# 10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals

Luigi Rizzo, Luca Deri, Alfredo Cardigliano

## ABSTRACT

The network stack of operating systems has been designed for general purpose communications. Network drivers are responsible for bridging network adapters with kernel packet management facilities. While this approach is pretty flexible and general, it makes it unsuitable for high-speed network applications. This is because the journey of a packet between the network adapter and the target application is pretty long, as packet have to cross several kernel layers that in specific cases could be circumvented.

This paper presents a survey of techniques used to achieve fast packet capture and generation on commodity hardware. Then it shows in detail how two solutions developed by the authors can be used effectively to improve some previous proposals, not only reaching line rate but also exploiting advanced features of the NIC and limiting the number of cpu cycles spent for moving packets between the wire and the application.

The validation on both FreeBSD and Linux operating systems has confirmed that legacy applications can significantly improve their performance when using our solutions, and new applications can benefit from the exposure of hardware features usually hidden by standard APIs such as pcap.

## 1. INTRODUCTION

In the evolution of computer systems there has always been a race between the speed of communication links, and the ability of systems to cope with the maximum packet rates achievable by the link. The tension comes from the fact that processing costs have a constant per-packet component which becomes dominant on smaller packets. Taking as a reference a 10 Gbit/s link, the raw throughput is well below the memory bandwidth of modern systems (between 6 and 8 GBytes/s for CPU to memory, up to 5 GBytes/s on PCI-Express x16). However a 10Gbit/s link can generate up to 14.88 million Packets Per Second (pps), which means that the system must be able to process one packet every 67.2 ns. This translates to about 200 clock cycles even for the faster CPUs, and might be a challenge considering the per-packet overheads normally involved by general-purpose operating systems. The use of large frames reduces the pps rate by a factor of 20..50, which is great on end hosts only concerned in bulk data transfer. Monitoring systems and traffic generators, however, must be able to deal with worst case conditions. In particular they should be able to both handle efficiently minimum packet size and absorbe traffic spikes that can occur at any time on the network through smart buffers. With the introduction of multi-core systems, applications can efficiently process packets if they are able to properly exploit such architectures. Since packet capture is the core activity of network monitoring applications, partitioning them into multiple concurrent execution threads might not be enough. In fact the standard mechanisms provided by general-purpose operating systems are unsuitable for granting good performance in packet capture. Non-Uniform Memory Access (NUMA) is a multi-processor computer design which is now widely used in symmetric multiprocessor (SMP) architectures. In NUMA design, each processor is directly connected with separate memory, thus avoiding performance hit when several processors attempt to access the same memory. This means that threads active on a CPU can efficiently access memory allocated on the directly connected memory, whereas accessing memory on a non-local CPU is significantly slower as this happens by means of the Quick-Path Interconnect (QPI) bus. The result is that when applications are partitioned into several execution threads, it is important to preserve memory locality thus access via QPI bus is minimized. This is even more important for network activities, as whenever a packet is received some memory (skb in Linux, and mbuf on FreeBSD) needs to be allocated. The consequence is that packets should be allocated and consumed on the same physical processor where they have been received, as otherwise the per-packet QPI penalty will significantly reduce the overall application performance if packet memory is allocated, accessed, and released on different physical processors.

In order to expoit multi-core architectures, network cards manufacturers have changed the design of network adapters by logically partitioning them into several independent RX/TX queues. Usually the number

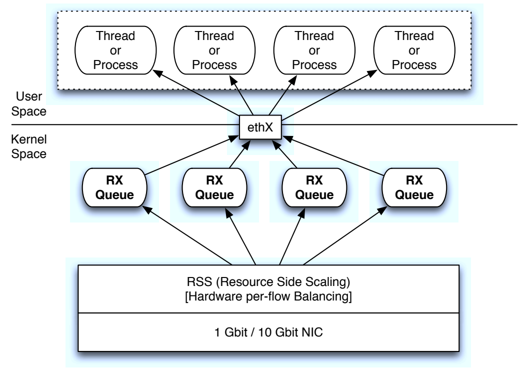

**Figure 1: Multi-Queue Network Adapters**

of queues is limited to the number of available processor cores; thus a 8 core system can access up to 8 RX/TX queues. Packets are distributed across queues by means of Receive-Side Scaling (RSS) [4] that is implemented in hardware by network adapters. Incoming packets are decoded and a hash value is computed on specific header fields such as IP address, protocol and port. This value is used to decide on which queue a specific packet will be assigned. RSS uses a one-way hash that does not guarantee that all packets belonging to the same communication flow (e.g. a TCP connection) will be assigned to the same RX queue. From the operating system point of view, it is now possible to simultaneously poll and send packets per queue thus maximizing the overall throughput. Unfortunately, these queues are not exposed to applications as the operating system presents multi-queue network adapters as legacy single-queue adapter. The result is that multi-threaded applications cannot poll/transmit packets by accessing directly the available queues, but they still need to serialize the operations as all threads need to access the same ethernet device.

