that, for instance, packet-traffic monitoring at rates ranging from 100 Mb/s to 1 Gb/s was considered very challenging only a few years ago [4–6], whereas contemporary commercial routers typically feature 10 Gb/s interfaces, reaching aggregated rates as high as 100 Tb/s [7].

As a consequence, network operators have entrusted specialized hardware (HW) devices such as FPGA-based solutions [8, 9], TCAMs [10, 11], or high-end commercial solutions with their network tasks [12]. These solutions respond to high performance needs for a very specific task, e.g., lossless packet handling in a multi-Gb/s link while routing or classifying traffic [13].

However, the initial investment is high and such specialization, as with any custom-made development, lacks both extensibility and flexibility, which in turn also have an indirect cost impact. As an example, in the case of large-scale networks featuring numerous Points of Presence (PoP), extensibility and ease of update are key. Equivalently, in the case of a smaller network, it is desirable to have hardware flexible enough to carry out different tasks as specifications change. Additionally, in any scenario, network managers must prevent their infrastructure from being locked into a particular vendor. As a workaround to these limitations, some initiatives have provided extra functionalities in network elements through supported Application Programming Interfaces (APIs) that allow the extension of the software part of their products —e.g., OpenFlow [14].

It has been only recently that the research community has proposed, as a real alternative, the use of software-based solutions running on top of commodity general-purpose hardware to carry out high-performance network tasks [15, 16]. The key point is that this provides flexibility and extensibility at a low cost. Additionally, leveraging commodity hardware to develop network services and applications brings other advantages. All the components a commodity system is based on are well known and popular hardware. This makes these systems both more robust, due to extensive validation, easy to replace, and cheaper, as the development cost per unit is lower thanks to the economies of scale of large-volume manufacturers.

Unfortunately, the use of commodity hardware in high-speed network tasks is not trivial due to limitations on both hardware capacities and standard operating systems performance. Essentially, commodity hardware is limited in terms of memory and internal bus throughputs, while operating systems provide a general network stack that prioritizes protocol and hardware compatibility.

To give some figures about the size of the challenge, in a fully-saturated 10 Gb/s link the time gap between consecutive packets assuming an Ethernet link and minimum IP packet size is lower than 68 ns, while an operating system may need more than half a microsecond to move each packet from kernel to application layer [17]. This calls for a careful tuning of both hardware and the operating system stack to improve the figures. Nonetheless, even using the best-tuned hardware and operating system combination, there will be packet losses if the application layer is not aware of the lower levels' implementation.

The development of high-performance network services and applications over commodity hardware typically follows a four-layer model. The first layer comprises the Network Interface Card (NIC), that is, the hardware aimed at capturing the incoming packets. There are several major NIC vendors but it is Intel, and especially its 10 GbE model with chipset 82599 controller, that has received most attention from the community. This chipset provides performance at competitive prices, and more importantly, it offers novel characteristics that turn out to be fundamental in order to achieve multi-Gb/s rates at the application layer. The next layer includes the driver. There are of two kinds: standard or vanilla drivers, i.e., as provided by Intel; or user customized ones. The third layer moves packets from the kernel level to the application layer. This includes the standard way operating systems work, i.e., by means of a socket and network stack. In addition to this, there are different libraries that help application developers to interact with traffic by means of a framework. Among all these libraries, PCAP is considered the *de facto* standard [18]. The combination of a driver and a framework is known as a packet capture engine and the literature gives several examples of high-quality engines using commodity hardware [16]. These engines typically feature a new driver and performance-aware frameworks. Finally, we have the application layer, which encompasses any service or functionality built on top of a capture engine. As previously mentioned, examples can be found nowadays in software routers, firewalls, traffic classification, anomaly and intrusion detection systems, and many other monitoring applications [19].

Importantly, while the optimization of specific applications at the highest layer has already receive significant attention. We note that there is a lack of a rigorous analysis and comparison compendium that spans the three lower layers. For example, although the authors in [20] focus on how to extract flow records in high-speed networks once packets are captured, they emphasize the relevance of fully understanding the packet capture process to create and operate high-performance reliable network applications. This tutorial aims at filling this gap.

Specifically, we present, in a tutorial-like fashion, all the knowledge required to build high-performance network services and applications over novel capture engines running on commodity hardware. Furthermore, we will show shortcuts to speed-up the development of such services and applications, by explaining the hardware and software keys to implement a custom packet capture engine, by detailing and illustrating with command prompt instructions how the capture engines proposed in the literature work, as well as by carrying out a performance comparison that allows users to select the capture engine best fitting their needs. In addition to this, we present a state-of-the-art overview of current services and applications already benefiting from this new network paradigm.

The rest of this tutorial is organized as follows. The next section explores the characteristics of the hardware referred to as commodity hardware as well as how current operating systems' network stacks work. Then Section 3 gives a strong background on the limitations of commodity hardware, necessary to understand the keys to overcome them. This knowledge would suffice for

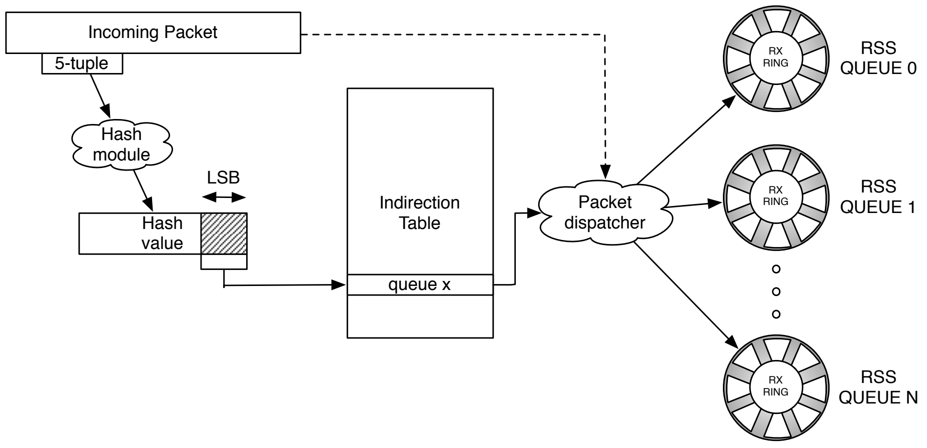


Figure 1: RSS architecture

users to develop their own high-performance network drivers starting from vanilla ones. However, there is the option of using one of the high performance packet capture engines proposed in the literature. This is discussed in Section 4, thus enabling practitioners not interested in low-level details but in developing applications on commodity hardware, to skip much of the effort to master low-level interactions. Section 5 is devoted to evaluate the performance of capture engines. First, we explain how to evaluate capture engines and then we present a fair comparison between the engines in the literature in terms of packet losses and computational resource consumption to allow potential users to choose the most suitable engine for their ultimate aims. After all the theoretical knowledge has been introduced, Section 6 gives a cookbook on how to know the system's architecture, load a driver (customized or not), modify the driver's characteristics, optimize performance, essentially a practical tutorial on how to get started with packet capture engines. After that, in Section 7, we survey applications that have leveraged packet capture engines successfully since this novel paradigm emerged. This may awaken new ideas but the reader can also view this as the current state-of-the-art bounds to beat. Finally Section 8 provides a list of lessons learned and some conclusions.

We believe that all this background, know-how, comparisons and practical lessons are what practitioners and researchers need to develop high-performance network services and applications on commodity hardware. In addition, advanced readers may find enriching unknown details, benefit from the comparison of the different capture engines currently in the literature, which itself is of great interest, and also find the applications other researchers are developing.

2 Commodity Hardware

The computational power required to cope with network data is always increasing. Traditionally, when the requirements were tight an eye was turned to the use of ASIC designs, reprogrammable FPGAs or network processors. Those solution offer great computational power at the expense of high levels of specialization. Consequently, they only address the performance half of the problem but they fail at solving the other half, which is the inexorably need to perform more diverse, sophisticated and flexible forms of analysis.

Commodity hardware systems are computers that combine hardware with a common instruction set and architecture (memory, I/O and expansion capabilities) and open-source software. Such computers contain industry-standard PCI slots that allow expansion and mechanical compatibility to provide a wide range of configurations at minimal cost. Such characteristics position commodity hardware as a strong option in terms of economies of scale with reduced manufacturing costs per unit. Moreover, the widespread use of commodity NICs and multi-core CPUs enables computers to capture and process network traffic at wire-speed reducing packet losses in 10 GbE networks [21].

Lately the number of CPU cores per processor has been increasing and nowadays quad-core processors are likely to be found in home computers and even eight-core processors in commodity servers. Along with this step-up, modern NICs have significantly evolved both in terms of hardware design and capture paradigms. An example of this evolution is the technology developed by Intel [22] and Microsoft [23] known as Receive Side Scaling (RSS). The main role of RSS is to distribute network traffic load among the different cores of a multi-core system and optimize cache utilization. Such a distribution overcomes the processing bottleneck produced by single-core approaches. RSS traffic distribution among different receive queues is achieved by using an indirection table and a hash value calculated over a configurable set of fields of received packets. Each receive queue must be bound to a different core in order to balance the load efficiently across the system resources.

As can be observed in Fig. 1, once the hash value has been calculated, its Least Significant Bits (LSB) are used as index for the indirection table. Based on the values contained in the indirection table the received data can be assigned to a specific processing core. The default hash function used by RSS is a Toeplitz hash. This hash uses the IPv4/IPv6 source and destination addresses and protocol field; TCP/UDP source and destination ports; and, optionally, IPv6 extension headers. The resulting hash assigns traffic to queues maintaining unidirectional flow-level coherency, that is, packets containing the same 5-tuple will be delivered to the same processing core. Hash function can be changed to distribute the traffic in different ways. For instance an approach for maintaining bidirectional flow-level (session-level) coherency is presented in [24].

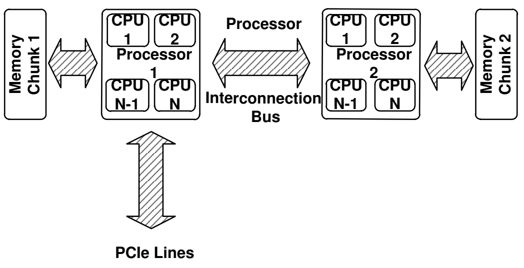
2.1 NUMA architectures

Besides new hardware improvements, the interaction between the software and the hardware is an aspect of paramount importance in commodity hardware systems. For instance, Non-Uniform Memory Access (NUMA) has become the *de facto* standard for multiprocessor architectures and has been exploited for high-speed traffic capture and processing. Briefly, NUMA architecture divides all available system memory into chunks and assigns each chunk to a different Symmetric MultiProcessor (SMP). The combination of a processor and a memory chunk is known as a NUMA node. Some topology examples of NUMA architectures are shown in Fig. 2.

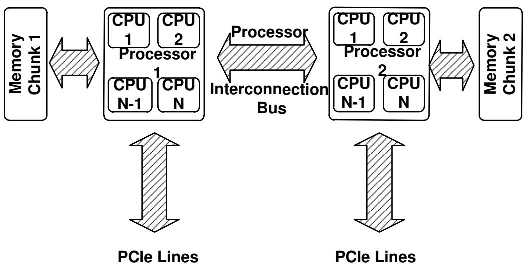
In NUMA architectures each processor may access its own chunk of memory in parallel, boosting system performance and reducing the CPU data starvation problem. Notwithstanding that NUMA architecture increases performance in terms of both memory accesses and cache misses [25], a careful process placement must be performed in order to avoid accessing memory located on another NUMA node.

To get the most out of NUMA architectures the distribution of NUMA nodes must be known in advance since it may vary depending on the hardware platform. To obtain the NUMA node distance matrix the `numactl`[1] command can be used. This matrix describes the distance from

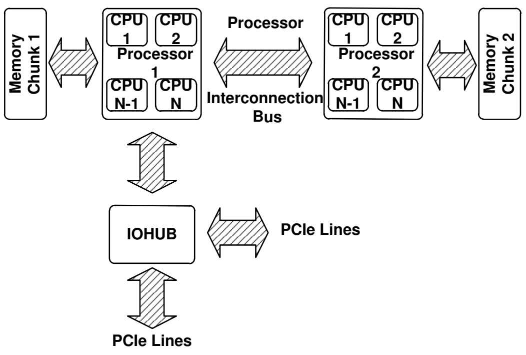
[1] `linux.die.net/man/8/numactl`



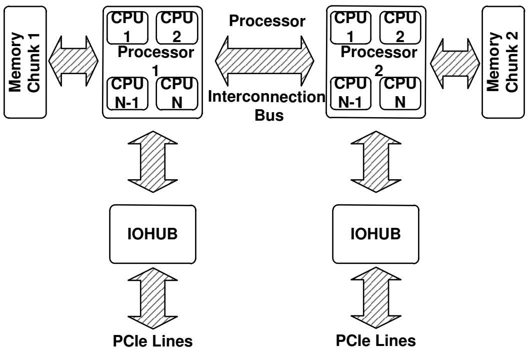
((a)) PCIe lines connected to one processor



((b)) PCIe lines connected to two processors



((c)) PCIe lines connected to one IOHUB



((d)) PCIe lines connected to two IOHUB

Figure 2: NUMA architectures

each NUMA node memory chunk to the others. As expected, the lower the distance the lower the access latency to other NUMA nodes.

Another aspect of paramount importance is the location and interconnection of devices. Typically in commodity hardware network capture systems, NICs make use of PCI-Express (PCIe) buses to connect to the processors. Connection may vary depending on the motherboard used in the commodity hardware capture system. Fig. 2 shows the most common interconnection patterns on current motherboards. In more detail, Fig. 2(a) shows an asymmetric architecture with all PCIe lines directly connected to a processor in contrast to Fig. 2(b) showing a symmetric scheme with PCIe lines distributed between two processors.

Figs. 2(c) and 2(d) show two topologies in which IO-hubs are used. The main difference between them is the existence of PCIe lines connected to one or more IO-hubs. Such IO-hubs interconnect PCIe buses as well as USB, standard PCI buses and other devices. Due to this architecture the bus between the IO-hub and the processor is shared with the subsequent performance problems. All these architectural issues must be considered when building a capture system. For instance, if a NIC is attached to a PCIe slot assigned to a NUMA node, all capturing threads should be executed on the corresponding cores of the NUMA node. If this is not done, data transmission between processors using the Processor Interconnection Bus may occur, degrading system performance.

For all the above reasons, modern commodity systems are highly attractive to accomplish the demanding task of network traffic monitoring at high speeds as their performance is on par with today's specialized hardware, such as network processors [26–28], FPGAs [29], Endace DAG cards [30] or commercial solutions provided by router vendors [31] while keeping down the

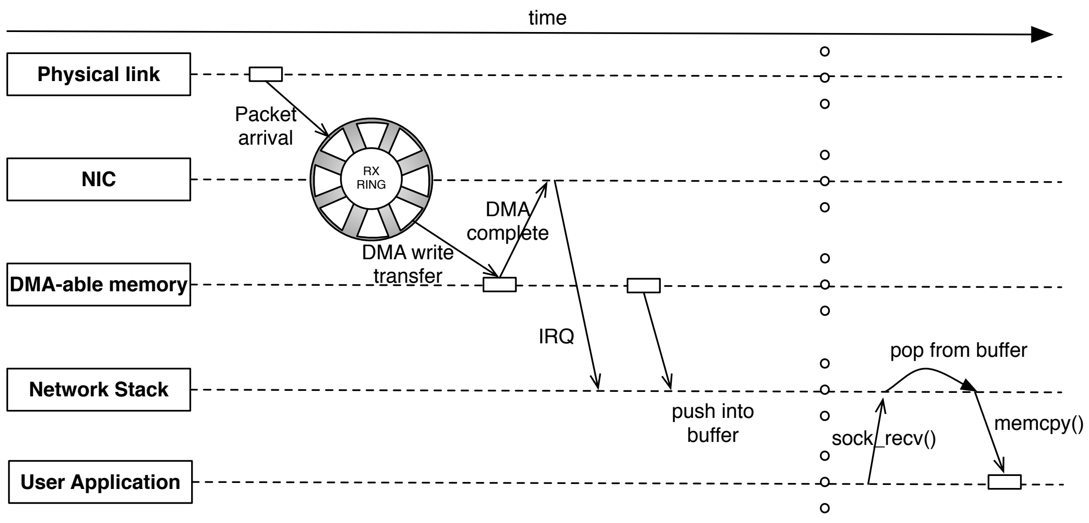


Figure 3: Linux Network Stack RX scheme in kernels previous to 2.6

expense.

