Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience. Lecture Notes in Computer Science, Volumen 7754. Springer, 2013. 3-27

DOI: http://dx.doi.org/10.1007/978-3-642-36784-7_1

High-Performance Network Traffic Processing Systems Using Commodity Hardware

José Luis García-Dorado, Felipe Mata, Javier Ramos,
Pedro M. Santiago del Río, Victor Moreno, and Javier Aracil

High Performance Computing and Networking,
Universidad Autónoma de Madrid,
Madrid, Spain

Abstract. The Internet has opened new avenues for information accessing and sharing in a variety of media formats. Such popularity has resulted in an increase of the amount of resources consumed in backbone links, whose capacities have witnessed numerous upgrades to cope with the ever-increasing demand for bandwidth. Consequently, network traffic processing at today's data transmission rates is a very demanding task, which has been traditionally accomplished by means of specialized hardware tailored to specific tasks. However, such approaches lack either of flexibility or extensibility—or both. As an alternative, the research community has pointed to the utilization of commodity hardware, which may provide flexible and extensible cost-aware solutions, ergo entailing large reductions of the operational and capital expenditure investments. In this chapter, we provide a survey-like introduction to high-performance network traffic processing using commodity hardware. We present the required background to understand the different solutions proposed in the literature to achieve high-speed lossless packet capture, which are reviewed and compared.

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

1 Introduction

Leveraging on the widespread availability of broadband access, the Internet has opened new avenues for information accessing and sharing in a variety of media formats. Such popularity has resulted in an increase of the amount of resources consumed in backbone links, whose capacities have witnessed numerous upgrades to cope with the ever-increasing demand for bandwidth. In addition, the Internet customers have obtained a strong position in the market, which has forced network operators to invest large amounts of money in traffic monitoring on attempts to guarantee the satisfaction of their customers—which may eventually entail a growth in operators' market share.

Nevertheless, keeping pace with such ever-increasing data transmission rates is a very demanding task, even if the applications built on top of a monitoring

system solely capture to disk the headers of the traversing packets, without further processing them. For instance, traffic monitoring at rates ranging from 100 Mb/s to 1 Gb/s was considered very challenging a few years ago, whereas contemporary commercial routers typically feature 10 Gb/s interfaces, reaching aggregated rates as high as 100 Tb/s.

As a consequence, network operators have entrusted to specialized HardWare (HW) devices (such as FPGA-based solutions, network processors or high-end commercial solutions) with their networks monitoring. These solutions are tailored to specific tasks of network monitoring, thus achieving a high-grade of performance—e.g., lossless capture. However, these alternatives for network traffic monitoring either lack of flexibility or extensibility (or both), which are mandatory nowadays for large-scale network monitoring. As a result, there have been some initiatives to provide some extra functionalities in network elements through supported Application Programming Interfaces (APIs) to allow the extension of the software part of their products—e.g., OpenFlow [1].

As an alternative, the research community has pointed to the utilization of commodity hardware based solutions [2]. Leveraging on commodity hardware to build network monitoring applications brings along several advantages when compared to commercial solutions, among which overhang the flexibility to adapt any network operation and management tasks (as well as to make the network maintenance easier), and the economies of scale of large-volume manufacturing in the PC-based ecosystem, ergo entailing large reductions of the operational and capital expenditures (OPEX and CAPEX) investments, respectively. To illustrate this, we highlight the special interest that software routers have recently awakened [3, 4]. Furthermore, the utilization of commodity hardware presents other advantages such as using energy-saving policies already implemented in PCs, and better availability of hardware/software updates enhancing extensibility [4].

To develop a network monitoring solution using commodity hardware, the first step is to optimize the default NIC driver to guarantee that the high-speed incoming packet stream is captured lossless. The main problem, the receive livelock, was studied and solved several years ago [5, 6]. The problem appears when, due to heavy network load, the system collapses because all its resources are destined to serve the per packet interrupts. The solution, which mitigates the interrupts in case of heavy network load, is now implemented in modern operating systems (Section 2.2), and specifically in the GNU Linux distribution, which we take in this chapter as the leading example to present the several capture engines proposed in the literature [4, 7–10]. These engines provide improvements at different levels of the networking stack, mainly at the driver and kernel levels. Another important point is the integration of the capturing capabilities with memory and CPU affinity aware techniques, which increase performance by enhancing the process locality.

The rest of the chapter is structured as follows. In Section 2 we provide the required background to understand the possibilities and limitations that contemporary commodity hardware provides for high-performance networking

tasks. Then, Section 3 describes the general solutions proposed to overcome such limitations. Section 4 details different packet capture engines proposed in the literature, as well as their performance evaluation and discussion. Finally, Section 5 concludes the chapter.

2 Background

2.1 Commodity Hardware: Low Cost, Flexibility and Scalability

Commodity computers are systems characterized by sharing a base instruction set and architecture (memory, I/O map and expansion capability) common to many different models, containing industry-standard PCI slots for expansion that enables a high degree of mechanical compatibility, and whose software is widely available off-the-self. These characteristics play a special role in the economies of scale of the commodity computer ecosystem, allowing large-volume manufacturing with low costs per unit. Furthermore, with the recent development of multi-core CPUs and off-the-self NICs, these computers may be used to capture and process network traffic at near wire-speed with little or no packet losses in 10 GbE networks [4].

On the one hand, the number of CPU cores within a single processor is continuously increasing, and nowadays it is common to find quad-core processors in commodity computers—and even several eight-core processors in commodity servers. On the other hand, modern NICs have also evolved significantly in the recent years calling both former capturing paradigms and hardware designs into question. One example of this evolution is Receive Side Scaling (RSS) technology developed by Intel [11] and Microsoft [12]. RSS allows NICs to distribute the network traffic load among different cores of a multi-core system, overcoming the bottleneck produced by single-core based processing and optimizing cache utilization. Specifically, RSS distributes traffic to different receive queues by means of a hash value, calculated over certain configurable fields of received packets and an indirection table. Each receive queue may be bound to different cores, thus balancing load across system resources.

As shown in Fig. 1, the Least Significant Bits (LSB) from the calculated hash are used as a key to access to an indirection table position. Such indirection table contains values used to assign the received data to a specific processing core. The standard hash function is a Toeplitz hash whose pseudocode is showed in Algorithm 1. The inputs for the function are: an array with the data to hash and a secret 40-byte key (K)—essentially a bitmask. The data array involves the following fields: IPv4/IPv6 source and destination addresses; TCP/UDP source and destination ports; and, optionally, IPv6 extension headers. The default secret key produces a hash that distributes traffic to each queue maintaining unidirectional flow-level coherency—packets containing same source and destination addresses and source and destination ports will be delivered to the same processing core. This behavior can be changed by modifying the secret key to distribute traffic based on other features. For example in [13] a solution for maintaining bidirectional flow-level (session-level) coherency is shown.

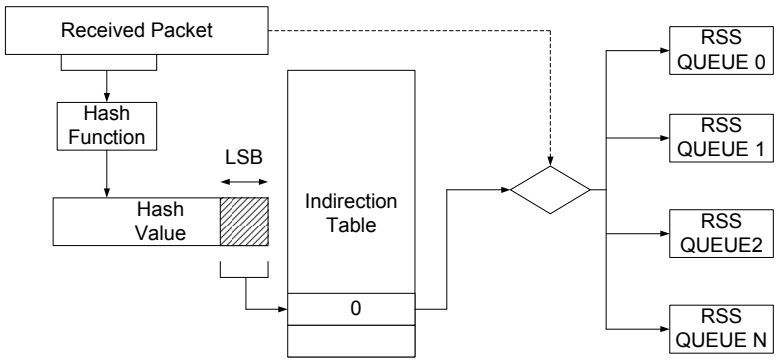


Fig. 1: RSS architecture

Modern NICs offer further features in addition to RSS technology. For example, advanced hardware filters can be programmed in Intel 10 GbE cards to distribute traffic to different cores based on rules. This functionality is called Flow Director and allows the NIC to filter packets by: Source/destination addresses and ports; Type Of Service value from IP header; Level 3 and 4 protocols; and, Vlan value and Ethertype.