Due to all these changes in computer architectures, the only way to achieve good performance on multi-core systems is to redesign the application. As stated before, wisely partitioning applications into multiple threads is a prerequisite for spreading the load across cores, but this might not be sufficient unless threads can independently receive and transmit packets. On the other hand, due to the one-way RSS hash, applications that need to process bi-directional traffic (also known as flow-based applications) cannot be bound to a specific queue, as the two traffic directions of the same connection will be sent to different queues. In a nutshell, it is possible to maximize performance of packet-based applications just spawning one application per RX queue, as they process each packet independently. Whereas flow-based applications cannot benefit from RSS traffic balancing as they need to maintain the flow state observing both flow directions and process packets according to it.

| Packet-based Applications | Flow-based Applications |
|---|---|
| Network Bridges and Firewalls | Intrusion Detection Systems |
| NetFlow and sFlow Probes | Intrusion Protection Systems |
| VoIP (Voice over IP) Probes | Network Performance Analyzer |
| Packet-to-Disk Applications | HTTP(S) Traffic Analyzers |
| | Network Latency Monitors |

**Table 1: Packet and Flow-based Applications**

As the price per port is below 500$, 10 Gbit network adapters are not becoming very common in data centers. Considered all the recent advances in computer and network architectures, we decided to analyze whether available packet processing frameworks can enable applications to cope with high-speed traffic analysis. In this paper we do not want to focus on specific applications, nor consider per-packet processing time as this changes significantly according to the application. The goal is to analyze how the various frameworks handle network packets, whether they can exploit modern network adapters, and also if they offer an application programming interface (API) for simplifying the developments of packet-processing applications.

In the first part of this paper, we present a survey of line rate packet capture and generation techniques on high speed (1 to 10 Gbit/s) links. In the second part, we focus on a couple of systems developed by the authors, showing when they can be useful, and how they improve the state of the art, in the context of packet capture and generation.

In more detail, the paper is structured as follows. In Section 2 we briefly describe how network cards are managed in operating systems, the basic techniques used to implement packet capture and generation, and the performance issues that they have. Section 3 presents solutions based on dedicated hardware, traditionally used to provide reliable performance for dedicated capture and generation systems. Software based solutions are presented in Section 4. Section 5 covers the design and implementation of two packet processing frameworks developed by the authors.

## 2. BACKGROUND

### 2.1 Limitations in Operating System Design

The network support on operating systems has been designed for general purpose networking. While this solution is flexible enough to accommodate all the various networking needs, it is suboptimal for high-speed network processing. Both network drivers and the network stack have been designed to be generic allowing new protocols to be accommodated by means of kernel modules. The drawback is that the packet journey from the network adapter to the user-space application is pretty long. Packets have to traverse several layers including the firewall and traffic shaper, that increase latency and limit the overall performance as they add per-packet processing overhead.

In a similar fashion, network drivers have also been designed for general purpose networking. Operating systems allocate memory for storing incoming packets and queueing them into the networking stack. This memory is then freed as soon as the packet has been processed. Although memory allocation for small packet buffers is a relatively fast operation, it has a cost and operating systems try to minimize it by applying various memory recycling techniques [14, 18]. Avoding at all memory allocation is not possible as long as packets need to be queued in kernel datastructures and nor processed immeditely as they leave the driver. Please note that in order to preserve a good performance, NUMA systems must free the packet memory on the same core on which the original memory was allocated.

Another design principle of the network stack, is that packets can be consumed by multiple applications. This happens for instance in case of multicast communications or packet capture. In order to reduce memory usage, operating systems do not duplicate packets but use references to the original packet. The drawback is that this practice delays memory release until the slowest packet consumer has processed the packet, and also might cause out-of-memory faults in case of traffic spikes when the memory allocator needs fresh memory for incoming packets, while the actual memory cannot be released as still in use by packet consumers.