2.2 Current and past operating system network stacks

While network hardware has been rapidly evolving focusing on high-speed packet capture, software has not followed the same trend. In the case of software, modern operating systems are nowadays designed to provide compatibility rather than performance. Such operating systems present a general-purpose network stack that provides a simple socket user-level interface for data exchange and handles different hardware and network protocols. Nevertheless, such an interface is not optimal in terms of high-speed traffic capture.

Particularly, Linux kernels prior to 2.6 presented an interrupt-driven approach in the network stack. Focusing on behavior: whenever a packet arrives at the corresponding NIC, a descriptor in a NIC's receive (RX) queue is allocated and assigned to that packet. These queues are also known as rings due to their circular topology. Each packet descriptor contains a pointer to the memory region needed to perform the incoming packet transfer via Direct Memory Access (DMA). In the case of packet transmission, DMA transfers are performed in the opposite direction and an interrupt line is raised upon completion to allow the transmission of new packets. This mechanism is common to all the different packet I/O commodity hardware solutions.

Fig. 3 shows the traditional Linux network stack behavior. When an incoming packet DMA transfer from the NIC to the host's memory (DMA-able memory region) is finished, an interrupt is signaled. Then, the software interrupt routine copies the packet's data into a local kernel `sk_buff` structure. This structure is typically called a kernel packet buffer. Once the packet has been copied, the packet descriptor is released so the NIC can reuse it to receive new packets. The `sk_buff` structure with the received packet data traverses the system's network stack until delivered to a user application. Following this I/O scheme, an interrupt must be raised each time a packet is received or transferred. This mechanism overcrowds the host system when the network load is high [32].

To avoid such behavior, current network drivers implement NAPI (New API) [33] to increase

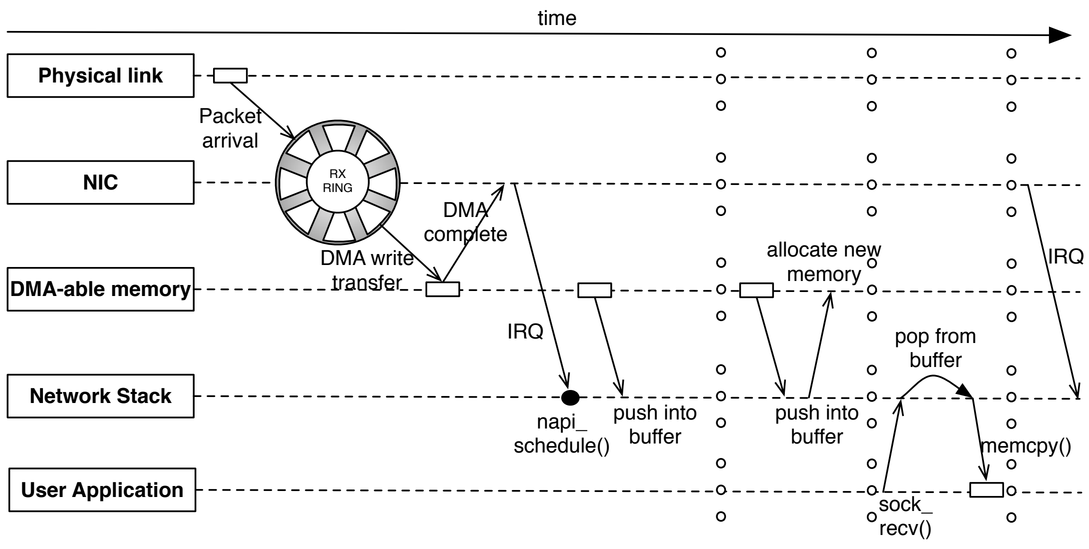


Figure 4: Linux NAPI RX scheme

performance. NAPI was included in Linux kernel 2.6 to boost packet processing in high-speed scenarios. To speed up the packet capture, NAPI is based on two ideas:

(i) **Interrupt mitigation**: To reduce the overload produced by high-interrupt rate when high-speed traffic is present, the NAPI-aware driver interrupt routine is launched when RX/TX interrupts arrive. Unlike the traditional approach, the interrupt routine schedules the execution of a `poll()` function and disables interrupts instead of copying and queuing the packet. The `poll()` function checks if new packets are received, and copies and enqueues them into the network stack when available in an interrupt-less way. The function reschedules itself to be executed in the near future (without waiting for interruptions). If no packets are available in this time period, packet interrupts are activated again. Note that polling mode demands more CPU time than interrupt-driven mode when network load is low but it becomes worthwhile as speed grows. Depending on the network load, NAPI compliant drivers adapt themselves to increase the performance as shown in Fig. 4.

(ii) **Packet throttling**: Traditionally, when high-speed traffic surpassed system capacity, packets were dropped at kernel-level rendering the previous communication and copying between drivers and kernel useless. NAPI compliant drivers drop traffic at network adapter level using flow control mechanisms thus preventing unnecessary work.

In what follows, the GNU Linux operating system and NAPI mechanism will be used to illustrate performance problems and limitations, as well as to explain proposed solutions. This choice has been made as Linux is a widely used open-source operating system that allows full code modification for instrumentation and performance analysis purposes. Although the vast majority of the proposals in the literature have been developed for different flavors of the GNU Linux distribution, some of them are also available for other operating systems such as FreeBSD [34]. We assume users to have a working knowledge of Linux-based operating systems otherwise the reader is referred to [35,36] to obtain the background required to make the most of this tutorial.

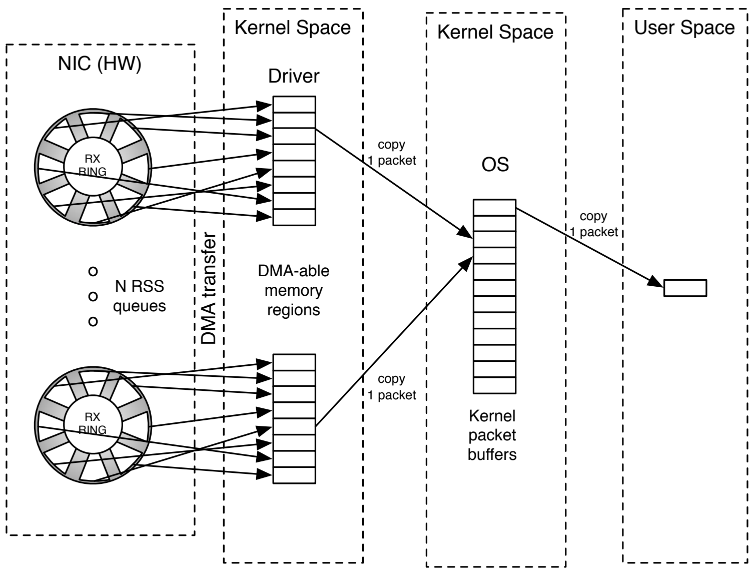


Figure 5: Legacy Linux Network Stack (serialized paths)

3 Packet capturing

3.1 Limitations: wasting the potential performance

Although the way operating systems' network stacks work has evolved, their robustness and flexibility remain a burden in terms of packet processing rates. The NAPI technique by itself is not enough to overcome the challenging task of very high-speed traffic capturing since other inherent architectural problems degrade performance. After extensive code analysis and performance tests, several problems have been identified [21, 34, 37, 38]:

(i) **Per-packet allocation and deallocation of resources**: Whenever a new packet arrives at a NIC, an `sk_buff` data structure is allocated by the kernel to store the packet's information. Once the packet has been delivered to user-level, its descriptor is released. This resource allocation and deallocation process generates a significant overhead, especially when receiving at high packet rates —as high as 14.88 million packets per second (Mp/s) in 10 GbE. Additionally, this `sk_buff` data structure is large because it may comprise information from many protocols on multiple layers, but most of this information is not necessary for numerous networking tasks. Modern drivers tend to group this structure allocation requests as a workaround in order to reduce the impact on performance of this process. As shown in [37], `sk_buff` conversion and allocation consume nearly 1200 CPU cycles per packet, while buffer release needs 1100 cycles. Indeed, `sk_buff`-related operations consume 63% of the CPU usage in the reception process of a single 64-byte sized packet [21].

(ii) **Serialized access to traffic**: Modern NICs include multiple hardware (HW) RSS queues which are used to distribute the incoming traffic using a hardware-based hash function applied to the packet's 5-tuple (Section 2). By exploiting this feature the capture process can be parallelized, since different NAPI threads could be bound to different CPU cores so each thread gathers the packets from a specific RSS queue. However, once packets are fetched, the GNU Linux network stack merges all packets at a single point on network and transport layers for analysis. Fig. 5 shows the architecture of the standard GNU Linux network stack. As a result, two problems arise from the use of this philosophy: first, all traffic is merged in a single point, which creates a processing bottleneck thus limiting overall throughput; second, user processes are capable of receiving the traffic from a certain RSS queue. Consequently, we cannot make the most of parallel capabilities of modern NICs delivered to a specific queue associated with a socket descriptor. This serialization process degrades the system's performance regardless of any optimizations a particular network driver might implement. Furthermore, merging traffic from different receive queues may entail packet disordering [39] and affect upper-layer packet processing policies.

(iii) **Multiple data copies from driver to user-level**: Packets are transferred to system memory through a DMA transaction. Until those packets are received from a user-level application they are copied several times, at least twice: from the DMA-able memory region in the driver to a packet buffer `sk_buff` structure at kernel-level, and from the kernel packet buffer to the user level. Each additional copy will obviously damage overall performance: a single data copy consumes between 500 and 2000 cycles depending on the packet length [37]. Another important idea related to data copying is the fact that copying data packet-by-packet is not efficient, and deteriorates when packets are small. This is caused by the constant overhead inserted in each copy operation, which favors large data copies.

  Modern drivers reduce in one the amount of data copies required by re-using the same memory area containing the packet along the multiple layers traversed. However, this policy has collateral effects: buffers can not be released until all the upper layers are finished with them, and thus the driver must allocate new buffers or wait until some become available, which may turn into performance losses. Note that, by applying this policy, modern drivers do not need copying the data from the DMA buffer to kernel memory, but one more copy is still needed when transferring the packet's data (o a subset of its original data) to user-space applications.

(iv) **Kernel-to-userspace context switching**: Every time a user-level network application is to receive one packet, a system call must be performed. Each of these system calls will entail a user-level to kernel-level context switch and vice versa, with the consequent CPU time consumption. Such system calls and context switches may consume up to 1000 CPU cycles per packet [37].

(v) **No exploitation of memory locality**: The first access to a recently written DMA-able memory region entails cache misses, as DMA transactions invalidate cache lines. Such cache misses represent 13.8% out of the total CPU cycles consumed in the reception of a single 64 byte packet [21]. Additionally, in a NUMA-based system the latency of memory access depends on the memory node accessed. Thus, inefficient memory placement may entail performance degradation due to greater memory access latencies each time a cache miss is triggered.

3.2 How to overcome limitations

In the previous sections, we have shown that modern NICs are a great alternative to specialized hardware for network traffic processing tasks at high speed. However, both the networking stack of current operating systems and applications at user-level do not properly exploit their new

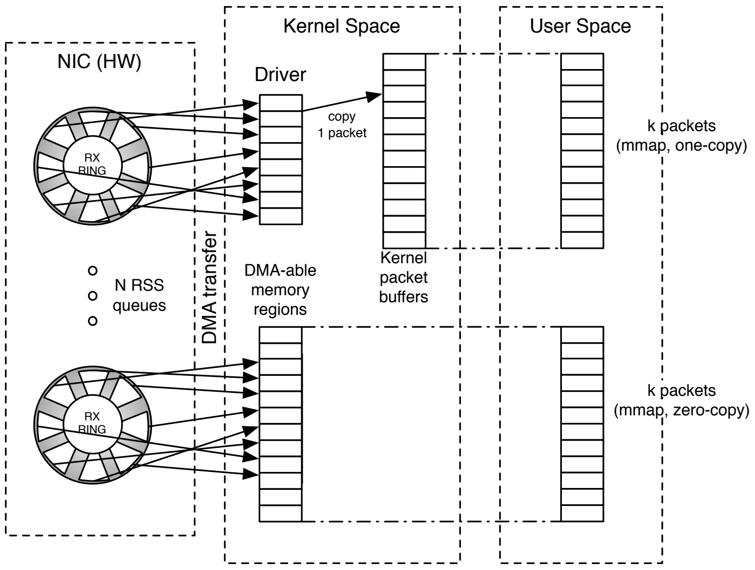


Figure 6: Optimized Linux Network Stack (independant parallel paths)

features. Here we present several proposed techniques to overcome limitations described above in default operating system network stacks.

Such techniques may be applied either at driver level, kernel level or between kernel level and user level, and are specifically applied to the data they exchange, as will be explained below.

(i) **Pre-allocation and re-use of memory resources**: This technique consists in allocating all memory resources required to store incoming packets, i.e., data and metadata (packet descriptors), before starting the packet reception process. Specifically, N rings of descriptors (one for each HW RSS queue on each device) are allocated when the network driver is loaded. Note that this means that the driver loading process will take some extra time, but once the reception process begins the per-packet allocation overhead is suppressed. Likewise, once each packet has been processed and transferred to userspace, its corresponding data structure will not be released, but it will be marked as available so it can be re-used to store a new incoming packet. This policy eradicates the bottleneck produced by per-packet allocation/deallocation. Additionally, the `sk_buff` data structures in use may be simplified to reduce memory requirements. These techniques must be applied at driver level.

(ii) **Exploiting queue parallelism**: This technique pretends to solve serialization in the access to traffic, by creating direct parallel paths between the RSS queues and the network applications as shown in Fig. 6. In order to achieve the best performance, specific and independent cores must be assigned for taking packets from each RSS queue and forwarding them to the user level. This technique supports the creation of new new parallel paths as the number of cores and RSS queues grow, which is an advantage in terms of scalability. In order to obtain such parallel direct paths, we have to modify the data exchange mechanism between kernel and user levels.

On the downside, the use of this technique has two main limitations: First, each parallel path will make use of a CPU core, which reduces the number of cores available for different tasks. Second, RSS distributes traffic to each receive queue by means of a hash function. If our process does not analyze packet interaction, we can maximize parallelism by creating and linking one or more instances of this process to each capture core. However, if our networking tasks require the analysis of related packets, flows or sessions, it will need to fetch packets from different queues. For example, a Voice over IP (VoIP) monitoring system, assuming that such a system is based on the SIP protocol, needs to monitor not only the signaling traffic (i.e., SIP packets) but also calls themselves —typically, RTP traffic. Obviously, SIP and RTP flows may not share either level 3 or 4 header fields that the hash function uses to distribute packets to each queue, hence they might be assigned to different queues and cores. The approach to circumvent this latter problem is that the capture system performs itself some aggregation tasks. The idea is that before packets are forwarded to userspace (for example to a socket queue), a block of the capture system aggregates the traffic according to a given metric. However, this is of course at the expense of performance.