Hardware/software interactions are also of paramount importance in commodity hardware systems. For example Non-Uniform Memory Access (NUMA) design has become the reference for multiprocessor architectures, and has been extensively used in high-speed traffic capturing and processing. In more detail, NUMA design splits available system memory between different Symmetric MultiProcessors (SMPs) assigning a memory chunk to each of them. The combination of a processor and a memory chunk is called NUMA node. Fig. 2 shows some examples of NUMA architectures. NUMA-memory distribution boosts up systems' performance as each processor can access in parallel to its own chunk of memory, reducing the CPU data starvation problem. Although NUMA architectures increase the performance in terms of cache misses and memory accesses [14], processes must be carefully scheduled to use the memory owned by the core in which they are being executed, avoiding accessing to other NUMA nodes.

Algorithm 1 Toeplitz standard algorithm

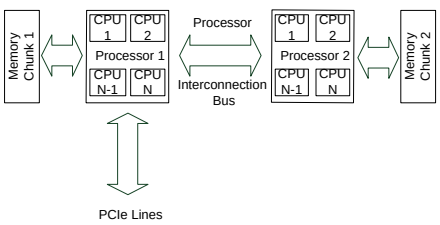
```
1: function ComputeHash(input[],K)
2:     result = 0
3:     for each bit b in input[] from left to right do
4:         if b == 1 then
5:             result^= left-most 32 bits of K
6:         shift K left 1 bit position
7:     return result
```

Essentially, the accesses from a core to its corresponding memory chunk results in a low data fetching latency, whereas accessing to other memory chunk increases this latency. To explode NUMA architectures, the NUMA-node distribution must be previously known as it varies across different hardware platforms. Using the `numactl`[1] utility a NUMA-node distance matrix may be obtained. This matrix represents the distance from each NUMA-node memory bank to the others. Thus, the higher the distance is, the higher the access latency to other NUMA nodes is. Other key aspect to get the most of NUMA systems is the interconnection between the different devices and the processors.

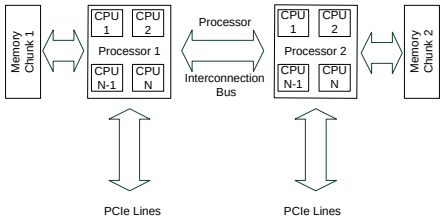
Generally, in a traffic capture scenario, NICs are connected to processors by means of PCI-Express (PCIe) buses. Depending on the used motherboard in the commodity hardware capture system, several interconnection patterns are possible. Fig. 2 shows the most likely to find schemes on actual motherboards. Specifically Fig. 2a shows an asymmetric architecture with all PCIe lines directly connected to a processor whereas Fig. 2b shows a symmetric scheme where PCIe lines are distributed among two processors. Figs. 2c and 2d show similar architectures with the difference of having their PCIe lines connected to one or several IO-hubs. IO-hubs not only connect PCIe buses but also USB or PCI buses as well as other devices with the consequent problem of sharing the bus between the IO-hub and the processor among different devices. All this aspects must be taken into account when setting up a capture system. For example, when a NIC is connected to PCIe assigned to a NUMA node, capturing threads must be executed on the corresponding cores of that NUMA node. Assigning capture threads to another NUMA node implies data transmission between processors using Processor Interconnection Bus which leads to performance degradation. One important implication of this fact is that having more capture threads than existing cores in a NUMA node may be not a good approach as data transmission between processors will exist. To obtain information about the assignment of a PCIe device to a processor, the following command can be executed on Linux systems `cat /sys/bus/pci/devices/PCI_ID/local_cpulist` where PCI_ID is the device identifier obtained by executing `lspci`[2] command.

All the previously mentioned characteristics make modern commodity computers highly attractive for high-speed network traffic monitoring, because their performance may be compared to today's specialized hardware, such as FPGAs (NetFPGA[3], Endace DAG cards[4]), network processors[5,6,7] or commercial solutions provided by router vendors[8], but they can be obtained at significantly lower prices, thus providing cost-aware solutions. Moreover, as the monitoring functionality is developed at user-level, commodity hardware-based solutions are

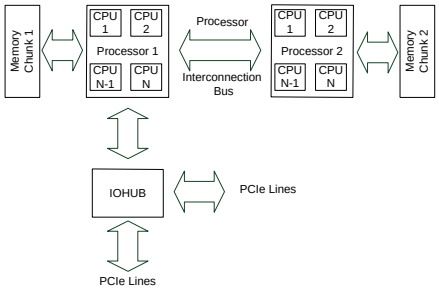
[1] `linux.die.net/man/8/numactl`
[2] `linux.die.net/man/8/lspci`
[3] `www.netfpga.org`
[4] `www.endace.com/`
[5] `www.alcatel-lucent.com/fp3/`
[6] `www.lsi.com/products/networkingcomponents/Pages/networkprocessors.aspx`
[7] `www.intel.com/p/en_US/embedded/hwsw/hardware/ixp-4xx`
[8] `www.cisco.com/go/nam`



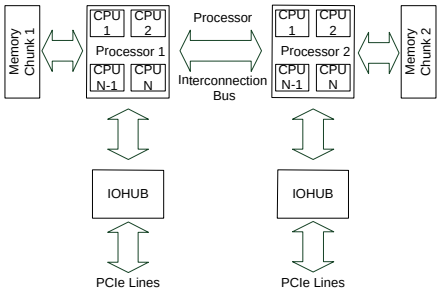
(a) PCIe lines connected to one processor



(b) PCIe lines connected to two processors



(c) PCIe lines connected to one IOHUB



(d) PCIe lines connected to two IOHUB

Fig. 2: NUMA architectures

largely flexible, which in addition to the mechanical compatibility, allows designing scalable and extensible systems that are of paramount importance for the monitoring of large-scale networks.

2.2 Operating System Network Stack

Nowadays, network hardware is rapidly evolving for high-speed packet capturing but software is not following this trend. In fact, most commonly used operating systems provide a general network stack that prioritizes compatibility rather than performance. Modern operating systems feature a complete network stack that is in charge of providing a simple socket user-level interface for sending/receiving data and handling a wide variety of protocols and hardware. However, this interface does not perform optimally when trying to capture traffic at high speed.