Network device drivers are responsible for both receiving and transmitting packets, interfacing network adapters with the kernel. Incoming packets are copied into memory buffers that are then queued into kernel datastructures, outgoing packets are placed into network adapters memory until they are transmitted on the wire. Ideally packets directed to user-space applications should be delivered directly, using the shortest possible path. In practice these packets are first copied in kernel and then passed to user-space. In order to reduce the packet journey, several packet capture frameworks [5, 2] have adopted memory-map techniques for reducing the cost of copying packets from kernel to user-space though system calls. Although the performance has been greatly improved with respect to solutions like Linux PF_PACKET, many CPU cycles are wasted as incoming packets are copied from the driver to the kernel, and then to specific memory datastructures mapped to user-space. These solutions address just the problem of passing efficiently packets from kernel to user-space but do not tackle other issues including packet memory allocation/deallocation.

## 2.2 Legacy Device Drivers

Network device drivers allocate two circular buffers, one for RX and one for TX, on which packets are copied. Whenever a packet is received, the network adapter copies the packet in DMA inside the first available slot of the RX ring and notifies the driver by means of an interrupt. When packets need to be transmitted, the network driver copied the packet inside the first available slot of the TX ring and it initiates the transmission by usually updating a card register. In order to minimize memory allocations, the circular buffers memory slots point to packet memory buffers; adding/removing packets from the ring can be achieved by setting the circular buffer slot pointer to the address of the packet memory buffer.

In modern operating systems, network device drivers usually rely on a flexible packet representations (`mbuf, skbuf, NdisPacket`) to support a variety of requests from the OS: arbitrary data fragmentation, buffer sharing, offloading of tasks to the network adapter. This approach however causes large per-packet overheads, impacting performance in many ways, including energy efficiency and the ability to work at wire speed (up to 14.8 Mpps on 10 Gbit/s interfaces).

Dedicated appliances avoid this overhead by taking direct control of the hardware, or removing all unnecessary software layers. Following a similar approach, researchers have used modified Click drivers [11, 8], or export packet buffers to user space [9], to reach processing speeds of millions of packets per second per core. A limitation of this approach is that the application (even though through libraries) must take full control of the hardware, which makes the system extremely vulnerable in case of a crash of the application itself, and often prevents the use of some convenient OS system primitives (e.g. select/poll).

## 2.3 Improving Packet Processing

As described in the previous sections, the design of operating systems is limiting the packet processing performance. Tests performed in our lab have shown that most state of the art packet processing frameworks, as those listed on 4, can very seldom process more that 50% of the capacity of a 10 Gbit/s ethernet link when using minimal size packets. Even if some packet processing applications such as NetFlow/IPFIX probes can operate with sampled packets, beside the fact that packet sampling might reduce measurement accuracy, on most packet capture frameworks its usage does not reduce the load on the system. This is because discarding sampled packets is still a costly activity as device drivers capture and copy packets on memory buffers prior to discard them, and free the memory used to store the packet. All these facts demonstrate that in order to improve packet processing performance it is not enough to just address selected issues such as packet memory allocation, but the overall packet process infrastructure has to be redesigned. Furthermore, most packet process frameworks limit their scope to just packet capture without offering specific packet transmission facilities, preventing them from being used in systems that require packet transmission, such as IPSs and OpenFlow switching [16]. In fact, an area that is mostly unexplored, is the use of 10 Gbit commodity hardware for packet transmission. Besided rare exceptions [17] based on commercial hardware cards, the only available alternative are commercial traffic generators. Considered that modern computer systems are pretty fast, we believe that it is now time to see whether this niche market could be filled using commodity network adapters for generating traffic at 10 Gbit/s wire-rate.

## 3. HARDWARE SOLUTIONS

In 2001 Endace has created the market of FPGA (Field Programmable Gate Array)-based card introducing the DAG card [1, 10]. Recently an open-source FPGA-based adapter named NetFPGA [13] has entered this market. These cards leverage on FPGAs to reduce load on the CPU during packet capture and transmission. The result is that even at minimum size packets it is possible to capture and transmit traffic with a modest CPU load. Other advantages of these cards are the ability to precisely timestamp in hardware incoming packets, features that is available just on a few commodity network adapters such as Intel 82580-based NICs. These adapters receive and transmit packets in DMA (Direct Memory Accees) by reading/writing packets into the memory banks installed on the network adapter. Incoming packets can be tagged according to filtering rules (available only on selected card models), so that they can be balanced across virtual network queues, similar to what happens with multi-queue commodity network adapters. The ability to synchronize the card clock used for packet timestamping via a GPS makes these cards able to satisfy advanced packet processing needs.