(iii) **Memory mapping**: This feature supported by Linux's memory management model allows user-level applications to map kernel memory regions. Thus, applications are capable of directly reading and writing those memory areas without intermediate copies. This technique can be used to map from user-space those memory areas containing the data from the incoming packets and thus saving one kernel-to-user copy operation. Note that, if the memory areas mapped at user level are not those DMA-able regions where the NIC copies the packet data into, a copy is sill required so will refer to this configuration as *one-copy*. This approach is implemented on current GNU Linux as a standard raw socket when opened with the `RX_RING/TX_RING` socket options. Conversely, if the memory areas mapped by user applications are the DMA-able regions, no data copies are needed to access the packets' data, and thus the term *zero-copy* is used. As an inconvenient, a *zero-copy* driver can not re-use the DMA-able buffers until user applications are done with them. Additionally, exposing NIC rings and register memory areas to user-level access may entail risks for the systems stability [34], which must be properly handled. However, this is considered a minor issue as the APIs provided typically protect the critical regions from incorrect access. In fact, graphic cards make use of memory mapping techniques without major concerns.

Fig. 6 illustrates two different approaches in which memory mapping techniques is used to achieve packet reception with *one-copy* or *zero-copy*. Applying these methods requires either driver-level or kernel-level modifications as well as in the data exchange mechanism between kernel and user levels. The use of memory mapping techniques to share data between kernel and user spaces allows reducing the amount of context switches in the packet capture process and thus improve overall performance. That is the case of modern versions of the libpcap library.

(iv) **Batch processing**: This technique is based on processing several packets at the same time, in order to reduce the overhead of per-packet operations. Packets are grouped into a buffer and copied to the target memory region in groups called batches. This technique reduces the number of system calls made by network applications, with their related context switches. This minimizes the overhead of processing and copying packets individually. In the NAPI architecture, there are two points where batches can be intuitively used. First, if packets are fetched via polling requests, more than one packet can be processed per poll request. Alternatively, if the packet fetcher works on an interrupt-driven basis, an intermediate buffer can be used to collect traffic until upper layers ask for it. However, the use of batching techniques may entail issues such as an increase in latency and jitter, and timestamp inaccuracy on received packets because packets have to wait until a batch is full or a timer expires [40]. In order to implement batch processing, we must modify the data exchange between kernel and user levels.

(v) **Byte-stream oriented**: One step further than packet capture lies packet storage in non-volatile devices. In order to accomplish such task, a packet-by-packet policy issues many write operations and may not perform optimally. To mitigate this effect, some packet capture engines offer access to a byte-stream for user-level applications so they can work in terms of big blocks of bytes.

(vi) **Affinity issues**: In NUMA architectures, in order to increase performance and exploit memory locality, processes must allocate their memory in such a way that it is assigned to the processor (or NUMA node) in which it is being executed. This is known as memory affinity, but CPU and interrupt affinities must also be considered by software designers. CPU affinity allows control of the processors and cores where a given process (process affinity) or thread (thread affinity) is to be executed. Process affinity may be performed using Linux `taskset`[2] utility, and the thread affinity can be managed by means of the `pthread_setaffinity_np`[3] function inside the POSIX pthread library. On the other hand, software and hardware interrupts can also be bound for handling by specific cores or processors using a similar approach known as interrupt affinity. This can be done by writing a binary mask to the file `/proc/irq/IRQ#/smp_affinity`. The importance of setting capture threads and interrupts to the same core lies in the exploitation of cached data and load distribution. Whenever a thread accesses the incoming packets, finding them in the local cache will be more likely if they have been received by an interrupt handler assigned to the same core. Another affinity issue that must be taken into account is to map the capture threads to the NUMA node attached to the PCIe slot the NIC has been plugged into. This PCI affinity allows maximum throughput to be obtained in DMA transfer operations. To accomplish this, the system information provided by the `sysctl` interface (shown in Section 2) may be useful.

(vii) **Prefetching**: Additionally, in order to eliminate inherent cache misses, the driver may prefetch the next packet (both packet data and packet descriptor) while the current packet is being processed. The idea behind prefetching is to load the memory locations that will be potentially used in the near future in the processor cache in order to access them faster when required. Some drivers, such as Intel's `ixgbe`, apply several prefetching strategies to improve performance. Thus, any capture engine making use of such a vanilla driver, will see its performance benefit from the use of prefetching. Further studies such as [21, 41] have shown that more aggressive prefetching and caching strategies may boost network throughput performance.

(viii) **Capture and process isolation**: although we restrict this tutorial to the packet capture process, it is important to remark that any network processing task carried out on top of any of the explained packet capture engines will likely require additional per-packet computation, e.g., packet filtering, protocol classification, flow record extraction, ... Importantly, depending on the packet capture engine in use, this computation may have to be added into the packet capture process and thus increment per-packet processing latency and potentially damage capture performance. However, if the packet capture and processing processes are isolated one from the other and properly pipelined, this performance loss effect can be mitigated.

[2]`linux.die.net/man/1/taskset`
[3]`linux.die.net/man/3/pthread_setaffinity_np`

Table 1: Comparison of the various proposals (D=Driver, K=Kernel, K-U=Kernel-User interaction)

Characteristics/ Techniques	**PF_RING DNA**	**PacketShader**	**netmap**	**PFQ**	**Intel DPDK**	**HPCAP**
Memory pre-allocation and re-use	✓	✓	✓	✓	✓	✓
Parallel direct paths	✓	✓	✓	✓	✓	✓
Memory mapping	✓	✓	✓	✓	✓	✓
Zero-copy	✓	×	✓	×	✓	×
One-copy	×	✓	×	✓	×	✓
Batch processing	×	✓	✓	✓	✓	×
Byte-stream processing	×	×	×	×	×	✓
Capture & process isolation	×	×	×	✓	×	✓
CPU and interrupt affinity	✓	✓	✓	✓	✓	✓
Memory affinity	✓	✓	×	✓	✓	✓
Aggressive prefetching	×	✓	×	×	✓	✓
Multiple listeners	×	×	×	✓	✓	✓
Accurate timestamping	×	×	×	×	×	✓
Level modifications	D,K, K-U	D, K-U	D,K, K-U	D (minimal), K,K-U	D, K-U	D, K-U
API	libpcap	custom	standard libc	socket-like/C, C++, Haskell, pcap	custom	libpcap-like
Supported 10Gb NICs	Intel	Intel	Intel, Mellanox	Any	Intel, Emulex Cisco, Mellanox	Intel
Supported 1Gb NICs	Intel	Intel	Intel, Realtek, nVidia	Any	Intel	×

4 Capture Engine Implementations

In what follows, we present six proposed capture engines, namely: PF_RING DNA [17, 42, 43], PacketShader [21], netmap [34, 44, 45], PFQ [46, 47], Intel DPDK [48] and HPCAP [49, 50], all of which have achieved significant performance levels. For each engine, we describe the system architecture (noting differences from the other proposals), which of the optimization techniques mentioned above have been applied, the API provided for client applications to develop network applications, and what additional functionality it may offer, while the following section will evaluate their performance. Table 1 shows a summarized qualitative comparison between the proposals under study. We have not included some capture engines, previously proposed in the literature, because they are obsolete or unable to be installed in current kernel versions (Routebricks [51], UIO-IXGBE [52]) or where a newer version of these proposals has been released (PF_RING TNAPI [53]).

Specifically, Table 1 illustrates which of the packet capture engines under study implement which of the diverse techniques explained in Subsection 3.2. Two features appearing in Table 1 are not purely performance-related and for this reason they were not mentioned in Subsection 3.2: accurate timestamping and multiple listeners. The first one refers to the ability of the packet capture engine to accurately assign to each incoming packet a timestamp with the moment the packet arrived to the CPU. The latter refers to the ability of the capture engine to feed the same incoming traffic to different user-level applications. Those features will be explicitly remarked in the following subsections when an engine implements them.

4.1 PF_RING DNA

PF_RING Direct NIC Access (DNA) [17, 54] is a framework and engine to capture packets based on Intel 1/10 Gb/s cards. This engine implements pre-allocation and re-use of memory in all its processes. PF_RING DNA also allows building parallel paths from hardware receive queues to user processes, i.e., it allows a CPU core to be assigned to each receive queue whose memory can be allocated observing NUMA nodes, thus permitting the exploitation of memory affinity techniques.

Unlike the other proposals, PF_RING implements full *zero-copy*, i.e., PF_RING maps userspace memory into the DMA-able memory region of the driver allowing users' applications to access to card registers and data directly in a DNA fashion. This avoids the intermediation of the kernel packet buffer and reduces the number of copies. As previously noted, however, this is at the expense of a slight weakness to errors from user applications not following the PF_RING DNA API (which explicitly does not allow incorrect memory accesses) and this may potentially cause system crashes. In the rest of the proposals, direct accesses to the NIC are protected. PF_RING DNA behavior is shown in Fig. 7, where it can be observed that some of the steps that the NAPI approach follows disappear due to the use of the zero-copy technique.

PF_RING's API provides a set of functions for managing network devices and capturing incoming traffic. It works as follows: first, the network application must be registered with `pfring_set_application_name()`. Before starting the capture process, the socket descriptor can be configured via several functions, such as `pfring_set_{direction|mode|duration}()`. Once the socket is properly configured, traffic reception is enabled using the `pfring_enable_ring()` call. After this initialization process, user applications can receive new packets by calling the `pfring_recv()` function. Finally, when the user finishes capturing traffic `pfring_shutdown()` and `pfring_close()` functions are called. This process has to be replicated for each receive queue, as each user application will only receive the traffic corresponding to the RX queue the socket was configured for.

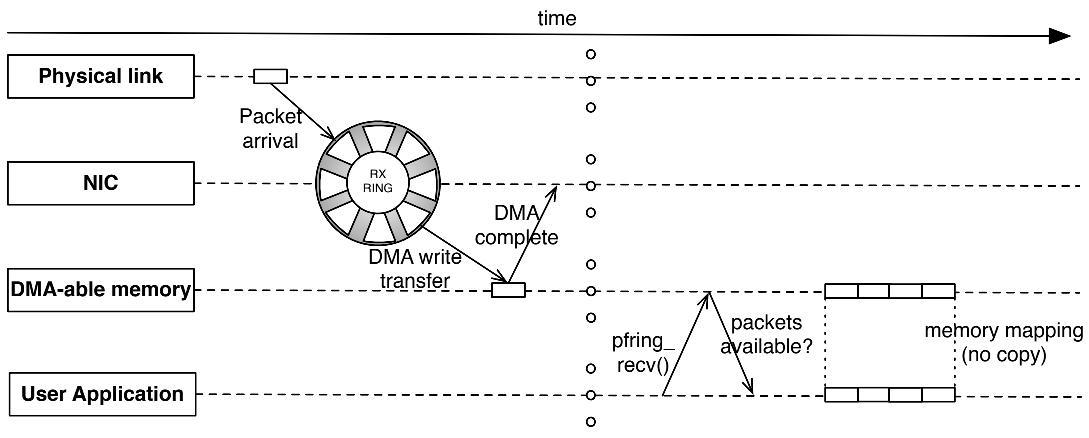


Figure 7: PF_RING DNA's RX scheme

As one of the major advantages of this solution, PF_RING's API comes with a set of wrappers for the above-mentioned functions providing extensive flexibility and ease of use, essentially following the *de facto* standard of the libpcap library. Additionally, the API provides a set of functions for applying filtering rules (for example, BPF filters), network bridging, and IP reassembly.

4.2 PacketShader

The authors of PacketShader (PS) [21, 55] developed their own packet capture engine to highly optimize the traffic capture module as a first step in the process of developing a software router based on General-Purpose Graphic Processing Units (GPGPU) able to work at multi-10 Gb/s rates. However, their efforts are applicable to any generic task that involves capturing and processing packets. They apply memory pre-allocation and re-use: specifically, two consecutive large memory regions are allocated: one for the packet data, and another for its metadata. Each buffer has fixed-size cells corresponding to the data and metadata for one packet. The size for each cell of packet data is set to 2048 bytes, which corresponds to the next highest power of two for the standard Ethernet MTU. Metadata structures are compacted from 208 bytes (as used by Linux's kernel) to only 8 bytes (96%) removing fields unnecessary for many networking tasks.

Additionally, PS implements memory mapping to those data and metadata buffers, thus allowing users to avoid additional copies when accessing the information. In this regard, the authors highlight the importance of NUMA-aware data placement in the performance of its engine. Similarly, it provides parallelism between different RX queues, as their data may be independently processed at user level.

To reduce the per-packet processing overhead, batching techniques are used when a user-level application asks for new packets. For each batch requested, the driver copies data from the hardware descriptors to the above-mentioned packet data region and completes the corresponding metadata information. Once those copies are finished, the driver returns control to the user-level application which can now process the new packets without additional copies. In order to eliminate inherent cache misses, the modified device driver tries to prefetch the next packet's associated memory while still processing the previous one.

PS's API works as follows: (i) a user application opens a character device to communicate

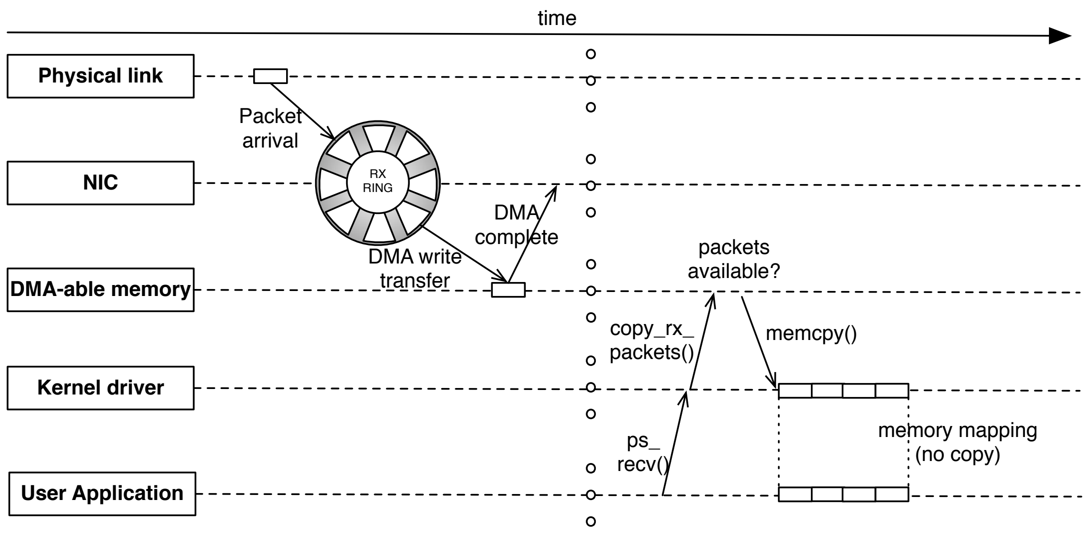


Figure 8: PacketShader's RX scheme

with the driver using the `ps_init_handle()` function, (ii) the application is attached to a given reception device (queue) using an `ioctl()` call, namely `ps_attach_rx_device()`, and (iii) kernel memory is allocated and mapped to userspace, in order to exchange data with the driver, using `ps_alloc_chunk()`. Then, when the user application requests new packets by means of an `ioctl()`, `ps_recv_chunk()`, PS driver copies a batch of them, if available, to the kernel packet buffer. PS kernel-user interaction during the reception process is summarized in Fig. 8.