Specifically, Linux network stack in kernels previous to 2.6 followed an interrupt-driven basis. Let us explain its behavior: each time a new packet arrives into the corresponding NIC, this packet is attached to a descriptor in a NIC's receiving (RX) queue. Such queues are typically circular and are referred as rings. This packet descriptor contains information regarding the memory region address where the incoming packet will be copied via a Direct Memory Access (DMA) transfer. When it comes to packet transmission, the DMA transfers are

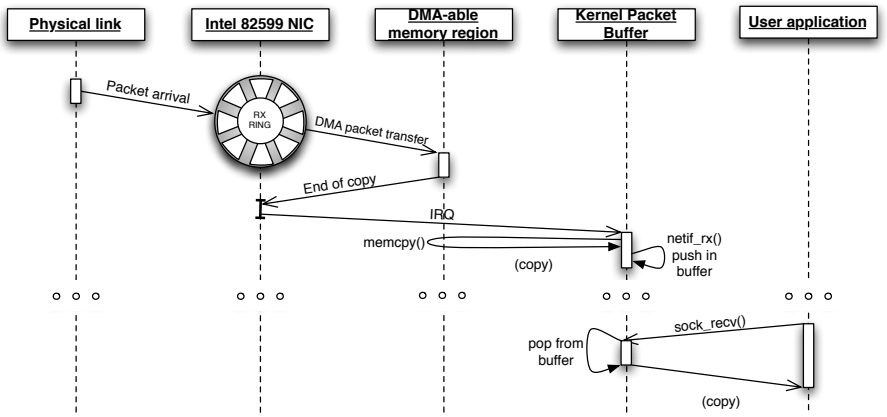


Fig. 3: Linux Network Stack RX scheme in kernels previous to 2.6

made in the opposite direction and the interrupt line is raised once such transfer has been completed so new packets can be transmitted. This mechanism is shared by all the different packet I/O existing solutions using commodity hardware. The way in which the traditional Linux network stack works is shown in Fig. 3. Each time a packet RX interrupt is raised, the corresponding interrupt software routine is launched and copies the packet from the memory area in which the DMA transfer left the packet, DMA-able memory region, into a local kernel `sk_buff` structure—typically, referred as packet kernel buffer. Once that copy is made, the corresponding packet descriptor is released (then the NIC can use it to receive new packets) and the `sk_buff` structure with the just received packet data is pushed into the system network stack so that user applications can feed from it. The key point in such packet I/O scheme is the need to raise an interrupt every time a packet is received or transferred, thus overloading the host system when the network load is high [15].

With the aim of overcoming such problem, most current high-speed network drivers make use of the NAPI (New API)[9] approach. This feature was incorporated in kernel 2.6 to improve packet processing on high-speed environments. NAPI contributes to packet capture speedup following two principles:

(i) *Interrupt mitigation.* Receiving traffic at high speed using the traditional scheme generates numerous interrupts per second. Handling these interrupts might lead to a processor overload and therefore performance degradation. To deal with this problem, when a packet RX/TX interrupt arrives, the NAPI-aware driver interrupt routine is launched but, differently from the traditional approach, instead of directly copying and queuing the packet the interrupt routine schedules the execution of a `poll()` function,

[9] `www.linuxfoundation.org/collaborate/workgroups/networking/napi`

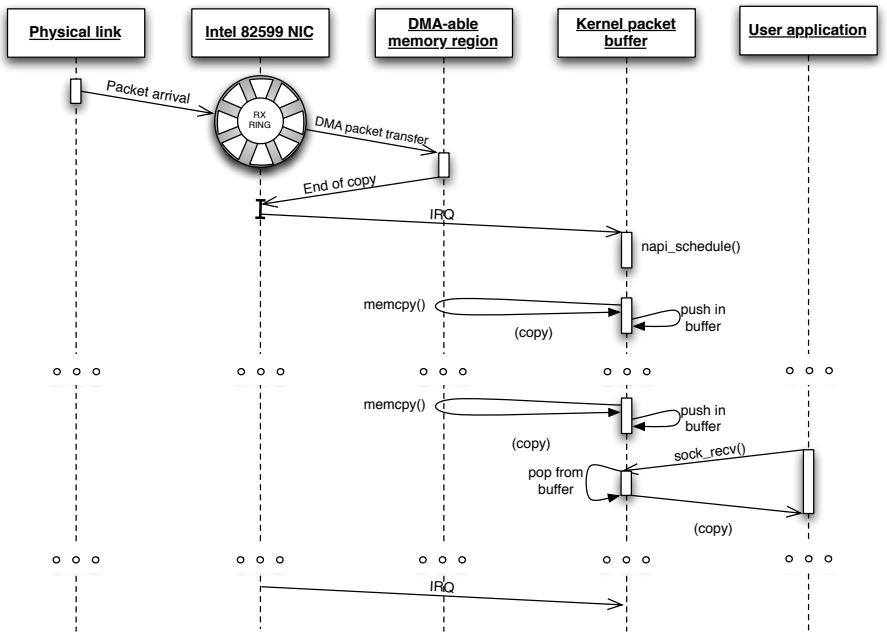


Fig. 4: Linux NAPI RX scheme

and disables future similar interrupts. Such function will check if there are any packets available, and copies and enqueues them into the network stack if ready, without waiting to an interruption. After that, the same `poll()` function will reschedule itself to be executed in a short future (that is, without waiting to an interruption) until no more packets are available. If such condition is met, the corresponding packet interrupt is activated again. Polling mode is more CPU consumer than interrupt-driven when the network load is low, but its efficiency increases as speed grows. NAPI compliant drivers adapt themselves to the network load to increase performance on each situation dynamically. Such behavior is represented in Fig. 4.

(ii) *Packet throttling.* Whenever high-speed traffic overwhelms the system capacity, packets must be dropped. Previous non-NAPI drivers dropped these packets in kernel-level, wasting efforts in communication and copies between drivers and kernel. NAPI compliant drivers can drop traffic in the network adapter by means of flow-control mechanisms, avoiding unnecessary work.

From now on, the GNU Linux NAPI mechanism will be used as the leading example to illustrate the performance problems and limitations as it is a widely used open-source operating system which makes performance analysis easier and

code instrumentation possible for timing statistics gathering. Furthermore, the majority of the existing proposals in the literature are tailored to different flavors of the GNU Linux distribution. Some of these proposals have additionally paid attention to other operating systems, for example, FreeBSD [8], but none of them have ignored GNU Linux.

2.3 Packet Capturing Limitations: Wasting the Potential Performance

NAPI technique by itself is not enough to overcome the challenging task of very high-speed traffic capturing since other architectural inherent problems degrades the performance. After extensive code analysis and performance tests, several main problems have been identified [4, 8, 16, 17]:

(i) *Per-packet allocation and deallocation of resources.* Every time a packet arrives to a NIC, a packet descriptor is allocated to store packet's information and header. Whenever the packet has been delivered to user-level, its descriptor is released. This process of allocation and deallocation generates a significant overhead in terms of time especially when receiving at high packet rates—as high as 14.88 Million packets per second (Mpps) in 10 GbE. Additionally, the `sk_buff` data structure is large because it comprises information from many protocols in several layers, when the most of such information is not necessary for numerous networking tasks. As shown in [16], `sk_buff` conversion and allocation consume near 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 64B sized packet [4].

(ii) *Serialized access to traffic.* Modern NICs include multiple HW RSS queues that can distribute the traffic using a hardware-based hash function applied to the packet 5-tuple (Section 2.1). Using this technology, the capture process may be parallelized since each RSS queue can be mapped to a specific core, and as a result the corresponding NAPI thread, which is core-bound, gathers the packets. At this point all the capture process has been parallelized. The problem comes at the upper layers, as the GNU Linux network stack merges all packets at a single point at network and transport layers for their analysis. Fig. 5 shows the architecture of the standard GNU Linux network stack. Therefore, there are two problems caused by this fact that degrade the system's performance: first, all traffic is merged in a single point, creating a bottleneck; second, a user process is not able to receive traffic from a single RSS queue. Thus, we cannot make the most of parallel capabilities of modern NICs delivered to a specific queue associated with a socket descriptor. This process of serialization when distributing traffic at user-level degrades the system's performance, since the obtained speedup at driver-level is lost. Additionally, merging traffic from different queues may entail packet disordering [18].

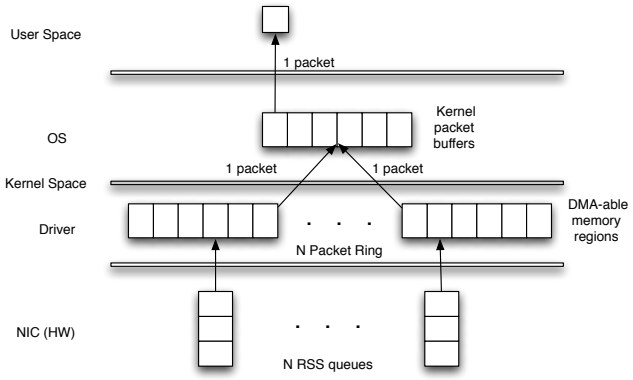


Fig. 5: Standard Linux Network Stack

(iii) *Multiple data copies from driver to user-level.* Since packets are transferred by a DMA transaction until they are received from an application in user-level, packets are copied several times, at least twice: from the DMA-able memory region in the driver to a packet buffer in kernel-level, and from the kernel packet buffer to the application in user-level. For instance, a single data copy consumes between 500 and 2000 cycles depending on the packet length [16]. Another important idea related to data copy is the fact that copying data packet-by-packet is not efficient, so much the worse when packets are small. This is caused by the constant overhead inserted on each copy operation, giving advantage to large data copies.

(iv) *Kernel-to-userspace context switching.* From the monitoring application in user-level is needed to perform a system call for each packet reception. Each system call entails a context switch, from user-level to kernel-level and vice versa, and the consequent CPU time consumption. Such system calls and context switches may consume up to 1000 CPU cycles per-packet [16].

(v) *No exploitation of memory locality.* The first access to a DMA-able memory region entails cache misses because DMA transactions invalidate cache lines. Such cache misses represent 13.8% out of the total CPU cycles consumed in the reception of a single 64B packet [4]. Additionally, as previously explained, in a NUMA-based system the latency of a memory access depends on the memory node accessed. Thus, an inefficient memory location may entail a performance degradation due to cache misses and greater memory access latencies.

3 Proposed Techniques 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. How-

ever, both the networking stack of current operating systems and applications at user-level do not properly exploit these new features. In this section, we present several proposed techniques to overcome the previous described limitations in the default operating systems' networking stack.

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

(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 packet reception. Particularly, N rings of descriptors (one per HW queue and device) are allocated when the network driver is loaded. Note that some extra time is needed at driver loading time but per-packet allocation overhead is substantially reduced. Likewise, when a packet has been transferred to user-space, its corresponding packet descriptor is not released to the system, but it is re-used to store new incoming packets. Thanks to this strategy, the bottleneck produced by per-packet allocation/deallocation is removed. Additionally, `sk_buff` data structures may be simplified reducing memory requirements. These techniques must be applied at driver-level.

(ii) *Parallel direct paths.* To solve serialization in the access to traffic, direct parallel paths between RSS queues and applications are required. This method, shown in Fig. 6, achieves the best performance when a specific core is assigned both for taking packets from RSS queues and forwarding them to the user-level. This architecture also increases the scalability, because new parallel paths may be created on driver module insertion as the number of cores and RSS queues grow. In order to obtain parallel direct paths, we have to modify the data exchange mechanism between kernel-level and user-level.

In the downside, such technique entails mainly three drawbacks. First, it requires the use of several cores for capturing purposes, cores that otherwise may be used for other tasks. Second, packets may arrive potentially out-of-order at user-level which may affect some kind of applications [18]. Third, RSS distributes traffic to each receive queue by means of a hash function. When there is no interaction between packets, they can be analyzed independently, which allows to take the most of the parallelism by creating and linking one or several instances of a process to each capture core. However, some networking tasks require analyzing related packet, flows or sessions. For example, a Voice over IP (VoIP) monitoring system, assuming that such a system is based on the SIP protocol, requires not only to monitor 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 by itself some aggregation task. The idea is that before packets

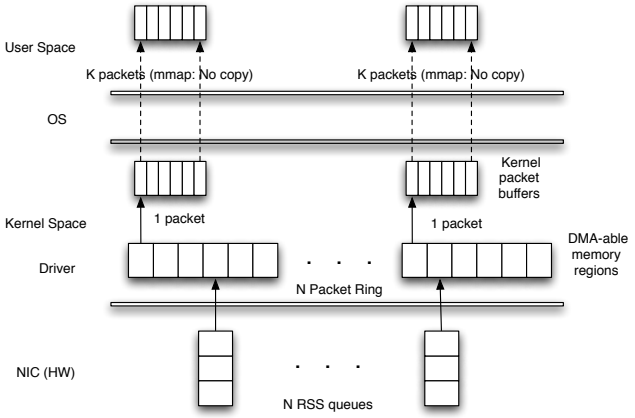


Fig. 6: Optimized Linux Network Stack

are forwarded to user-space (for example to a socket queue), a block of the capture system aggregates the traffic according to a given metric. However, this for sure is at the expense of performance.

(iii) *Memory mapping.* Using this method, a standard application can map kernel memory regions, reading and writing them without intermediate copies. In this manner, we may map the DMA-able memory region where the NIC directly accesses. In such case, this technique is called zero-copy. As an inconvenient, exposing NIC rings and registers may entail risks for the stability of the system [8]. However, this is considered a minor issue as typically the provided APIs protect NIC from incorrect access. In fact, all video boards use equivalent memory mapping techniques without major concerns. Another alternative is mapping the kernel packet memory region where driver copies packets from RX rings, to user-level, thus user applications access to packets without this additional copy. Such alternative removes one out of two copies in the default network stack. This technique is implemented on current GNU Linux as a standard raw socket with RX_RING/TX_RING socket option. Applying this method requires either driver-level or kernel-level modifications and in the data exchange mechanism between kernel-level and user-level.

(iv) *Batch processing.* To gain performance and avoid the degradation related with per-packet copies, batch packet processing may be applied. This solution groups packets into a buffer and copies them to kernel/user memory in groups called batches. Applying this technique permits to reduce the number of system calls and the consequent context switchings, and mitigates the number of copies. Thus, the overhead of processing and copying packets individually is removed. According to NAPI architecture, there are intuitively two points to use batches, first if packets are being asked in a

polling policy, the engines may ask for more than one packet per request. Alternatively, if the packet fetcher works on a interrupt-driven basis, one intermediate buffer may serve to collect traffic until applications ask for it. The major problem of batching techniques is the increase of latency and jitter, and timestamp inaccuracy on received packets because packets have to wait until a batch is full or a timer expires [19]. In order to implement batch processing, we must modify the data exchange between kernel-level and user-level.

(v) *Affinity and prefetching.* To increase performance and exploit memory locality, a process must allocate memory in a chunk assigned to the processor in which it is executing. This technique is called memory affinity. Other software considerations are CPU and interrupt affinities. CPU affinity is a technique that allows fixing the execution localization in terms of processors and cores of a given process (process affinity) or thread (thread affinity). The former action may be performed using Linux `taskset`[10] utility, and the latter by means of `pthread_setaffinity_np`[11] function of the POSIX pthread library. At kernel and driver levels, software and hardware interrupts can be handled by specific cores or processors using this same approach. This is known as interrupt affinity and may be accomplished writing a binary mask to `/proc/irq/IRQ#/smp_affinity`. The importance of setting capture threads and interrupts to the same core lies in the exploitation of cache data and load distribution across cores. Whenever a thread wants to access to the received data, it is more likely to find them in a local cache if previously these data have been received by an interrupt handler assigned to the same core. This feature in combination with the previously commented memory locality optimizes data reception, making the most of the available resources of a system.