4.3 netmap

The netmap [34, 56] proposal shares most of the characteristics of PacketShader's architecture. It applies memory pre-allocation during the initialization phase, buffers of fixed sizes (also 2048 bytes), batch processing and parallel direct paths. It also implements memory mapping techniques to allow users' applications to access kernel packet buffers (direct access to NIC is protected) with a simple and optimized data structure. Its similarities with PacketShader also apply to the user-kernel interaction policy (see Fig. 8), except that netmap implements *zero-copy* from the NIC to buffers that will later be mapped from user level. Differently from other *zero-copy* solutions, netmap makes an emphasis on system's security and scalability by making sure no critical kernel structure is mapped by user applications and thus entail a potential threat.

This simple data structure is referred to as a *netmap memory ring* and contains information such as the ring size, a pointer to the current position of the buffer (*cur*), the number of received packets in the buffer or the number of empty slots for each reception or transmission buffer (*avail*), a set of flags related to the status, the memory offset of the packet buffer, and the array with the metadata information; it has also one slot per packet that includes the length of the packet, the index in the packet buffer and some flags. Note that there is a *netmap ring* for each RSS queue, for both reception and transmission directions, to allow exploiting parallel direct paths.

Netmap's API usage is intuitive: first, a user process opens a netmap device and maps kernel buffers with an `ioctl()` call. To receive packets, the process polls the driver about the number of available packets with another `ioctl()` and, when the system call is over, the lengths and payloads of the packets are available for reading in the slots of the *netmap ring* data structure. Note that this operation mode makes a batch of packets accessible for reading in each operation. Additionally, netmap supports blocking mode through standard system calls, such as `poll()` or `select()`, using the corresponding netmap file descriptors as arguments for those standard system calls. In addition, netmap comes with a library that maps libpcap functions to their netmap equivalents, thus allowing user applications to exploit netmap features without needing to be recompiled. A distinctive feature of netmap is that it works in an extensive set of hardware solutions: Intel 10 Gb/s adapters and several 1 Gb/s adapters (Intel, RealTek and nVidia), and even Mellanos's infiniband adapters.

4.4 PFQ

PFQ [46, 57] is a novel packet capture engine that enables packet sniffing in user applications with a tunable degree of parallelism. The approach of PFQ is different from the previous ones studied. Instead of carrying out major modifications to the driver in order to skip the interrupt scheme of NAPI or mapping DMA-able memory and kernel packet buffers to user space, PFQ implements a general architecture supporting any NIC driver.

PFQ has been designed so that it benefits from the vanilla network driver managing the NIC's hardware details. Those drivers connect with PFQ by redefining those functions that previously connected them with the operating system's network stack. In the latest version, those redefinitions are automatically made by a set of scripts, so users are isolated from those low-level details.

PFQ's kernel module implements a new layer, named *Functional Engine*, where packets are delivered by the NIC's driver. This layer distributes the traffic across different active receive sockets, without limits on the number of queues than can receive a given packet. The distribution tasks are carried out by independent packet fetcher threads. Importantly, those fetcher threads are executed in parallel and push the incoming packets' data in the *Functional Engine* with minimal overhead due to lockless access control policy. PFQ's architecture allows several fetcher to push the same packet to different sockets, which may imply more that one packet copy. However, those additional packet copies have low impact due to the use of caching mechanisms. Note that, as each receive socket has an independent lock-free queue, the packet capture performance is not limited by the slowest application fetching traffic from a common source. This functionality circumvents one of the drawbacks of using the parallel paths technique, namely scenarios where packets of different flows or sessions must be analyzed by different applications as explained in Subsection 3.2. Fig. 9 shows a temporal scheme of the process of requesting a packet in this engine.

PFQ's API defines a `pfq` class which contains methods for device initialization and packet reception. Whenever a user wants to capture traffic: (i) a `pfq` object must be created using the provided C++ constructor, (ii) devices must be added to the object by calling its `add_device()` method, (iii) timestamping can be enabled using the `toggle_time_stamp()` method, and (iv) packet capture must be enabled using the `enable()` method. After initialization, each time a user wants to read a batch of packets, the `read()` method must be invoked. Using a custom C++ iterator provided by PFQ, users can read each packet in the received batch. When a user-level application has finished working with the `pfq` object, it is destroyed by means of its defined C++ destructor. A `stats()` method is also provided in order to obtain statistics about the received network traffic.

Moreover, PFQ supports high-level programming via functional programming languages,

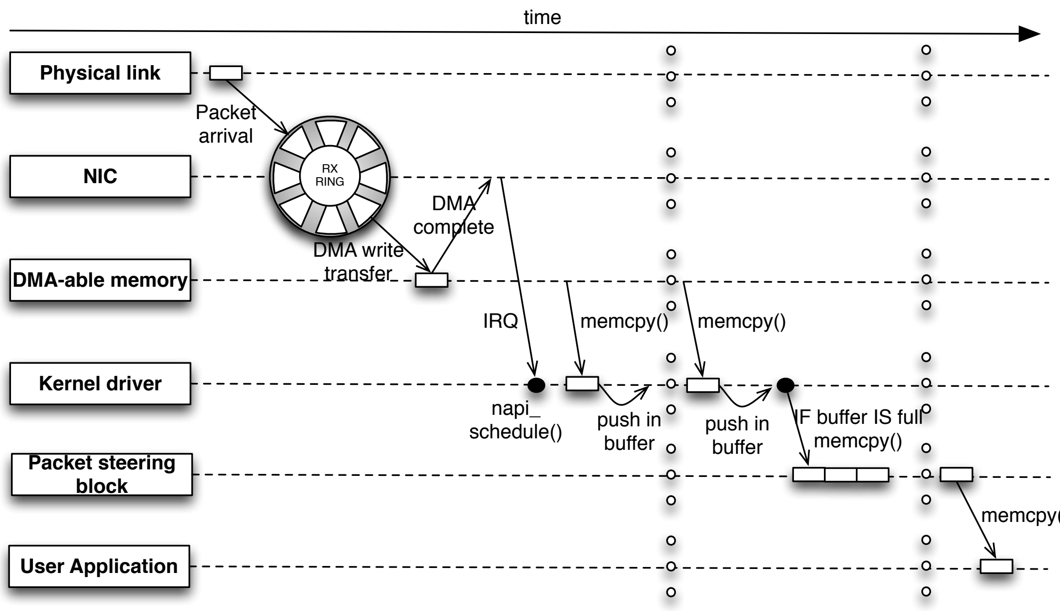


Figure 9: PFQ's RX scheme

PFQ-Lang [47]. By using PFQ-Lang, developers can rapidly develop network processing applications in a flexible way while coping with high-speed rates.

4.5 Intel DPDK

Intel's Data Plane Development Kit (DPDK) [48, 58] was created with the goal of providing a simple and complete framework for fast packet processing network application operating on the data plane. DPDK implements a new model for packet processing, following a modular approach. This way, users instantiate a set of worker threads, or listeners as we previously called them, which will be able to receive and send packet from/to a certain distributor, as shown in Fig. 10. Users will place their logic inside the worker modules, and will be able to configure the distributor thread connected to each worker for RX and TX purposes. Those connections are made by means of packet rings, and they are managed automatically by the library provided.

Intel's DPDK optimizes communications between the NIC and the distributor cores by pre-allocating and reusing data structures. These data structures are mapped from the user-level distributor threads and both memory and CPU affinity is carefully planned. Moreover, the multi-producer multi-consumer packet rings used to communicate the different threads in a DPDK-based application are generated using hugepages, which ensures these rings are always available on main memory and reduces the page fault overhead when accessing such regions. Note that each distributor thread will be in charge of dispatching the packets corresponding to a certain RSS queue from a certain NIC. Intel's DPDK architecture allows several worker threads to fetch packets from the same distributor, but each thread will only receive the set of packets previously requested. Additionally, a single worker thread may receive packets from different distributors.

An application making use of Intel DPDK has to initialize its resources by calling the

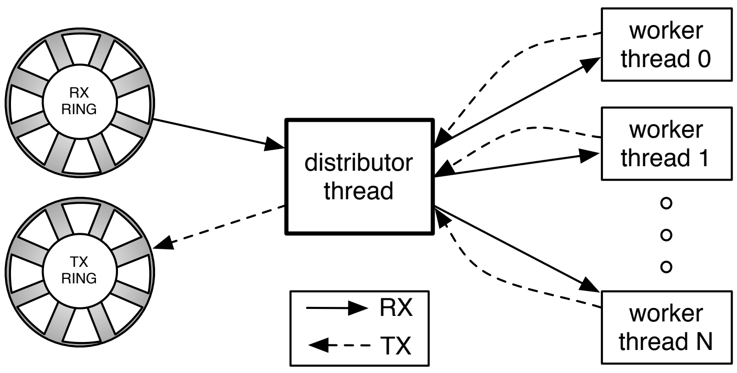


Figure 10: Intel DPDK's architecture

`rte_eal_init()` function. Distributor threads will call the `rte_distributor_process()` function to begin capturing packets from the NIC. Meanwhile, worker threads make use of the `rte_distributor_get_pkt()()` call to request a new packet once it has finished processing the previous one (see [59]). These library calls isolate users from operating with the intermediate packet rings used to communicate with the different modules. In order to optimize packet-transfer operations, local cache lines are shared between the distributor and worker threads. This feature makes it impossible for two worker threads to process the same packet simultaneously. Thus, every time a worker thread is done with a packet, it must notify the distributor via the final parameter of the `rte_distributor_get_pkt()()` function, so the distributor knows this packet can be sent to another worker if requested. Note that the packets are distributed to the diverse worker threads using meta-structures pointing to the corresponding NIC's packet descriptor, which are the ones cloned when several workers ask for the same packet. This allows sharing packet data without additional copies at the expense of potentially locking the descriptors for a longer period.

In order to keep packets ordered they are identified via a tag (the 5-tuple hash calculated by the NIC), thus packets with the same tag will be orderly processed by all worker threads processing them. Note that this policy may include latency in the processing of some packets and packet order is only guaranteed between packets with the same tag. Additionally, workers can temporally stop processing packets by using the `rte_distributor_return_pkt()` function and resume their execution afterwards, which may be interesting in order to save CPU power depending on network load.

Finally, Intel DPDK comes with a set of libraries to ease the user's work with packets at the different network layers, and extra functionalities to manage issues such as IP fragmentation and TCP re-assembly.

4.6 HPCAP

HPCAP [49, 60] is a novel packet capture engine designed, unlike other approaches, to optimize incoming traffic storage into non-volatile storage devices. The design of HPCAP has also focused on using just one receive queue in order to avoid packet reordering issues.

To do so, for each NIC to be monitored, HPCAP instantiates an independent kernel-level thread assigned to a different RSS queue. Once the kernel threads are launched, they will

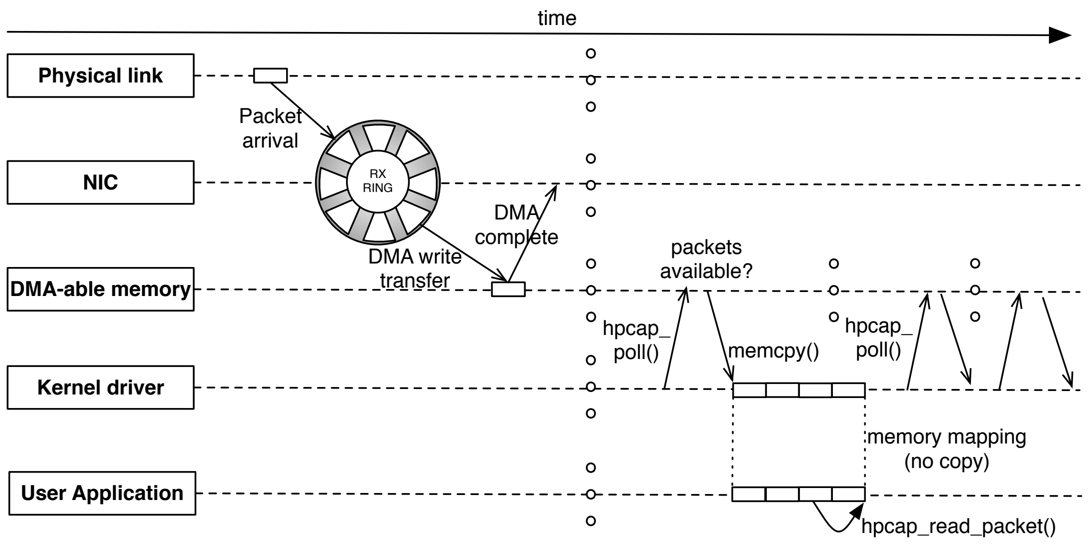


Figure 11: HPCAP's RX scheme

constantly poll their corresponding descriptor ring for incoming packets. If a new packet is detected, the thread copies it to a kernel-level packet buffer together with a header containing the packet's timestamp (second and nanosecond), capture length (packets may be capped) and actual length. The kernel-level packet buffers behave as a circular queue, which is allocated before the capture begins, avoiding dynamic per-packet allocations. Its size may be configured, with a default value of 1 GB, the maximum size for static buffers at kernel level. This limit will be surpassed in future versions by allocating memory using Linux's hugepages. A 1 GB size is convenient for providing robustness against traffic burstiness that may appear either at wire-level or when processing data in upper-layers.

The kernel buffers are mapped by user level applications and so are capable of accessing the data on it avoiding additional intermediate copies. These applications can be executed in any core not in use by a kernel polling thread, but in order to maximize performance, they should be executed in a core belonging to the same NUMA node as the core where the corresponding kernel polling thread is executed. A read pointer updated by the user application and a write pointer updated from the kernel polling thread control access to the buffers to prevent invalid read/write operations. Note that, if there are several user applications, or listeners, when accessing the packets from the same buffer, the packet read throughput will limited by the slowest listener. HPCAP provides an API for the listeners to manage devices and access packet data in isolation from the management of the read and write pointers.

As the packet capture and packet processing tasks are isolated, see Fig. 11, both tasks can be overlapped in order to maximize the system's throughput. This also means that user-level applications can access packets in the active region of the buffer following a byte-stream oriented policy. That is, user applications can operate with blocks of bytes instead of following a per-packet basis. Both the packet buffer memory alignment and the byte-stream oriented data format are crucial in order to maximize performance when storing data in a non-volatile volume.

Another design goal of HPCAP has been to provide certain additional features. First, incoming packets captured by HPCAP are timestamped when the kernel polling thread copies them into the intermediate buffer [40]. Second, each of the said kernel-level buffers can be mapped by several user-level applications, multiple-listeners, but differently from other approaches without additional copies. This is achieved as HPCAP updates a general write pointer while each upper-layer application updates its own read pointer. Finally, a framework called M^{3}Omon [61] is provided together with HPCAP, which was constructed based on HPCAP's API. This framework makes network data accessible for end-applications with three different granularities, namely Multi-Router Traffic Grapher (MRTG) time series for incoming packets and bytes as well as concurrent active flows; incoming packets; and expired flow registers. All these data are accessible to any number of applications via loop methods, receiving a callback function pointer as argument, similar to the `pcap_loop()` method.