Another affinity issue that must be taken into account is to map the capture threads to the NUMA node attached to the PCIe slot where the NIC has been plugged. To accomplish such task, the system information provided by the `sysctl` interface (shown in Section 2.1) may result useful. Additionally, in order to eliminate the 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 a near future in processor's cache in order to access them faster when required. Some drivers, such as Intel `ixgbe`, apply several prefetching strategies to improve performance. Thus, any capture engine making use of such vanilla driver, will see its performance benefited from the use of prefetching. Further studies such as [4, 20] have shown that more aggressive prefetching and caching strategies may boost network throughput performance.

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

4 Capture Engines implementations

In what follows, we present four capture engine proposals, namely: PF_RING DNA [10], PacketShader [4], Netmap [8] and PFQ [9], which have achieved significant performance. For each engine, we describe the system architecture (remarking differences with the other proposals), the above-mentioned techniques that applies, what API is provided for clients to develop applications, and what additional functionality it offers. Table 1 shows a summary of the comparison of the proposals under study. We do not include some capture engines, previously proposed in the literature, because they are obsolete or unable to be installed in current kernel versions (Routebricks [3], UIO-IXGBE [21]) or there is a new version of such proposals (PF_RING TNAPI [7]). Finally, we discuss the performance evaluation results, highlight the advantages and drawbacks of each capture engine and give guidelines to the research community in order to choose the more suitable capture system.

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

Characteristics/ Techniques	**PF_RING DNA**	**PacketShader**	**netmap**	**PFQ**
Memory Pre-allocation and re-use	✓	✓	✓	×/✓
Parallel direct paths	✓	✓	✓	✓
Memory mapping	✓	✓	✓	✓
Zero-copy	✓	×	×	×
Batch processing	×	✓	✓	✓
CPU and interrupt affinity	✓	✓	✓	✓
Memory affinity	✓	✓	×	✓
Aggressive Prefetching	×	✓	×	×
Level modifications	D,K, K-U	D, K-U	D,K, K-U	D (minimal), K,K-U
API	Libpcap-like	Custom	Standard libc	Socket-like

4.1 PF_RING DNA

PF_RING Direct NIC Access (DNA) 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, both RX and PF_RING queue allocations. PF_RING DNA also allows building parallel paths from hardware receive queues to user processes, that is, it allows to assign a CPU core to each received queue whose memory can be allocated observing NUMA nodes, permitting the exploitation of memory affinity techniques.

Differently from the other proposals, it implements full zero-copy, that is, PF_RING DNA maps user-space memory into the DMA-able memory region of the driver allowing users' applications to directly access to card registers and data in a DNA fashion. In such a way, it avoids the intermediation of the kernel packet buffer reducing the number of copies. However, as previously noted, this is at the expense of a slight weakness to errors from users' applications that occasionally do not follow the PF_RING DNA API (which explicitly does not allow incorrect memory accesses), which may potentially entail 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 some NAPI steps have been replaced by a zero-copy technique.

PF_RING DNA API provides a set of functions for opening devices to capture packets. It works as follows: first, the application must be registered with `pfring_set_application_name()` and before receiving packets, the reception socket can be configured with several functions, such as, `pfring_set_direction()`, `pfring_set_socket_mode()` or `pfring_set_poll_duration()`. Once the socket is configured, it is enabled for reception with `pfring_enable_ring()`. After the initialization process, each time a user wants to receive data `pfring_recv()` function is called. Finally, when the user finishes capturing traffic `pfring_shutdown()` and `pfring_close()` functions are called. This process is replicated for each receive queue.

As one of the major advantages of this solution, PF_RING API comes with a wrapping to the above-mentioned functions that provides large flexibility and ease of use, essentially following the de facto standard of the libpcap library. Additionally, the API provides functions for applying filtering rules (for example, BPF filters), network bridging, and IP reassembly. PF_RING DNA and a user library for packet processing are free-available for the research community[12].

4.2 PacketShader

The authors of PacketShader (PS) developed their own capture engine to highly optimize the traffic capture module as a first step in the process of developing a software router able to work at multi-10Gb/s rates. However, all their efforts are applicable to any generic task that involves capturing and processing packets. They apply memory pre-allocation and re-use, specifically, two memory regions are allocated—one for the packet data, and another for its metadata. Each buffer has fixed-size cells corresponding to one packet. The size for each cell of packet data is aligned to 2048 bytes, which corresponds to the next highest power of two for the standard Ethernet MTU. Metadata structures are compacted from 208

[12] `www.ntop.org/products/pf_ring/libzero-for-dna/`

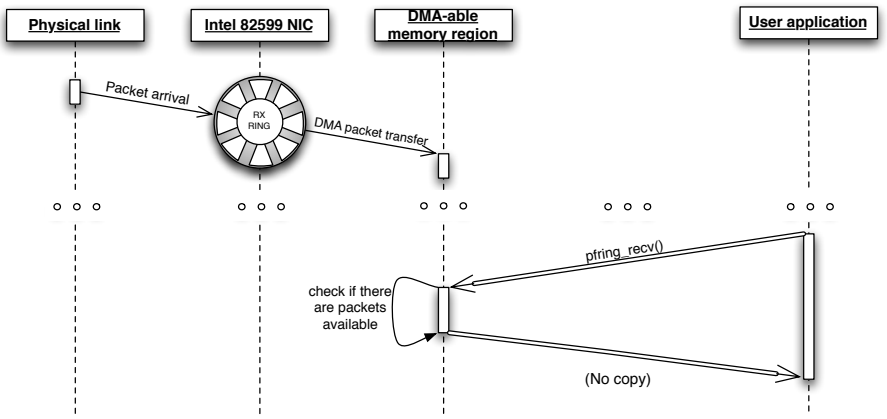


Fig. 7: PF_RING DNA RX scheme

bytes to only 8 bytes (96%) removing unnecessary fields for many networking tasks.

Additionally, PS implements memory mapping, thus allowing users to access to the local kernel packet buffers avoiding unnecessary copies. In this regard, the authors highlight the importance of NUMA-aware data placement in the performance of its engine. Similarly, it provides parallelism to packet processing at user-level, which balances CPU load and gives scalability in the number of cores and queues.

To reduce the per-packet processing overhead, batching techniques are utilized in user-level. For each batch request, the driver copies data from the huge packet buffer to a consecutive mapped memory region which is accessed from user-level. In order to eliminate the inherent cache misses, the modified device driver prefetches the next packet (both packet data and packet descriptor) while the current packet is being processed.

PS API works as follows: (i) user application opens a char device to communicate with the driver, `ps_init_handle()`, (ii) attaches to a given reception device (queue) with an `ioctl()`, `ps_attach_rx_device()`, and (iii) allocates and maps a memory region, between the kernel and user levels to exchange data with the driver, `ps_alloc_chunk()`. Then, when a user application requests for packets by means of an `ioctl()`, `ps_recv_chunk()`, PS driver copies a batch of them, if available, to the kernel packet buffer. PS interaction with users' applications during the reception process is summarized in Fig. 8.