5 Testing traffic capture performance

A quantitative comparison between capture engines based on the literature is not possible for two reasons: first, the hardware used by the different studies is not equivalent —in terms of type and clock speed of the CPU, amount and clock speed of main memory, server architecture and number of network cards. Second, the performance metrics used in the different studies are not the same —with differences in the type of traffic and in the measurement of the burden on CPU or memory.

Consequently, let us first elaborate a fair basis for comparison of capture engines at 10 Gb/s rates. And, then, let us carry out our own quantitative comparison based on such a common basis using the same hardware.

5.1 General concerns

The first metric to be considered is the amount of traffic that an engine may process. We assume a fully-saturated link of 10 Gb/s, and both constant-sized packets and real trace with variable packet sizes are injected. The constant-size oriented tests aim at evaluate worst case scenarios. As the effort to capture a packet is almost constant but small packets have smaller time gap between consecutive packets and consequently less time to carry out any subsequent task, those scenarios with small-size packets are the most challenging. Unfortunately, small-sized packet traffic profiles are not uncommon on the Internet as for example Voice over IP (VoIP) traffic, distributed databases or even anomalous traffic [62].

According to 10GbE standard, 60-byte packets in a 10 Gb/s fully-saturated link gives a throughput in Mp/s of 14.88: 10^{10} / ((60 + 4 (CRC) + 8 (Preamble) + 12 (Inter-Frame Gap)) $\cdot$ 8), and an effective throughput of 7.14 Gb/s (due to the preamble and inter-frame gap overheads). Equivalently, if packet sizes grow to 64 bytes, the throughput in Mp/s decreases to 14.22 and the effective throughput rises to 7.27 Gb/s. Table 2 shows how those values evolve for the packet sizes used in our test experiments. In order to avoid dealing with these values that depend on packet sizes, we find it more intuitive to evaluate this metric in terms of the percentage of packets received for each scenario.

Similarly, it is important to agree if the four Ethernet CRC bytes are considered in the packet size or not. In the following analysis, when referring to X-byte packets, those X bytes will not take Ethernet's CRC into account. In order to avoid Ethernet management mechanisms to contaminate network measurements, pause frame negotiation and hardware offload capabilities must be disabled as shown in Section 6. If pause frame negotiation is enabled the receiver side could send a pause frame when it is congested and prevent the sender from transmitting new frames. On the other hand, offload mechanisms allow NICs to merge small packets belonging to the same flow into bigger ones and may affect network diagnosis algorithms.

Table 2: Maximum throughput in terms of packets and bits for different packet sizes in a fully-saturated 10GbE link

Max.	**Packet size (bytes, CRC excluded)**								
throughput	**60**	**64**	**128**	**256**	**512**	**750**	**1024**	**1250**	**1514**
Gb/s	7.14	7.27	8.42	9.14	9.55	9.69	9.77	9.81	9.84
Mp/s	14.88	14.21	8.22	4.46	2.33	1.62	1.19	0.98	0.82

It is worth remarking that extremely positive results may arise in short duration experiments due to caching effects. Thus experiments regarding packet capture performance must be carried out for a period of time such that the amount of traffic processed does not fit in system memory. Thus, all the experiments carried out in this section have been obtained by replaying the corresponding traffic over a period of 30 minutes.

In order to provide a comparison as fair as possible, capture performance results refer only to a simple packet receive and update counters application developed for each engine, without any additional characteristic except from timestamping when it was possible (see Table 1). In all cases, we have paid attention to NUMA affinity by executing capture threads in the processor the NIC is connected to, which is only possible when there are less concurrent threads than cores available in the target NUMA node. In fact, ignoring NUMA affinity entails extremely significant performance losses, especially in the case of the smallest packet size where performance may be halved.

Another metric that must be considered is the number of CPUs and usage made by a solution. More computational power is available for additional tasks if the number of CPUs and usage are low. However using more than a couple of CPU and RSS queue may cause collateral effects such as packets belonging to the same session or flow to be processed by different CPUs. This may be a vital drawback for certain monitoring applications. Instant CPU usage measurements are obtained using the `pidstat`[4] command, instead of `ps`[5] command that provides the amount of time that the process has been using the processor since it started.

The amount of system memory used is the third main metric to take into account. High memory requirements may increment the cost of the monitoring system, and limit the number of additional processes that can be simultaneously executed. Memory usage for a certain process can be obtained by means of the Linux `ps` command.

5.2 Captures engines performance evaluation

Our testbed setup consists of two machines (one for traffic generation purposes and another for receiving traffic and evaluation) directly connected through a 10 Gb/s fiber optic link. The receiver side is based on Intel Xeon with two 6-core processors each running at 2.30 GHz, with 128 GB of DDR3 RAM at 1,333 MHz and fitted with a 10 GbE Intel NIC based on the 82599 chip. The server motherboard model is Supermicro X9DR3-F with two processor sockets and three PCIe 3.0 slots per processor, directly connected to each processor, following a scheme similar to that depicted in Fig. 2(b). The NIC is connected to a slot corresponding to the first processor or NUMA node. The system runs an Ubuntu server 12.10 with a 3.8.0.29-generic kernel.

The sender uses a HitechGlobal HTG-V5TXT-PCIe card with a Xilinx Virtex-5 FPGA (XC5VTX240) and four 10 GbE SFP+ ports. Using such a hardware-based sender guarantees accurate packet interarrivals and 10 Gb/s throughput regardless of packet size. The sender

[4]http://linux.die.net/man/1/pidstat
[5]http://linux.die.net/man/1/ps

server also has an Intel 82599 NIC and a software traffic generator, which has been developed as a tool on top of PacketShader's [21] API capable of replaying PCAP traces at variable rates.

For our experiments, we have used both synthetic and real traffic. Synthetic traffic is sent by using the FPGA generator and consists of TCP segments encapsulated into fixed-sized Ethernet frames, forged with incremental IP addresses and TCP ports. Note that synthetic traffic allows us to test worst-case scenarios in terms of byte and packet throughput, but they are not useful for testing the flow-related modules. Real traffic is generated using a software generator replaying a trace consisting of a packet-level trace sniffed on an OC192 backbone link of a Tier-1 ISP located between San Jose and Los Angeles (both directions), available from CAIDA [63]. Replaying the backbone trace at line-rate leads to a throughput of 9.59 Gb/s[6], and 1.65 Mp/s.

In addition to the capture engines presented in this paper, we have considered it interesting to evaluate the packet capture performance offered by the traditional solution, i.e., the `ixgbe` driver following a NAPI approach plus the use of the PCAP library. It is worth pointing out that the behavior of the Linux version of netmap is to set the number of RSS queues to match the number of cores. Thus, we have had to modify the number of queues used in netmap by writing a 1 or a 0 in the `/sys/devices/system/cpu/cpuX/online` file to respectively enable or disable CPUs. Regarding netmap and PFQ, we evaluated their performance by respectively installing the netmap-aware `ixgbe` driver and the `ixgbe` vanilla driver compiled with a script shipped with PFQ. We wanted to evaluate each capture engine using a number of queues ranging from 1 to 12 (as our system has 12 CPUs). It is worth noting that in the case of Intel DPDK 11 is the maximum amount of queues reached, because the capture system needs one core to be reserved for management purposes, reducing by 1 the number of cores eligible for packet capture.

First, Fig. 12 aims to show both the worst-case scenario of a fully-saturated 10 GbE link (packets with a constant size of 60 bytes) and an average scenario. Note that the worst case represents an extremely demanding scenario, 14.88 Mp/s, but probably not very realistic given that the average Internet packet size is clearly larger [64].

In the worst-case scenario (left-hand side of Fig. 12) the traditional solution (`ixgbe` + PCAP library) reaches peak performance with 5 queues, capturing over 24.4% of the incoming packets. PF_RING DNA and Intel DPDK are the only capture engines that achieve line rate if fewer than 4 receive queues are used. PacketShader is also able to handle nearly the total throughput when the number of queues ranges between 1 and 4, after which point the performance declines. Netmap has a performance level similar to PacketShader with one queue, but capture performance worsens as more receive queues are used. Conversely, PFQ increases its performance while the number of queues rises to a maximum with four queues, when improvement stalls. Finally, HPCAP shows peak performance, capturing 97.6% of the traffic, when using two queues but this figure reduces as the number of queues increases.

In the average scenario shown on the left-hand side of Fig.12, capture with `ixgbe` shows the best performance using four or five queues, with 0.1% of incoming packets lost. In that same scenario, PF_RING, PacketShader, PFQ (with two or more queues), Intel DPDK and HPCAP are capable of working with 0% packet loss. Netmap suffers a slight packet loss ranging from 1% to 0.1% (reached with four queues).

In conclusion, our tests have confirmed that all capture engines suffer from scalability issues in the worst-case scenario. This effect becomes more relevant when the number of cores in use needs more than one NUMA node. To further investigate this phenomenon, Fig. 13 depicts the results for the packet sizes shown in Table 2 using one, six and twelve queues respectively.

PF_RING DNA shows the best results with one and six queues. It does not show packet losses for any scenarios except for those with packet sizes of 64 bytes and, even in this case,

[6]This is the maximum achievable speed due to the preamble and inter-frame gaps that the Ethernet protocol requires.

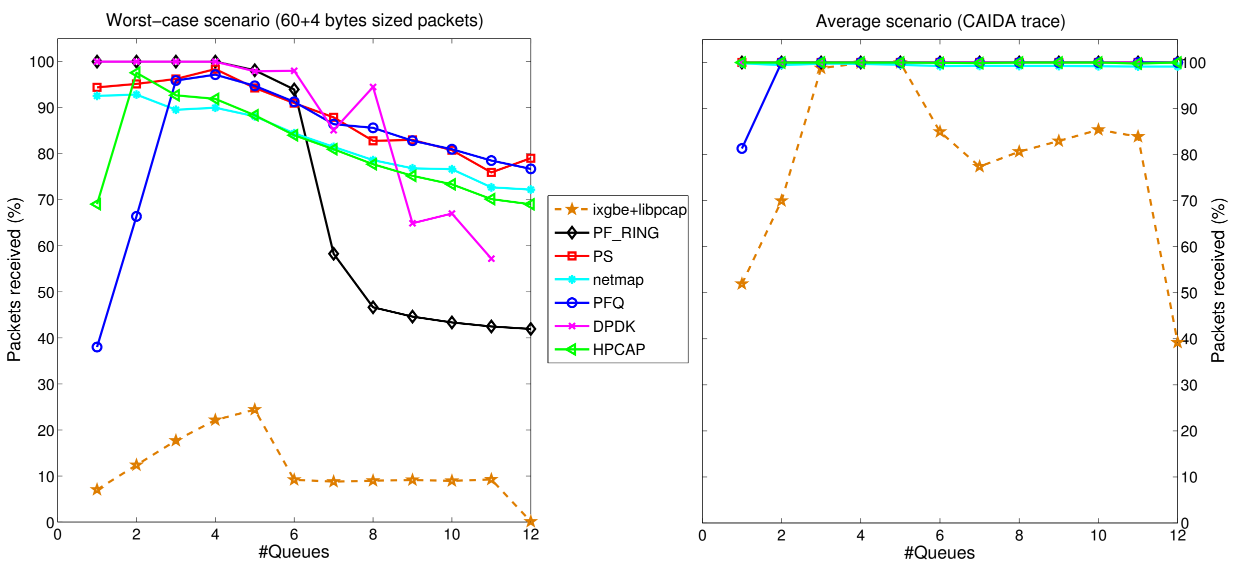


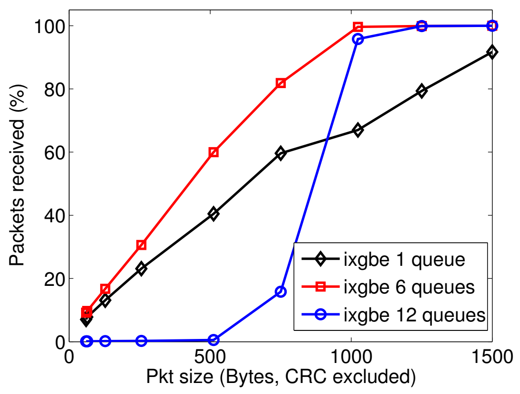
Figure 12: Engines' performance for worst and average scenarios

the figure is very low (about 4% with six queues and lower than 0.5% with one). Surprisingly, increasing packet sizes from 60 to 64 bytes entails a degradation in the PF_RING DNA performance, although the performance recovers 0% loss rates beyond these packet sizes. Note that, as stated before, larger packet sizes imply lower throughputs in terms of Mp/s. According to [34], investigation in this regard has shown that this behavior is due to the design of NICs and I/O bridges that make certain packet sizes fit better with their architectures.

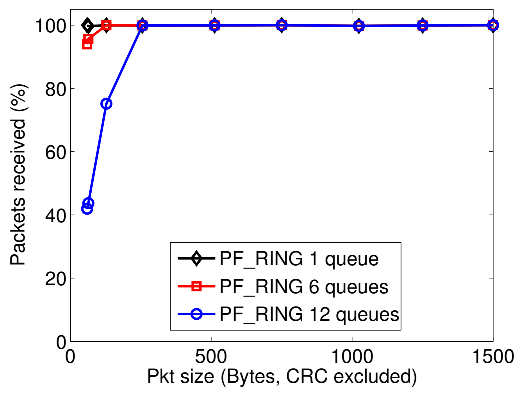
In a scenario where one single user-level application is unable to handle all received traffic, it may be of interest to use more than one receive queue (with one user-level application per queue). With the maximum number of queues, PacketShader and HPCAP have shown comparatively the best result, although, like PF_RING DNA, they perform better with a smaller number of queues. Specifically, for packet sizes larger or equal to 128 bytes, they achieve full packet received rates regardless of the number of queues. Conversely, Intel DPDK shows the worst results for the maximum number of queues, showing packet losses for packets below 750 bytes.

Analyzing PFQ's results, we note that this engine also achieves 100% received packet rates but, conversely to the other approaches, works better with several queues. It requires at least three to achieve no losses with packets of 128 bytes or more, whereas with one queue, packets must be larger or equal to 256 bytes to achieve full rates. This behavior was not unexpected due to the importance of parallelism in the implementation of PFQ.

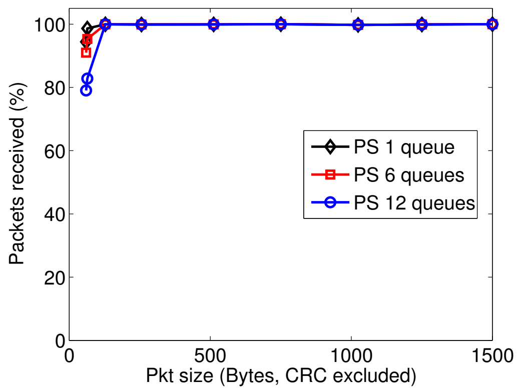
We have found that these engines may cover different scenarios, even the most demanding ones, distinguishing them on the basis of two criteria: whether or not we may assume the availability of multiple cores, and whether or not the traffic intensity (in Mp/s) is extremely high (for example, packet size averages smaller than 128 bytes, which is not very common). In other words, if the number of queues is not relevant, given that the capture machine has many cores available, or no other process is executing except for the capture process itself and the traffic is not maximal, PFQ seems to be a suitable option. On the other hand, if traffic intensity is near to maximum, PF_RING, PacketShader, netmap, Intel DPDK and HPCAP present a good compromise between the number of queues used and the performance offered.



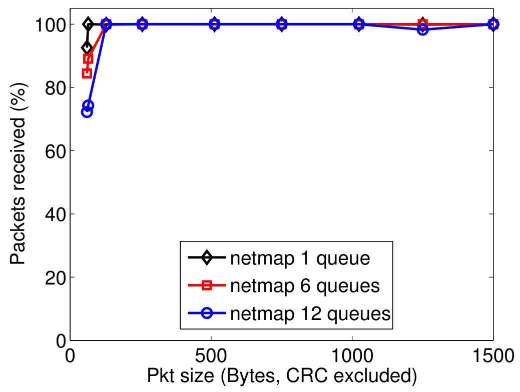
((a)) NAPI scheme



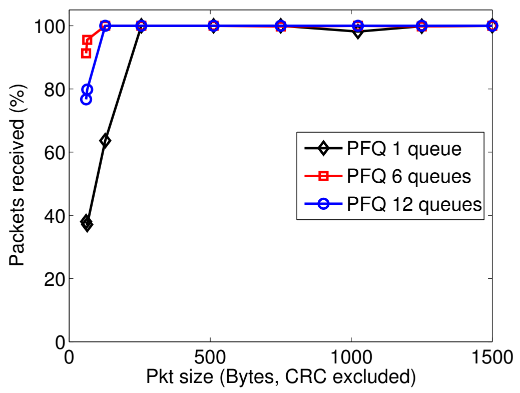
((b)) PF_RING DNA



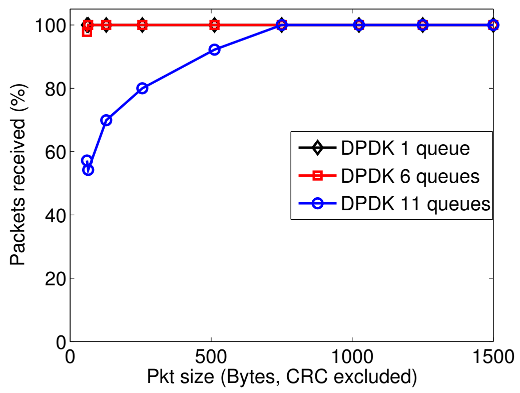
((c)) PacketShader



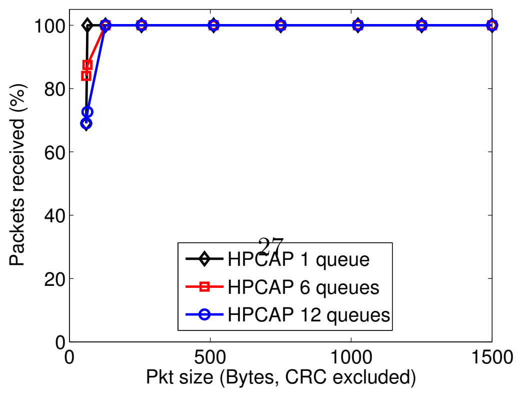
((d)) netmap



((e)) PFQ



((f)) Intel DPDK



((g)) HPCAP

Figure 13: Performance in terms of packets received for different numbers of queues and constant packet sizes for a fully-saturated 10 Gb/s link

Table 3: Memory and CPU usage in a 10Gb/s average scenario

Capture engine	**Number of RSS queues**	**Memory in use (MB)**	**Number of cores used**	**Average CPU usage (per active core)**
`ixgbe`	5	9.2	6	5×3.4% (kernel) + 82.7% (user)
PF_RING	1	110.3	1	75.8% (user)
PS	1	12.6	1	77.4% (user)
netmap	1	351.8	1	66.2 (user)
PFQ	1	425.2	1	99.9% (user)
DPDK	1	2192.4	2	99.2 (distributor) + 99.3 (worker)
HPCAP	1	1054.9	2	99.7 (kernel) + 99.8 (user)

Nonetheless, multi-queue scenarios are often not adequate. For example, accurate timestamps may be necessary [40], packet disorder may be a significant drawback (depending on the application running on top of the engine) [39], or it may simply be of interest to save cores for other tasks. In such a scenario, PF_RING DNA and Intel DPDK are great options, as they show (almost) full rates regardless of packet size even with only one queue (thus, avoiding any objections due to parallel paths).

In addition to the packet capture performance figures that each capture engine offers, users may also need to decide which solution they can use in terms of resource consumption. Table 3 shows the results for each capture engine in terms of CPU and memory usage. The table shows the resource consumption of each capture engine with the least resource-consuming configuration capable of capturing all packets (99.9% of the packets for `ixgbe`) mentioned previously: a CAIDA trace replayed at line-rate. Note that PF_RING, PacketShader, netmap and PFQ use as many cores as there are RSS queues. On the other hand, the default usage of `ixgbe` and Intel DPDK use one more core than the number of CPUs used. In the case of `ixgbe` the reason is that the kernel fetches packets from each queue into a different core, and a user-level application in a different core aggregates the traffic received by each queue. Intel DPDK uses one core to receive traffic from each RSS queue, where the distributor cores will be executed, and one additional core for each worker thread instantiated. HPCAP, on the other hand, uses two cores per RSS queue, as it instantiates one kernel-level thread to fetch the packets from the network and copy them to the intermediate buffer, and a user-level thread to process the packets already stored in that buffer.

6 How to use the packet capture engines

6.1 Getting started

Prior to setting up capture engines, some knowledge of the system architecture must be obtained in advance to exploit NUMA capabilities and perform and optimize scheduling.

The first step consists in getting an overview of the NUMA architecture and the devices attached to each node. To this end, the `lstopo` command should be used.

The command returns a text output with a tree scheme describing each NUMA node and the devices attached to it. Note that each output line containing L# references a processing core.In our case, two processors are present and each processor is assigned to a NUMA node with 6 processing cores and 64 GB of memory. Focusing on the NICs, two different NICs with two interfaces each are attached to the NUMA node 0.

Our system presents an architecture similar to Fig. 2(b) where PCIe lines are directly connected to one processor. In this scenario, our 10 GbE NIC corresponds to the interfaces `eth2` and `eth3` which are assigned to NUMA node 0. Capturing and processing tasks must be assigned to processing cores 0 to 5 in order to exploit memory locality.

The `numactl` command can be used to get an idea of how expensive this data transfer is in processing terms. Listing 1 shows the execution and output of this command. The output shows the available NUMA nodes specifying the processing cores and memory assigned as well as the distance matrix. Note that the figures shown in the distance matrix do not correspond to CPU cycles nor time measurements. Such numbers only provide a priority relationship where the higher value the slower the processor's access to that memory chunk.

```
numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 65503 MB
node 0 free: 61844 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 65536 MB
node 1 free: 60506 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

Listing 1: Usage example of the `numactl` command

Taking this memory access latency matrix into account, the memory allocation of the processes can be adjusted. Focusing on this example, if a capture or a processing task is assigned to cores 0 to 5, memory should be allocated in NUMA node 0. This configuration can be achieved by either using the `numactl` command or using the `libnuma` C API. Listing 2 gives an example of the former. In this case using the option `--membind` followed by a list of NUMA nodes, the memory used by the `my_program` application will be allocated on NUMA node 0 until no more memory is available and then subsequent allocations will use NUMA node 1. If no memory is available in either node 0 or 1, the application will terminate. Note that the execution of a program with `numactl --membind` does not guarantee CPU affinity.

```
numactl --membind=0,1 ./my_program
```

Listing 2: Usage example of the `numactl` command to allocate memory

Using `libnuma`[7], memory allocation can be assigned to a specific NUMA node via programing by using the C API shown in Listing 3

```
void *numa_alloc_local(size_t size);
void *numa_alloc_on_node(size_t size,int node);
```

Listing 3: Libnuma memory allocation API

Another task of paramount importance when running captures or processing applications is assigning tasks to processing cores. These can be assigned with the `taskset` command. As can be seen in Listing 4, using the parameter `-c` followed by the number of a processing core assigns the execution of `my_program` to the core indicated. If multiple processing cores are needed, a comma separated list can be defined. Additionally, the assignment process can be done programmatically by means of the `pthread` library as shown in Listing 5.

[7]`http://linux.die.net/man/3/numa`

```
taskset -c=1 ./my_program
taskset -c=1,3,5 ./my_program
```

Listing 4: Usage example of the `taskset` command

```
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *
  cpuset);
```

Listing 5: `pthread` process assignment API

The `isolcpus` kernel option may be used to maximize the efficiency of the assignment of tasks to processing cores. This option allows a set of processing cores to be isolated from the general kernel SMP balancing and scheduler algorithms. Thus, the only way a process can be assigned to such a set of cores is by explicitly attaching it through the `taskset` command or similar. With this approach, a capture or processing application can be exclusively assigned to a processing core. Listing 6 shows a sample kernel boot configuration in which processing cores 0,1,2,3,4,5 are isolated.

```
linux /boot/vmlinuz-3.8.0-29-generic root=UUID=b11691e7-f968-4023-aa28-3
  f5a4d831fa5 isolcpus=0,1,2,3,4,5 ro
```

Listing 6: Usage example of the `isolcpus` option

6.2 Setting up capture engines

The goal of this subsection is to provide a quick reference guide for the commands and applications used to configure the driver and receive traffic for each capture engine. All commands shown in this section except compilation-related ones must be executed with superuser privileges.

First, we describe how the affinity-aware tests have been used in the default mechanism, i.e., the `ixgbe` driver plus the PCAP library. The `ixgbe` driver version used is 3.11.33-k, and the version of the PCAP library is 1.1.1-10. With regard to the number of queues to be used, this value can be modified by means of the `RSS` parameter at driver load time (`insmod` command), as shown in Listing 7. The next step is to wake up the network interface and using the `ethtool` utility to disable pause frames (we do not want the network probe to stop the other side's transmission) and the offload options (in order to prevent the NIC from merging incoming packets into bigger ones and polluting our sampling of the network). Once the number of desired queues has been set, a different core must be assigned to fetch packets from each queue in order to obtain maximum performance. An example of how this can be done using the system `/proc` interface is shown the usage example. In the scenario shown, we have set 5 RSS queues to be used, assigning them to cores 0 to 4. Finally, we have developed a simple application that fetches incoming packets and counts them using the PCAP library and we have called it `test`. This application has been scheduled to run on core 5 (still in the same NUMA node as the 5 cores fetching packets from the different queues) via the `taskset` Linux command.

```
insmod ixgbe.ko MQ=1,1 RSS=5,5
ifconfig eth1 up promisc
ethtool -A eth1 rx off tx off
ethtool -K eth1 tso off gso off gro off lro off
#configure IRQ affinity
core=0
cat /proc/interrupts | grep eth1 |
  awk '{split($1,a,":");print a[1]}' |
    while read irq
    do
```

```
        echo $core > /proc/${irq}/smp_affinity_list
        core=$(( $core + 1))
    done
taskset -c 5 ./test eth1
```

Listing 7: Usage example of the `ixgbe` driver in an affinity-aware scenario

In the case of PF_RING, we used version 6.0.0.1. Once we have entered the downloaded folder, we must compile both the `pf_ring` and the PF_RING-aware version of our NIC driver (see Listing 8). When both drivers have been compiled, they can be installed using the script provided: `load_dna_driver.sh`. Changing the number of receive queues must be done by editing the driver load script, changing the RSS parameter in the line inserting the `ixgbe` driver into the system. The `load_dna_driver.sh` script also adjusts all interrupt affinity issues for each receive queue. To receive traffic using PF_RING, we executed the `pfcount_multichannel` command. The arguments of this program are as follows: `-i` indicates the device name, `-a` enables active packet waiting, `-e` sets reception only mode and `-g` specifies the thread affinity for the different queues. In the example, only one receive thread mapped to core 0 is used; if more threads are to be used, the core affinity for each one must be separated using ':'.

```
./load_dna_driver.sh
pfcount_multichannel -i dna0 -a  -e 1 -g 0
```

Listing 8: PF_RING usage example

With respect to PacketShader, its version 0.2 was used. We installed the driver using the `install.py` script provided, whose arguments are the number of RX and TX queues to be used by each NIC controlled by PacketShader. As Listing 9 shows, the engine provides an installation script to decide the number of receive queues, and it configures the interrupt affinity schedule. The bundle downloaded includes a sample application named `rxdump`, designed to dump incoming packet information through the standard output just as `tcpdump` would do. We have slightly modified this sample program so it only receives and counts incoming packets and launched it with the desired network device as its argument. The execution of this sample program was attached to core 0 via the `taskset` utility as the installation script set the receive queue management to this core.

```
./install.py 1 1
taskset -c 0 ./rxdump <args>
```

Listing 9: PacketShader usage example

We downloaded the latest version of netmap from the authors' github repository, specifically, we downloaded the version committed on April 1, 2014. In order to use it, first both the netmap kernel module and the netmap-aware `ixgbe` driver must be compiled. Before inserting any of those modules, the user must disable or enable CPUs in the system to accommodate the number of receive queues desired to be used. Note that netmap's default behavior is to use all available CPUs. The sample code shown in Listing 10 shows a way of doing that for one CPU (`num_cpus=1`). Once the corresponding CPUs have been disabled/enabled, the `netmap.ko` and `ixgbe.ko` drivers must be inserted in that order. Now it is time to wake up our interface, disable pause frames and offload settings, and configure interrupt affinity. Finally, the `pkt-gen` sample application can be used to receive network traffic. The `-i` parameter tells the program which device to receive traffic from, and the `-f rx` parameter indicates that the program is to work in rx-only mode. When using this application, the core affinity must be set via the `taskset` utility. Note that the program should be scheduled on the cores the queues' interrupts were previously mapped to. It is important to remember to re-enable all CPUs once you have finished using netmap.

```
#Set the number of active CPUs
totall_cpus=12
num_cpus=1
for i in $(seq 0 $(( $total_cpus - 1 )) )
do
        if [ $i -ge $num_cpus ]
        then
                echo 0 | tee /sys/devices/system/cpu/cpu${i}/online
        else
                echo 1 | tee /sys/devices/system/cpu/cpu${i}/online
        fi
done
insmod netmap_lin.ko
insmod ixgbe.ko
#wake up interface in promisc mode
#disable pause negotiation and offload settings
...
#configure IRQ affinity as with plain ixgbe
...
taskset -c 0 ./pkt-gen -i eth1 -f rx
```

Listing 10: netmap usage example

The version of PFQ, used is the 3.7. First we install the `pfq` driver and then the custom version of `ixgbe` setting the desired amount of queues, which is set to 2 in the example shown in Listing 11. We must then wake up the interface, disable pause frame and offload setting and set interrupt affinity, just as shown before in Listing 7. To receive packets from `eth1` using two queues with the right CPU affinity, we run the `pfq-counters` sample application. This application allows to instantiate different socket groups, each receiving all or a fraction of the traffic assigned to a certain interface. Those groups must be defined with their CPU binding via the `-t` parameter with the following syntax: `sock_id.core.iface[.queue.queue...]`. Where `core` is the CPU in which the thread receiving this socket's traffic will be executed, this core should be mapped not to collide with those configured to run the interface's interrupt code. Note that if no queues are specified, the traffic from all queues belonging to the specified interface will be processed.