PS I/O engine is available for the community[13]. Along with the modified Linux driver for Intel 82598/82599-based NICs network interface cards, a user library is released in order to ease the usage of the driver. The release also

[13] `shader.kaist.edu/packetshader/io_engine/index.html`

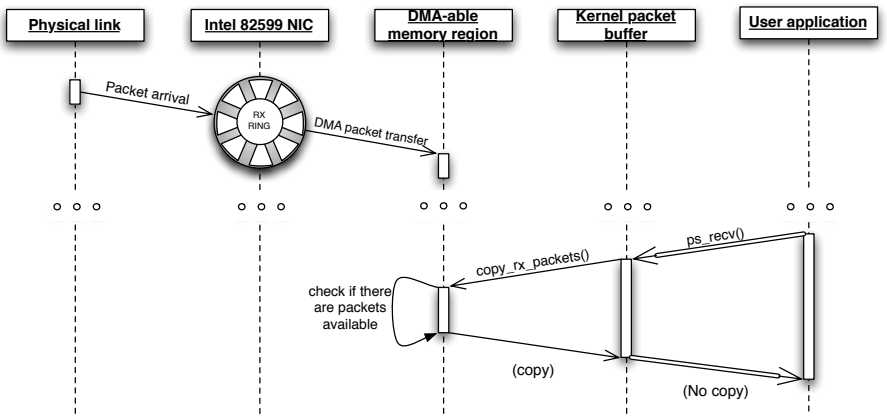


Fig. 8: PacketShader RX scheme

includes several sample applications, namely: a simplified version of `tcpdump`[14], an `echo` application which sends back all traffic received by one interface, and a packet generator which is able to generate UDP packets with different 5-tuple combinations at maximum speed.

4.3 netmap

Netmap proposal shares most of the characteristics of PacketShader's architecture. That is, it applies memory pre-allocation during the initialization phase, buffers of fixed sizes (also of 2048 bytes), batch processing and parallel direct paths. It also implements memory mapping techniques to allow users' application to access to kernel packet buffers (direct access to NIC is protected) with a simple and optimized metadata representation.

Such simple metadata is named netmap memory ring, and its structure 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 in the buffer, in reception and transmission buffers respectively (*avail*), flags about the status, the memory offset of the packet buffer, and the array of metadata information; it has also one slot per packet which includes the length of the packet, the index in the packet buffer and some flags. Note that there is one netmap ring for each RSS queue, reception and transmission, which allows implementing parallel direct paths.

Netmap API usage is intuitive: first, a user process opens a netmap device with an `ioctl()`. To receive packets, users ask the system the number of available packets with another `ioctl()`, and then, the lengths and payloads of the packets

[14] `www.tcpdump.org`

are available for reading in the slots of the netmap_ring. This reading mode is able to process multiple packets in each operation. Note that netmap supports blocking mode through standard system calls, such as `poll()` or `select()`, passing the corresponding netmap file descriptors. In addition to this, netmap comes with a library that maps libpcap functions into own netmap ones, which facilitates its operation. As a distinguish characteristic, Netmap works in an extensive set of hardware solutions: Intel 10 Gb/s adapters and several 1Gb/s adapters—Intel, RealTek and nVidia. Netmap presents other additional functionalities as, for example, packet forwarding.

Netmap framework is available for FreeBSD (HEAD, stable/9 and stable/8) and for Linux[15]. The current netmap version consists of 2000 lines for driver modifications and system calls, as well as a C header file of 200 lines to help developers to use netmap's framework from user applications.

4.4 PFQ

PFQ is a novel packet capture engine that allows packet sniffing in user applications with a tunable degree of parallelism. The approach of PFQ is different from the previous studies. Instead of carrying out major modifications to the driver in order to skip the interrupt scheme of NAPI or map DMA-able memory and kernel packet buffers to user-space, PFQ is a general architecture that allows using both modified and vanilla drivers.

PFQ follows NAPI to fetch packets but implements two novel modifications once packets arrive at the kernel packet buffer with respect to the standard networking stack. First, PFQ uses an additional buffer (referred as batching queue) in which packets are copied once the kernel packet buffer is full, those packets are copied in a single batch that reduces concurrency and increases memory locality. This modification may be classified both as a batching and memory affinity technique. As a second modification, PFQ makes the most of the parallel paths technique at kernel level, that is, all its functionalities execute in parallel and in a distributed fashion across the system's cores which has proven to minimize the overhead. In fact, PFQ is able to implement a new layer, named Packet Steering Block, in between user-level and batching queues, providing some interesting functionalities. Such layer distributes the traffic across different receive sockets (without limitation on the number of queues than can receive a given packet). These distribution tasks are carried out by means of memory mapping techniques to avoid additional copies between such sockets and the user level. The Packet Steering Block allows a capture thread to move a packet into several sockets, thus a socket may receive traffic from different capture threads. This functionality circumvents one of the drawbacks of using the parallel paths technique, that is, scenarios where packets of different flows or sessions must be analyzed by different applications—as explained in Section 3. Fig. 9 shows a temporal scheme of the process of requesting a packet in this engine.

[15] `info.iet.unipi.it/~luigi/netmap/`

It is worth remarking that, as stated before, PFQ obtains good performance with vanilla drivers, but using a patched driver with minimal modifications (a dozen lines of code) improves such performance. The driver change is to implement memory pre-allocation and re-use techniques.

PFQ is an open-source package which consists of a Linux kernel module and a user-level library written in C++[16]. PFQ API defines a `pfq` class which contains methods for 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 calling its `add_device()` method, (iii) timestamping must be enabled using `toggle_time_stamp()` method, and (iv) capturing must be enable using `enable()` method. After the initialization, each time a user wants to read a group of packets, the `read()` method is called. Using a custom C++ iterator provides by PFQ, the user can read each packet of the received group. When a user-level application finishes `pfq` object is destroyed by means of its defined C++ destructor. To get statistics about the received traffic `stats()` method can be called.

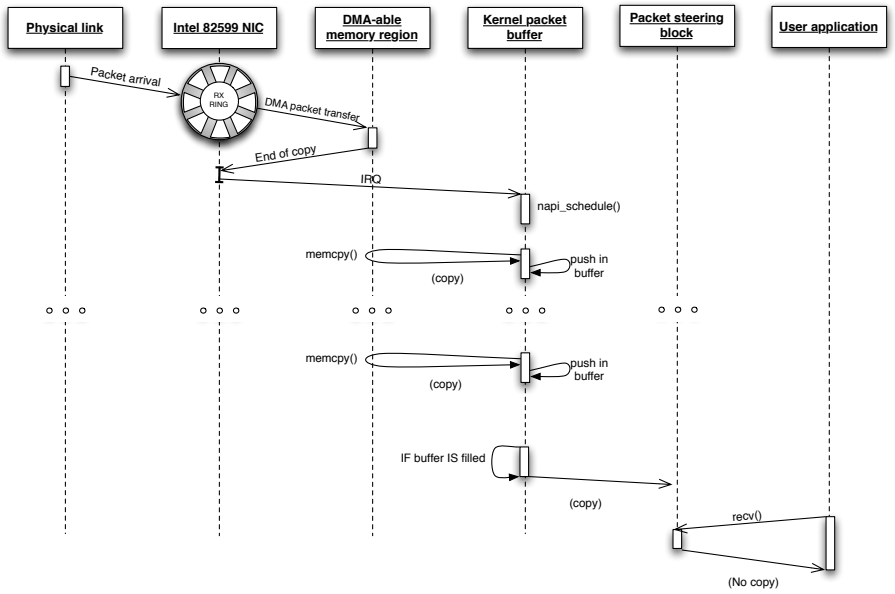


Fig. 9: PFQ RX scheme

[16] available under GPL license in `netserv.iet.unipi.it/software/pfq`

4.5 Capture Engines Comparison and Discussion

Once we have detailed the main characteristics of the most prominent capture engines in the literature, we turned our focus to their performance in terms of percentage of packets correctly received. It is noteworthy that the comparison between them, from a quantitative standpoint, is not an easy task for two reasons: first, the hardware used by the different studies is not equivalent (in terms of type and frequency of CPU, amount and frequency 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 measurement of the burden of CPU or memory.