```
modprobe ioatdma
modprobe dca
nqueues=2
pfq-load -q $ -c pfq.conf
#wake up interface in promisc mode
#disable pause negotiation and offload settings
...
#configure IRQ affinity as with plain ixgbe
...
core=$nqueues
pfq-counters -c 1514 -t 0.${core}.eth1
```

Listing 11: PFQ usage example

We used version 1.6.0r2 of Intel DPDK. As mentioned in the previous section, DPDK uses hugepages in order to gain performance, so the user must boot their system with the proper hugepages options. The example shown in Listing 12 allocates 4 hugepages each of 1 GB. Note that hugepages are evenly distributed between the different NUMA nodes of your system, which in our case means two hugepages per node. After booting the system, a `hugetlbfs` must be mounted for use by DPDK-based applications. Once the user has compiled DPDK's driver, both the system's `uio` and the compiled `igb_uio.ko` drivers must be loaded in that order. Intel's documentation encourages DPDK users to disable CPU frequency scaling governor in order to avoid performance losses due to power saving adjustments. Listing 12 shows a way of disabling it. Finally, the `testpmd` application is executed in interactive mode. Its invocation requires a large set of parameters which includes CPU affinity masks, queue configuration, the number of hugepages and mount point, number of memory channels, ... After properly launching `testpmd`, we must set the rx-only mode and give the capture start order.

```
# add to the grub boot line of your kernel
# ... default_hugepagesz=1G hugepagesz=1G hugepages=4
mount -t hugetlbfs -o pagesize=1G,size=4G none    /mnt/huge

modprobe uio
insmod igb_uio.ko
#properly set CPU's scaling governor
for g in /sys/devices/system/cpu/*/cpufreq/scaling_governor
do
  echo performance >$g
done
./testpmd <... parameter list ...>
testpmd> set fwd rxonly
testpmd> start
```

Listing 12: Intel DPDK usage example

Finally, we have used the version 4 of the HPCAP capture engine. This solution comes with a configuration file that the user may edit to change the engine's settings. A complete documentation of this file can be found in HPCAP's github repository. Once the parameters have been properly set, the user launches the `install_hpcap.bash` script which compiles the code, installs the driver, and configures the interface settings (interrupt affinity included). After running the installation script, users can use the `hpcapdd` sample application to receive traffic from the network, as shown in Listing 13. This application receives both the interface and queue indexes to receive traffic from (in the example, the application will receive traffic from interface `hpcap0`'s queue 0). The third argument is a directory path the program will write the incoming traffic to, but a null value means that nothing will be written (packets will only be captured from the network). CPU-affinity scheduling must be done with the `taskset` command. In our example, the `hpcapdd` application was scheduled in core 1 because the kernel-level thread was being executed in core 0 (which was set in the configuration file).

```
./install_hpcap.bash
taskset -c 1 ./hpcapdd 1 0 null
```

Listing 13: HPCAP usage example

7 Use cases of novel capture engines

Thus far, we have reviewed the first three layers of commodity high-performance network systems, NIC, driver and framework, paying special attention to the combination of the last two in a capture engine. In this section, we turn our attention to the upper layer: the final services or applications that are built on top of such engines.

Although the optimization of network drivers and capture engines is required to reach high performance, it has been shown to be insufficient in a final system [65]. In other words, if the high-level application is not capable of processing all traffic captured and provided by the capture engine, then we will only have shifted the bottleneck to the upper layer and we will not have solved the problem. Thus, regardless of the capture engine chosen to capture packets, application developers must follow ideas equivalent to those presented in Section 3.2: memory pre-allocation and reuse [66], tailored memory structures [67], exploiting inherent parallelism [68] and affinity-aware planning [61].

Table 4: Summary of the performance and characteristics of a set of high-performance network applications using commodity hardware

System Name	**Category**	**Capture Engine**	**Application**	**Throughput**	**Comments**
PacketShader [21]	Software Routers	PacketShader	IPv4 forwarding IPv6 forwarding OpenFlow Switch IPSec gateway	39 Gb/s 38 Gb/s 32 Gb/s 10.2 Gb/s	Packets of 64B
Ad-hoc version Click router [66]	Software Routers	netmap	IPv4 forwarding	6-8 Gb/s	Packets of 64B
MIDeA [68]	NID	PF_RING	Snort NID	7.2 Gb/s Below 2 Gb/s 5.7 Gb/s	Packets of 1500B Packets of 200B Real traces
Szabó et al. [69]	Traffic classification	PF_RING	DPI Connection pattern Port based	6.7 Gb/s	Real traces
Santiago et al. [70]	Traffic classification	PacketShader	Statistical classification	10 Gb/s	Packets of 64B
hpcapdd [50]	Monitoring	HPCAP	Packet storage	10 Gb/s	Packets of 64B
ffProbe [71]	Monitoring	PF_RING DNA	Netflow construction	10 Gb/s 7 Gb/s	Packets of 500B Packets of 60B
VoIPCallMon [67]	Monitoring	HPCAP	VoIP tracker	10 Gb/s	Codec G.711
Blockmon [72]	Monitoring	PFQ	Heavy hitters SYN flooding VoIP anomaly	3.8 Gb/s 5.5 Gb/s 10 Gb/s	 Codec G.711

We have found in the literature a number of final systems whose high performance is achieved by applying the techniques exposed along this tutorial. The rest of this section is devoted to survey them. Such survey may serve as the state-of-the-art bounds that any novel application should overcome to be of interest. Moreover, we believe the service examples reviewed may awaken new ideas and utilities in both the research community and practitioners. Table 4 summarizes the performance and characteristics of some of these applications as described by their authors. Through being based on commodity hardware, their costs are less than a few thousand dollars, apart from the cost of the common 10 Gb/s NICs.

Importantly, we note that some approaches are based on systems whose first step lies in the distribution of the load between several subsystems or clusters [73]. These subsystems may also work in isolation but at a lower rate. Thus, in order to show a fair performance comparison, we include the results for isolated systems instead of sets of distributed ones. Note that, in all cases leveraging an external traffic splitter (at higher cost) or with *ad-hoc* traffic balancing schemes, the load could be distributed over different machines to increase overall performance nearly linearly.

7.1 Software Routers

The use of commodity hardware to perform high-speed tasks started with the significant increase in popularity achieved by software routers in recent years. Software routers present some interesting advantages with respect to hardware-designed ones, essentially cost and flexibility. This increase has been strengthened by multiple examples of successful implementations and by the appearance of GPGPUs [74] which multiply the parallelism between processes while the cost remains low.

The authors of PacketShader [21], as stated in Section 4, developed their own network applications with both novel and optimized packet capture characteristics, but, in fact, their final target was to develop a software router able to work at multi-10 Gb/s rates. To this end, they proposed to move the routing process from the CPU to GPGPUs, where hundreds of threads can be executed in parallel. As most software routers operate on packet headers, the use of GPGPUs and parallel threads fits perfectly. Therefore, it is intuitive to bind each received packet to a thread in a GPGPU, multiplying the capacity of the router by the number of concurrent threads in each GPGPU. The results are astonishing given the use of commodity hardware and software solutions. IPv4 forwarding service achieves a throughput of 39 Gb/s with 64 byte packets, and even better results for larger packet sizes in a unique machine. The results for IPv6 forwarding are only slightly lower: 38 Gb/s. In addition to IP routing, the authors also evaluated the performance of their approach working as an OpenFlow Switch and an IPSec gateway. The results show that they are able to switch at 32 Gb/s, and they obtained a throughput of 10.2 Gb/s for IPSec overcoming commercial solutions.

Similarly to PacketShader's approach, the authors that proposed netmap illustrated their engine's functionality with a router software application [66], specifically, a Click Modular Router developed about fifteen years ago [65]. Conversely to PacketShader system, they neither use GPGPUs to parallelize tasks nor any further code optimization, thus the performance at application-level was lower, about 2 Gb/s with 64 byte packets although the capture engine worked at much higher rates. However thanks to this road block in the study of capture engines, they found that the capacities these novel devices achieved was far superior to many of the applications developed decades ago. In the specific case of Click, the authors pinpointed that the process of allocating memory in the C++ code was not ideal. In the original version, two blocks of memory were reserved per packet: one for the payload and another for its descriptor. However, this was not necessary as the memory can be recycled inside the code to avoid the allocation of new blocks and using fixed-size objects. The improvement ranges between 3x and 4x depending on the size of the batches, which represents a significant gain.

7.2 Network Intrusion Detection (NID) Systems

NID has become one of the most active research topics in the field of monitoring given its importance in network security. There are essentially two approaches to implement NID systems: those based on identifying (anomalous) characteristics of the traffic (for example, the distribution of port numbers' popularity) and those related to Deep Packet Inspection (DPI), which basically consists in searching for a given signature in the traffic payload. While the former typically results in faster speeds, the use of DPI approaches tends to be more accurate.

The authors in [68] evaluated this latter option proposing a full software implementation (called MIDeA) based on the PF_RING as capture engine. As application, they present a prototype implementation of a NID system based on Snort, the *de facto* standard software for this purpose, which includes more than 8,192 rules and 193,000 strings for string matching purposes. Similar to the previously explained PacketShader application, the key to its implementation is the use of GPGPUs. Especially, they optimized the way the application loaded data from/to the

GPGPU by adjusting data transfers to multiples of the minimum size for memory transaction on the GPGPU used. The results show that their system is able to achieve 7.2 Gb/s for synthetic traces in the ideal scenario of 1,500 byte packets. This represented an improvement of more than 250% over traditional multi-core implementations. However, the performance remains below 2 Gb/s in the case of packet sizes of 200 bytes. While this presents a significant reduction, it is worth noting that the average Internet packet size is clearly larger than such 200 bytes. In fact, when the system is evaluated with real traces, it achieves rates of 5.7 Gb/s.

7.3 Traffic Classification Technology

Traffic classification technology has gained in importance in recent years, as it has proved useful in tasks such as accounting, security, service differentiation policies, network design and research [75]. Since its inception to date, the research community has paid special attention to improving the accuracy of this technology, but it has not been until recently that the evaluation of their performance has gained relevance. Thus, some of the most accurate mechanisms have seen that their execution on high-speed networks is hardly likely. This has increased the interest in mechanisms to reduce the application burden required by classification. These mechanisms are essentially DPI and Machine Learning (ML) tools [76], once port-based classification has been ruled out because of the widespread use of random port numbers by P2P and VoIP applications.

In this regard, the authors in [69] show a system to classify traffic by leveraging both DPI and connection patterns (i.e., analyzing the interaction in terms of number of connections or ports involved in inter-host communications). The capture engine is implemented as a part of the system, but its foundations are equivalent to PF_RING. To deal with PF_RING's packet rates, the authors also exploit the parallelism provided by GPGPUs. In this case, the authors pay attention to the fact that GPGPUs' fast-cache memory tends to be too small to allocate the state machines that their traffic classification system requires. Thus, the authors propose to implement such state machines using the Zobrist hashing algorithm. Basically, this reduces memory requirements of state machines, which enables their allocation in cached memory. The throughput achieves a rate of 6.7 Gb/s with real traces —packet sizes of approximately 500 bytes. Again, this example shows that adapting applications to the capacities of novel hardware, in this case GPGPUs, is an essential step in obtaining the best performance.

The authors in [70] present a software-based statistical traffic classification engine that exploits the size of the first few packets of every observed flow. The application uses PacketShader as the packet engine. Unlike the previously explained application proposed in [69], this classification engine is not based on the utilization of GPGPUs, but runs only on commodity multi-core hardware. In addition to the use of PacketShader as capture engine and the lightweight statistical technique as classifier, the remarkable classification rates achieved are made possible by a careful tuning of critical parameters of both the hardware environment and the software application itself. In particular, the proposed system properly sets memory and CPU affinity of different threads composing it, processes packets in a batch-oriented fashion (replicating batch processing ideas from PacketShader at user level), reuses memory structures for flow storing, exploits multi-core parallelism overlapping the different tasks (namely, reception, flow-handling and classification) while asynchronously communicating them by means of intermediate buffers (chunk and job rings). The system achieves wire-speed classification in the worst-case scenario of 64 byte packets (10 Gb/s and even reaches 20 Gb/s when using two 10 GbE NICs and real traces (average packet size about 750 bytes).

7.4 Other monitoring tasks

The authors in [50] dig into the most intuitive service to build over a packet capture engine, packet storage in non-volatile drives. They present an application named `hpcapdd`, that running on the HPCAP engine that is able to store packets in commodity hard-drives at 10 Gb/s rates for all packet sizes. The contributions of this application are to exploit affinity by automatically executing threads in the same NUMA node, accessing hard drives to store packets as a stream of consecutive bytes instead of following a packet-by-packet fashion and using a huge intermediate buffer to handle the irregular throughput of mechanical hard drives.

ffProbe [71] is an implementation of a NetFlow probe [77], i.e., a probe that constructs a flow register for any consecutive set of packets sharing header information, e.g. the same IP addresses, port numbers and layer-4 protocol. These registers typically comprise the number of packets and bytes. NetFlow has become a fundamental tool for any network manager. ffProbe runs over the PF_RING DNA engine and has been proved to sustain 10 Gb/s rates with packets of about 500 bytes and rates over 7 Gb/s in the worst-case scenario of minimal packet size. The keys of the implementation of ffProbe is to leverage the concurrence capacity that PF_RING DNA provides by applying a hash function to packet headers and forwarding each of them to different concurrent processes. Note that this is possible as all packets comprising a flow share header fields used to construct flows. Additionally, ffProbe divides the work of constructing flows, looking for expired ones and the work of exporting to different processes. Due to the use of prefetching flow and memory caches the total latency is reduced.

The authors in [67] focus their attention on how HPCAP engine may turn out useful to monitor VoIP networks. They proposed the system, VoIPCallMon, which is able to track IP calls at aggregate rates of 10 Gb/s assuming codec G.711 with a packet size of 200 bytes. The heart of this proposal is dealing with the high rates served by bottom layers as well as the fast construction of flow records. To this end, the authors designed several tailored data-structures so that after any flow insertion and exportation, the active-flow list remained sorted.

Finally, we mention Blockmon [72,78] which is neither an application nor a system by itself but a framework to implement monitoring applications bearing in mind flexibility and re-usability, without losing sight of high performance. For example, reading packets, applying a level-4 filter and exporting a NetFlow registers. The authors leveraged PFQ as capture engine, and provided several sample applications running on it. Specifically, heavy hitter statistics (flows whose number of packets or bytes are over given thresholds), SYN flooding detection and VoIP anomaly detection. The authors do not provide many details of the implementations but report rates ranging between 3.8 and 10 Gb/s. These figures represent a 15% cut in performance compared to PFQ's capture process.

8 Summary, Lessons Learned and Conclusion

The use of commodity hardware on high-performance network tasks has opened an exciting scenario where even the hardest task can be carried out by a flexible, extensible, adaptable and even inexpensive system. Examples of these tasks that have been enriched by this novel paradigm are applications such as software routers, anomaly and intrusion detection, traffic classification, and VoIP monitoring.

Unfortunately, the process of developing a high-performance networking task on commodity hardware from scratch may turn out to be a non-trivial process composed of a set of thorny sub-tasks, each of which presents fine-tuned configuration details. In this light, this work has aimed at providing practitioners and researchers with a road-map to the exploration of this useful paradigm.

Such road-map can be summarized by means of the following lessons learned and pieces of advice sprinkled throughout this tutorial:

- Both default NIC drivers and network stack are shown to be insufficient to provide application layer with packets at multi-Gb/s rates. Depending on the scenario, bounds can be as low as 1 Gb/s.
- We have reviewed the main driver and stack limitations, and explained their respective countermeasures a capture engine should follow:
  - Dramatic cost of performing any operation at packet level:
    → Preallocating at driver load-time and reuse of memory during execution time.
    → Increasing packet data's access time by prefetching its contents while predecessor packets are still being processed.
    → Carrying out any task over a group of packets, not one-by-one, when possible.
  - Serialized access to traffic:
    → Exploiting parallel capacities of modern NICs by assigning a core to each RSS queue.
  - Multiple data copies from the wire to the user-level:
    → Mapping memory kernel regions at user-level.
  - Random placement of threads (or processes) across the available processors, leading to higher memory access latencies and cache inefficiency:
    → Carefully planning the thread-processor pair —threads must allocate memory in a chunk assigned to the NUMA node on top of which it is being executed.
    → Leveraging CPU and Interrupt affinity by setting both capture and interrupt threads to the same processor —thus exploiting cached data and load distribution.
  - Heavy kernel-to-userspace context switches:
    → Accessing as many packets as possible in a single system call: batches, streams.
- The industry and academia have applied most of these solutions giving rise to different and prominent off-the-shelf capture engines:
  - PF_RING DNA, PacketShader, netmap, PFQ, Intel DPDK and HPCAP.
- Practitioners and researchers interested in running applications over commodity hardware may base their development on one of these engines and skip most of the low-level details. They may make a decision based on both the additional features and the performance level offered by each engine.

  In terms of features we have identified:
  - Different APIs, some of them are similar to *de facto* standard, libpcap, and socket-alike.
  - Different level of timestamping precision and concurrent application support.
  - Different levels of compatibility with 1Gb NICs or NICs different from the Intel's 82598/82599 —which has become the *de facto* reference for all approaches.

  In terms of performance:
  - We have assessed that all engines surpass default NIC drivers and networking stacks' performance by far.

  - A certain scenario's defiance depends on both the available machine's topology and the traffic intensity in terms of Mp/s —essentially, the smallest packets are the most challenging ones.

  As take-away messages:

  - We have found that PFQ spans several advantages such as flexibility and ease of use.
  - PF_RING DNA, PacketShader, and netmap achieve full rates regardless of packet size and low resource utilization even with only one queue.
  - HPCAP provides accurate timestamping and enhanced packet storage.
  - Finally, Intel DPDK stands out given its extensive compatibility with NICs from several manufacturers.

- Finally, we remark the success of the explained capture engines by means of real-world applications, and state their significant state-of-the-art bounds. To illustrate this, we highlight that contemporary software routers based on commodity achieve rates above 30 Gb/s in tasks such as IP forwarding and OpenFlow switching. Moreover, flow construction, VoIP monitoring and packet storage applications achieve rates of 10 Gb/s, and even DPI and Snort-based applications can operate at rates higher than 5 Gb/s.

To conclude, we believe these lessons learned and pieces of advice may serve as a catalyst for the arrival of new high-performance network applications based on the pair of commodity-hardware and open software. We hope this also serves to stimulate further ideas and proposals to face the near and demanding future, paving the way for 40 Gb/s or even 100 Gb/s interfaces.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive and enriching comments. This research was carried out with the support of the Spanish National I+D *Packtrack* project (TEC2012-33754).

Selected References

[1] J. L. García-Dorado, A. Finamore, M. Mellia, M. Meo, and M. Munafò, "Characterization of ISP traffic: Trends, user habits, and access technology impact," *IEEE Transactions on Network and Service Management*, vol. 9, no. 2, pp. 142–155, 2012.

[2] B. Li, J. Springer, G. Bebis, and M. H. Gunes, "A survey of network flow applications," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 567 – 581, 2013.

[3] M.-S. Kim, H.-J. Kong, S.-C. Hong, S.-H. Chung, and J. Hong, "A flow-based method for abnormal network traffic detection," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2004.

[4] M. Mellia, R. Lo Cigno, and F. Neri, "Measuring IP and TCP behavior on edge nodes with tstat," *Computer Networks*, vol. 47, no. 1, pp. 1–21, 2005.

[5] M. Polychronakis, E. Markatos, K. Anagnostakis, and A. Oslebo, "Design of an application programming interface for IP network monitoring," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2004.

[6] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot, "Packet-level traffic measurements from the Sprint IP backbone," *IEEE Network*, vol. 17, pp. 6–16, 2003.

[7] J. Yu and X. Zhou, "Ultra-high-capacity DWDM transmission system for 100G and beyond," *IEEE Communications Magazine*, vol. 48, no. 4, pp. S56–S64, 2010.

[8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.

[9] G. Antichi, S. Giordano, D. Miller, and A. Moore, "Enabling open-source high speed network monitoring on NetFPGA," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2012.

[10] F. Yu, R. Katz, and T. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Proceedings of IEEE Conference on Network Protocols*, 2004.

[11] C. Meiners, J. Patel, E. Norige, E. Torng, and A. Liu, "Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems," in *Proceedings of USENIX Conference on Security*, 2010.

[12] Endace, "Packet capture performance evaluation," 2014, http://www.emulex.com, [15 February 2015].

[13] C. Systems, "Cisco network convergence system," 2013, http://www.cisco.com/en/US/products/ps13132/index.html, [15 February 2015].

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[15] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, "Comparing and improving current packet capturing solutions based on commodity hardware," in *Proceedings of ACM Internet Measurement Conference*, 2010.

[16] J. L. García-Dorado, F. Mata, J. Ramos, P. M. Santiago del Río, V. Moreno, and J. Aracil, "High-performance network traffic processing systems using commodity hardware," in *Data Traffic Monitoring and Analysis*. Springer Berlin Heidelberg, 2013, ch. 1, pp. 3–27.

[17] L. Rizzo, L. Deri, and A. Cardigliano, "10 Gbit/s line rate packet processing using commodity hardware: survey and new proposals," 2012, online: http://luca.ntop.org/10g.pdf [15 February 2015]. [Online]. Available: http://luca.ntop.org/10g.pdf

[18] S. Alcock, P. Lorier, and R. Nelson, "Libtrace: A packet capture and analysis library," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 42–48, 2012.

[19] S. Lee, K. Levanti, and H. Kim, "Network monitoring: Present and future," *Computer Networks*, vol. 65, no. 1, pp. 84–98, 2014.

[20] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: From packet capture to data analysis with netflow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.

[21] S. Han, K. Jang, K. S. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *Proceedings ACM SIGCOMM*, 2010.

[22] Intel, “82599 10 Gbe controller datasheet,” 2012, http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html, [15 February 2015].

[23] Microsoft, “Receive Side Scaling,” http://msdn.microsoft.com/en-us/library/windows/hardware/ff567236(v=vs.85).aspx [15 February 2015].

[24] S. Woo and K. Park, “Scalable TCP session monitoring with Symmetric Receive-Side Scaling,” *Technical report KAIST, http://www.ndsl.kaist.edu/~shinae/papers/TR-symRSS.pdf*, 2012.

[25] M. Dobrescu, K. Argyraki, and S. Ratnasamy, “Toward predictable performance in software packet-processing platforms,” in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[26] A. Lucent, “FP3: Breakthrough 400G network processor,” 2014, http://www3.alcatel-lucent.com/products/fp3/, [15 February 2015].

[27] LSI, “APP3000 Network,” 2014, http://www.lsi.com/products/mobile-communication-processors/pages/app-network-processors.aspx, [15 February 2015].

[28] Intel, “IXP4XX Product Line of Network Processors,” 2014, http://www.intel.com/p/en_US/embedded/hwsw/hardware/ixp-4xx, [15 February 2015].

[29] NetFPGA, “NetFPGA Project,” 2014, http://www.netfpga.org, [15 February 2015].

[30] Endace, “Endace EMULEX,” 2014, http://www.endace.com/, [15 February 2015].

[31] Cisco, “Network Analysis Module (NAM) Products,” 2014, http://www.cisco.com/go/nam, [15 February 2015].

[32] L. Zabala, A. Ferro, and A. Pineda, “Modelling packet capturing in a traffic monitoring system based on Linux,” in *Proceedings of Performance Evaluation of Computer and Telecommunication Systems*, 2012.

[33] L. Foundation, “NAPI,” 2014, http://www.linuxfoundation.org/collaborate/workgroups/networking/napi, [15 February 2015].

[34] L. Rizzo, “netmap: a novel framework for fast packet I/O,” in *Proceedings of USENIX Annual Technical Conference*, 2012.

[35] C. Schroder, *Linux networking cookbook*, ser. O’Reilly Media, 2007.

[36] C. Benvenuti, *Understanding Linux Network Internals*, ser. O’Reilly Media, 2005.

[37] G. Liao, X. Znu, and L. Bnuyan, “A new server I/O architecture for high speed networks,” in *Proceedings of Symposium on High-Performance Computer Architecture*, 2011.

[38] A. Papadogiannakis, G. Vasiliadis, D. Antoniades, M. Polychronakis, and E. Markatos, “Improving the performance of passive network monitoring applications with memory locality enhancements,” *Computer Communications*, vol. 35, no. 1, pp. 129–140, 2012.

[39] W. Wenji, P. DeMar, and M. Crawford, “Why can some advanced Ethernet NICs cause packet reordering?” *IEEE Communications Letters*, vol. 15, no. 2, pp. 253–255, 2011.

[40] V. Moreno, P. M. Santiago del Río, J. Ramos, J. Garnica, and J. L. García-Dorado, "Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines," *IEEE Communications Letters*, vol. 16, no. 11, pp. 1888–1891, 2012.

[41] W. Su, L. Zhang, D. Tang, and X. Gao, "Using direct cache access combined with integrated NIC architecture to accelerate network processing," in *Proceedings of IEEE Conference on High Performance Computing and IEEE Conference on Embedded Software and Systems*, 2012.

[42] L. Deri, "Improving passive packet capture: Beyond device polling," in *Proceedings of System Administration and Network Engineering Conference*, 2004.

[43] ——, "nCap: wire-speed packet capture and transmission," in *Proceedings of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2005.

[44] L. Rizzo, "Revisiting network I/O apis: The netmap framework," *ACM Queue*, vol. 10, no. 1, pp. 30–39, 2012.

[45] ——, "Portable packet processing modules for OS kernels," *IEEE Network*, vol. 28, no. 2, pp. 6–11, 2014.

[46] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi-gigabit packet capturing with multi-core commodity hardware," in *Proceedings of Passive and Active Measurement Conference*, 2012.

[47] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni, "A purely functional approach to packet processing," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2014.

[48] Intel, "Intel Data Plane Development Kit (Intel DPDK) Release Notes," 2014, http://www.intel.com/content/dam/www/public/us/en/documents/release-notes/intel-dpdk-release-notes.pdf, [15 February 2015].

[49] V. Moreno, "Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks," Master's thesis, Universidad Autónoma de Madrid, 2012, http://www.ii.uam.es/~vmoreno/Publications/morenoTFM2012.pdf, [15 February 2015].

[50] V. Moreno, P. M. Santiago del Río, J. Ramos, J. L. García-Dorado, I. Gonzalez, F. J. Gomez-Arribas, and J. Aracil, "Packet storage at multi-gigabit rates using off-the-shelf systems," in *Proceedings of IEEE International Conference on High Performance Computing and Communications*, 2014.

[51] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[52] M. Krasnyansky, "UIO-IXGBE," Online. https://opensource.qualcomm.com/wiki/UIO-IXGBE, 2012. [Online]. Available: https://opensource.qualcomm.com/wiki/UIO-IXGBE

[53] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proceedings of ACM Internet Measurement Conference*, 2010.

[54] ntop, "Libzero for DNA," 2014, http://www.ntop.org/products/pf_ring/libzero-for-dna, [15 February 2015].

[55] PacketShader, "Packet I/O Engine," 2012, http://shader.kaist.edu/packetshader/io_engine/index.html, [15 February 2015].

[56] netmap, "The fast packet I/O framework," 2014, http://info.iet.unipi.it/~luigi/netmap, [15 February 2015].

[57] PFQ, "PFQ homepage," 2015, http://netserv.iet.unipi.it/software/pfq, [15 February 2015].

[58] DPDK, "Data plane development kit," 2015, http://dpdk.org, [15 February 2015].

[59] Intel, "Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide," 2014, http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-dpdk-programmers-guide.pdf, [15 February 2015].

[60] HPCAP, "High-performance 10G network capture engine," 2015, http://github.com/hpcn-uam/HPCAP, [15 February 2015].

[61] V. Moreno, P. M. Santiago del Río, J. Ramos, D. Muelas, J. L. García-Dorado, F. J. Gomez-Arribas, and J. Aracil, "Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems," *International Journal of Network Management*, vol. 24, no. 4, pp. 221–234, 2014.

[62] M.-S. Kim, Y. J. Won, and J. W. Hong, "Characteristic analysis of Internet traffic from the perspective of flows," *Computer Communications*, vol. 29, no. 10, pp. 1639–1652, 2006.

[63] C. Walsworth, E. Aben, k. Claffy, and D. Andersen, "The CAIDA anonymized 2009 Internet traces," http://www.caida.org/data/passive/passive_2009_dataset.xml, [15 February 2015].

[64] CAIDA, "Traffic analysis research," http://www.caida.org/research/traffic-analysis/.

[65] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[66] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *Proceedings of IEEE INFOCOM*, 2012.

[67] J. L. García-Dorado, P. M. Santiago del Río, J. Ramos, D. Muelas, V. Moreno, J. E. Lopez de Vergara, and J. Aracil, "Low-cost and high-performance: VoIP monitoring and full-data retention at multi-Gb/s rates using commodity hardware," *International Journal of Network Management*, vol. 24, no. 3, pp. 181–199, 2014.

[68] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: a multi-parallel intrusion detection architecture," in *Proceedings of ACM Conference on Computer and Communications Security*, 2011.

[69] G. Szabó, I. Gódor, A. Veres, S. Malomsoky, and S. Molnár, "Traffic classification over Gbit speed with commodity hardware," *Journal of Communications Software and Systems*, vol. 5, no. 3, 2010.

[70] P. M. Santiago del Río, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil, "Wire-speed statistical classification of network traffic on commodity hardware," in *Proceedings of ACM Internet Measurement Conference*, 2012.

[71] M. Danelutto, L. Deri, and D. De Sensi, “Network monitoring on multicores with algorithmic skeletons,” 2011.

[72] A. Di Pietro, F. Huici, N. Bonelli, B. Trammell, P. Kastovsky, T. Groleat, S. Vaton, and M. Dusi, “Toward composable network traffic measurement,” in *Proceedings of IEEE INFOCOM*, 2013.

[73] F. Schneider, J. Wallerich, and A. Feldmann, “Packet capture in 10-Gigabit Ethernet environments using contemporary commodity hardware,” in *Proceedings of Passive and Active Measurement Conference*, 2007.

[74] J. Nickolls and W. Dally, “The GPU computing era,” *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[75] A. Callado, C. Kamienski, G. Szabo, B. Gero, J. Kelner, S. Fernandes, and D. Sadok, “A survey on Internet traffic identification,” *IEEE Communications Surveys & Tutorials*, vol. 11, no. 3, pp. 37–52, 2009.

[76] T. Nguyen and G. Armitage, “A survey of techniques for Internet traffic classification using machine learning,” *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[77] C. Systems, “White paper: Introduction to Cisco IOS NetFlow,” 2012, http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/white-paper-listing.html, [15 February 2015].

[78] M. Dusi, N. d’Heureuse, F. Huici, A. di Pietro, N. Bonelli, G. Bianchi, B. Trammell, and S. Niccolini, “Blockmon: Flexible and high-performance big data stream analytics platform and its use cases,” *NEC Technical Journal*, vol. 7, no. 2, pp. 102–106, 2012.