For such reason, we have stress tested the engines described in the previous section, on the same architecture. Specifically, 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-based link. The receiver side is based on Intel Xeon with two processor of 6 cores each running at 2.30 GHz, with 96 GB of DDR3 RAM at 1333 MHz and equipped with a 10 GbE Intel NIC based on 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 similar scheme to that depicted in Fig. 2b. The NIC is connected to a slot corresponding to the first processor. The sender uses a HitechGlobal HTG-V5TXT-PCIe card which contains 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 sizes.

We have taken into account two factors, the number of available queues/cores and packet sizes, and their influence into the percentage of correctly received packets. We assume a link of 10 Gb/s full-saturated with constant packet sizes. For example, 60-byte packets in a 10 Gb/s full-saturated link gives a throughput in Mpps of 14.88: 10^{10} / ((60 + 4 (CRC) + 8 (Preamble) + 12 (Inter-Frame Gap)) · 8). Equivalently, if packet sizes grow to 64 bytes, the throughput in Mpps decreases to 14.22, and so forth.

It is worth remarking that netmap does not appear in our comparison because its Linux version does not allow changing the number of receive queues being this fixed at the number of cores. As our testbed machine has 12 cores, in this scenario netmap capture engine requires allocating memory over the kernel limits, and netmap does not start. However, we note that according to [8], its performance figures should be similar to those from PacketShader. Regarding PFQ, we evaluated its performance installing the aware driver.

Before showing and discussing the performance evaluation results, let us describe the commands and applications used to configure the driver and receive traffic, for each capture engine. In the case of PF_RING, we installed driver using the provided script `load_dna_driver.sh`, changing the number of receive queues with the RSS parameter in the insertion of the driver module. To receive traffic using multiple queues, we executed the following command: `pfcount_multichannel -i dna0 -a -e 1 -g 0:1:...:n`, where `-i` indicates

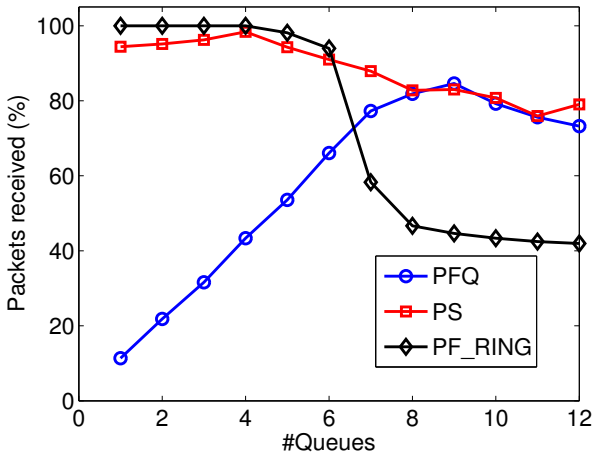


Fig. 10: Engines' performance for 60 (+4 CRC) byte packets

the device name, `-a` enables active waiting, `-e` sets reception mode and `-g` specifies the thread affinity for the different queues. Regarding PS, we installed the driver using the provided script `install.py` and receive packets using a slightly modified version of the provided application `echo`. Finally, in the case of PFQ, we install the driver using n reception queues, configure the receive interface, eth0, and set the IRQ affinity with the followings commands: `insmod ./ixgbe.ko RSS=n,n; ethtool -A eth0 autoneg off rx off tx off; bash ./set_irq_affinity.sh eth0`. To receive packets from eth0 using `n` queues with the right CPU affinity, we ran: `./pfq-n-counters eth0:0:0 eth0:1:1 ... eth0:n:n`. Note that in all cases, we have paid attention to NUMA affinity by executing the capture threads in the processor that the NIC is connected, as it has 6 cores, this is only possible when there are less than seven concurrent threads. In fact, ignoring NUMA affinity entails extremely significant performance cuts, specifically in the case of the smallest packet sizes, this may reduce performance by its half.

First, Fig. 10 aims at showing the worst case scenario of a full-saturated 10 Gb/s link with packets of constant size of 60 bytes (as in the following, excluding Ethernet CRC) for different number of queues (ranging from 1 to 12). Note that this represents a extremely demanding scenario, 14.88 Mpps, but probably not very realistic given that the average Internet packet size is clearly larger [22].

In this scenario, PacketShader is able to handle nearly the total throughput when the number of queues ranges between 1 and 4, being with this latter figure when the performance peaks. Such relatively counterintuitive behavior is shared by PF_RING DNA system, which shows its best permanence, a remarkable 100% packet received rate, with a few queues, whereas with when number of queues is larger than 7, the performance dips. Conversely to such behavior, PFQ increases

its performance according to the number of queues up to its maximum with nine queues, when such growth stalls. To further investigate such phenomenon, Fig. 11 depicts the results for different packets sizes (60, 64, 128, 256, 512, 1024, 1250 and 1500 bytes) and one, six and twelve queues.

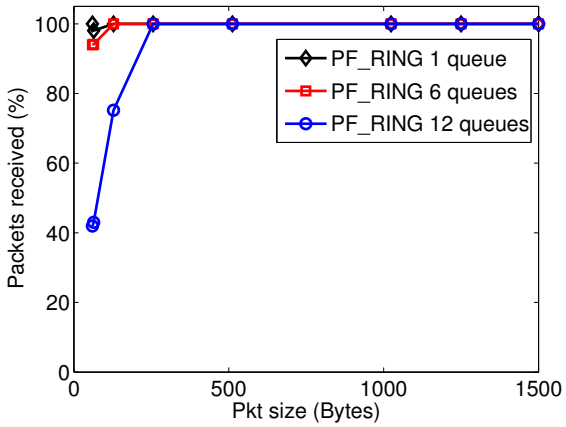
PF_RING DNA shows the best results with one and six queues. It does not show packet losses for all scenarios but those with packet sizes of 64 bytes and, even in this case, such figure is very low (about 4% with six queues and lower than 0.5% with one). Surprisingly, increasing packet sizes from 60 to 64 bytes, entails a degradation in the PF_RING DNA performance, although beyond these packe size, the performance recovers 100% rates. Note that larger packet sizes implies directly lower throughputs in Mpps. According to [8], investigation in this regard has shown that this behavior is because of the design of NICs and I/O bridges that make certain packet sizes to fit better with their architectures.

In a scenario in which one single user-level application is unable to handle all the received traffic, may result of interest to use more than one receive queue (with one user-level application per queue). In our testbed and assuming twelve queues, PacketShader has shown comparatively the best result, although, as PF_RING DNA, it performs better with a fewer number of queues. Specifically, for packet sizes of 128 bytes and larger ones, it achieves full packet received rates, regardless the number of queues. With the smallest packets sizes, it gives loss ratios of 20% in its worst case of twelve queues, 7% with six, and about 4% with one queue.

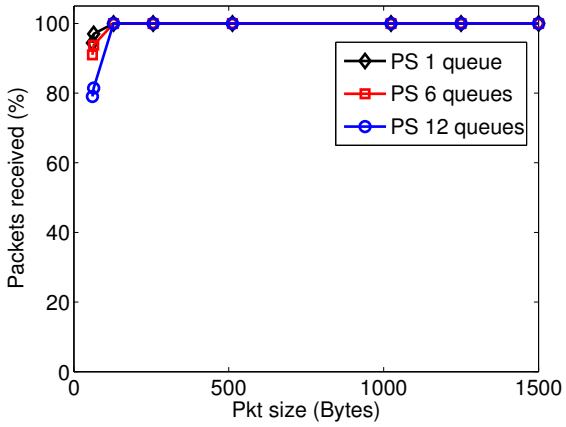
Analyzing PFQ's results, we note that such engine achieves also 100% received packet rates, but conversely to the other approaches, it works better with several queues. It requires at least six ones to achieve no losses with packets of 128 bytes or more, whereas with one queue, packets must be larger or equal to 1000 bytes to achieve full rates. This behavior was expected due to the importance of parallelism in the implementation of PFQ.

We find that these engines may cover different scenarios, even the more demanding ones. We state two types of them, whether we may assume the availability of multiple cores or not, and whether the traffic intensity (in Mpps) is extremely high or not (for example, packet size averages smaller than 128 bytes, which is not very common). That is, if the number of queues is not relevant, given that the capture machine has many available cores or no other process is executing but the capture process itself, and the intensity is relatively low (namely, some 8 Mpps), PFQ seems to be a suitable option. It comprises a socket-like API which is intuitive to use as well as other interesting functionalities, such as an intermediate layer to aggregate traffic, while it achieves full received packet rates for twelve queues. On the other hand, if traffic intensity is higher than the previous assumption, PacketShader presents a good compromise between the number of queues and performance.

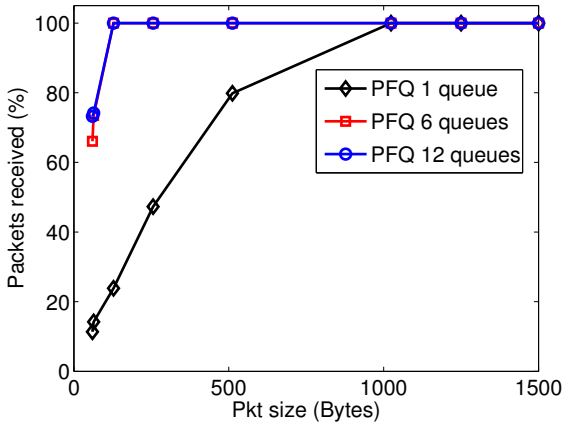
Nonetheless, often multi-queue scenarios are not adequate. For example, accurate timestamps may be necessary [19], packet disorder may be a significant inconvenient (according to the application running on the top of the engine) [18], or simply, it may be interesting to save cores for other tasks. In this scenario,



(a) PF_RING DNA



(b) PacketShader



(c) PFQ

Fig. 11: Performance in terms of percentage of received packet for different number of queues and constant packet sizes for a full-saturated 10 Gb/s link

PF_RING DNA is clearly the best option, as it shows (almost) full rates regardless packet sizes even with only one queue (thus, avoiding any drawbacks due to parallel paths).

5 Conclusions

The utilization of commodity hardware in high-performance tasks, previously reserved to specialized hardware, has raised great expectation in the Internet community, given the astonishing results that some approaches have attained at low cost. In this chapter we have first identified the limitations of the default networking stack and shown the proposed solutions to circumvent such limitations. In general, the keys to achieve high performance are efficient memory management, low-level hardware interaction and programming optimization. Unfortunately, this has transformed network monitoring into a non trivial process composed of a set of sub-tasks, each of which presents complicated configuration details. The adequate tuning of such configuration has proven of paramount importance given its strong impact on the overall performance. In this light, this chapter has carefully reviewed and highlighted such significant details, providing practitioners and researchers with a road-map to implement high-performance networking systems in commodity hardware. Additionally, we note that this effort of reviewing limitation and bottlenecks and their respective solutions may be also useful for other areas of research and not only for monitoring purposes or packet processing (for example, virtualization).

This chapter has also reviewed and compared successful implementations of packet capture engines. We have identified the solutions that each engine implements as well as their pros and cons. Specifically, we have found that each engine may be more adequate for a different scenario according to the required throughput and availability of processing cores in the system. As a conclusion, the performance results exhibited in this chapter, in addition to the inherent flexibility and low cost of the systems based on commodity hardware, make this solution a promising technology at the present.

Finally, we highlight that the analysis and development of software based on multi-core hardware is still an open issue. Problems such as the aggregation of related flows, accurate packet timestamping, and packet disordering will for sure receive more attention by the research community in the future.

References

1. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review **38**(2) (2008) 69–74
2. Braun, L., Didebulidze, A., Kammenhuber, N., Carle, G.: Comparing and improving current packet capturing solutions based on commodity hardware. In: Proceedings of ACM Internet Measurement Conference. (2010)

3. Dobrescu, M., Egi, N., Argyraki, K., Chun, B.G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., Ratnasamy, S.: Routebricks: exploiting parallelism to scale software routers. In: Proceedings of ACM SIGOPS Symposium on Operating Systems Principles. (2009)
4. Han, S., Jang, K., Park, K.S., Moon, S.: PacketShader: a GPU-accelerated software router. ACM SIGCOMM Computer Communication Review **40**(4) (2010) 195–206
5. Mogul, J., Ramakrishnan, K.K.: Eliminating receive livelock in an interrupt-driven kernel. ACM Transactions on Computer Systems **15**(3) (1997) 217–252
6. Kim, I., Moon, J., Yeom, H.Y.: Timer-based interrupt mitigation for high performance packet processing. In: Proceedings of the Conference on High Performance Computing in the Asia-Pacific Region. (2001)
7. Fusco, F., Deri, L.: High speed network traffic analysis with commodity multi-core systems. In: Proceedings of ACM Internet Measurement Conference. (2010)
8. Rizzo, L.: netmap: a novel framework for fast packet I/O. In: Proceedings of USENIX Annual Technical Conference. (2012)
9. Bonelli, N., Di Pietro, A., Giordano, S., Procissi, G.: On multi-gigabit packet capturing with multi-core commodity hardware. In: Proceedings of Passive and Active Measurement Conference. (2012)
10. Rizzo, L., Deri, L., Cardigliano, A.: 10 Gbit/s line rate packet processing using commodity hardware: survey and new proposals. Online: `http://luca.ntop.org/10g.pdf` (2012)
11. Intel: 82599 10 Gbe controller datasheet. `http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html` (2012)
12. Microsoft: Receive Side Scaling. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff567236(v=vs.85).aspx`
13. Woo, S., Park, K.: Scalable TCP session monitoring with Symmetric Receive-Side Scaling. Technical report KAIST, `http://www.ndsl.kaist.edu/~shinae/papers/TR-symRSS.pdf` (2012)
14. Dobrescu, M., Argyraki, K., Ratnasamy, S.: Toward predictable performance in software packet-processing platforms. In: Proceedings of USENIX Symposium on Networked Systems Design and Implementation. (2012)
15. Zabala, L., Ferro, A., Pineda, A.: Modelling packet capturing in a traffic monitoring system based on Linux. In: Proceedings of Performance Evaluation of Computer and Telecommunication Systems. (2012)
16. Liao, G., Znu, X., Bnuyan, L.: A new server I/O architecture for high speed networks. Proceedings of Symposium on High-Performance Computer Architecture (2011)
17. Papadogiannakis, A., Vasiliadis, G., Antoniades, D., Polychronakis, M., Markatos, E.P.: Improving the performance of passive network monitoring applications with memory locality enhancements. Computer Communications **35**(1) (2012) 129–140
18. Wu, W., DeMar, P., Crawford, M.: Why can some advanced Ethernet NICs cause packet reordering? IEEE Communications Letters **15**(2) (2011) 253–255
19. Moreno, V., Santiago del Río, P.M., Ramos, J., Garnica, J.J., García-Dorado, J.L.: Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines. IEEE Communications Letters **16**(11) (2012) 1888–1891
20. Su, W., Zhang, L., Tang, D., Gao, X.: 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)

21. Krasnyansky, M.: UIO-IXGBE. Online. https://opensource.qualcomm.com/wiki/UIO-IXGBE (2012)
22. CAIDA: Traffic analysis research. http://www.caida.org/research/traffic-analysis/