## 4. SOFTWARE SOLUTIONS

Software based solutions become popular whenever the performance of CPUs and I/O buses matches or exceeds that of network hardware. As of this writing, we are in this situation, with 10 Gbit/s data rates becoming manageable in software with modern multicore CPUs and multiqueue devices. As a consequence, recent literature has presented a number of solutions that range from custom Click drivers [11] to memory mapped packet buffers [12, 9, 15] to PF_RING and DNA [5, 6].

### 4.1 Polling Click drivers

Click [11] is a modular software architecture developed long ago to build packet processing systems, and widely used in research and production systems. Click makes it easy to connect software "elements" to build a packet processing chain connecting sources (typically the receive side of a network card) to sinks (the transmit side of those cards). Some special Click elements implement custom device drivers which, through the use of Polling mode and a variety of performance optimizations, reduce the per-packet overhead.

### 4.2 Memory mapped packet buffers

A number of solutions try to provide direct access to the packet buffers managed by the network card. UIO-IXGBE [12] is a custom kernel driver and user-space library developed by Qualcomm for Intel 82598-based 10 Gbit adapters, that creates a transparent memory mapping layer. User-space applications can expoit its API to both send and receive packets at very high-speed in DMA based on an abstraction layer developed as part of the project. Unfortunately the driver is not just a patch of the Intel driver but it has been recoded, thus support of new 82599-based NICs is not present. Netslice [15] is another such solution that falls into this category.

## 5. TOWARDS 10 GBIT LINE RATE PACKET PROCESSING

In this section we present the design principles and performance evaluation of two proposals developed by the authors for achieving line rate at 10 Gbit/s speeds and more.

### 5.1 Performance metrics

A packet processing system usually exercises many system components, and it is important to identify individual contributions to the performance of the overall system. CPU cycles are one of the main resources that we account for, but of course depending on the system under test there might be other resources that are in short supply, such as memory or I/O bus bandwidth. Even just limiting at software considerations, we should clearly distinguish between two cost factors:

- system costs, which account for the resources consumed in bringing packets from the network to the application, and vice-versa. This category includes, as an example, interrupt processing and system call overhead (including data copies);
- application costs, which accounts for the specific processing done by applications. As an example, timestamping, classification, checksumming etc. all fall into application costs.

In many systems, one of the factors is dominating over the other, so mixing the two would hide a lot of information on the behaviour of the system under analysis. In this paper, the focus is on mechanisms to support fast access to network traffic, thus we are almost exclusively interested in determining the system costs. It follows that we will structure our tests in a way that makes application costs negligible or at least clearly defined.

In an orthogonal dimension, cost factors can be split into per-packet and per-byte components. The latter are usually simple to deal with, because they are proportional to the memory bus bandwidth of the system. Modern CPUs easily reach 4..10 Gbytes/s on the memory bus, meaning that the per-byte costs are in the microsecond range even for the largest packets. Only on slow CPUs or systems with slow memory or I/O buses the per-byte costs can become important. On the contrary, per-packet costs are highly variable depending on the system and the application.

From these considerations it follows that the correct way to evaluate the performance of packet processing systems is often to drive the system with the shortest possible packets and measure the throughput in packets per second.

With this in mind we present two solutions, one especially suited to packet generation, the other to packet capture and filtering, that improve the state of the art

## 6. NETMAP

**netmap** is a recent proposal that falls in the category of memory mapped buffer access. In common with other proposals of this class, **netmap** reduces the per-packet overhead through the use of memory mapped buffers and metadata. However, it goes a step further in terms of other features, such as device independence, safety of use, and integration with the operating system.

In **netmap**, a Network Interface Controller (NIC) can be dynamically switched between regular mode (where the NIC exchanges data packet with the host stack as usual) and **netmap** mode, where the operating system remains in control of the configuration of the NIC, but the data path is disconnected. Adapters in **netmap** mode can be controlled by user programs through a few data structures residing in a shared memory region, and ioctl() and select() (or poll()) system calls. The shared data structures contain three types of objects called *packet buffers*, *netmap rings*, and *netmap_if*. Packet buffers are fixed size memory blocks, non pageable and allocated by the kernel, where the NIC reads/writes packets from/to the network. Each buffer is identified by a unique index in the range $1 \ldots N_B$, with the upper value determined by the kernel depending on memory availability. A *netmap ring* is a shadow, device-independent version of the *hardware rings* normally used by the NIC to manage the transmit and receive queue. A netmap ring, also residing in the shared memory regions where buffers are, contains fields indicates the ring size, the current read (or write) position, the number of slots available for read (or write), some flags, and an array of *slots*, one for each buffer which is part of the

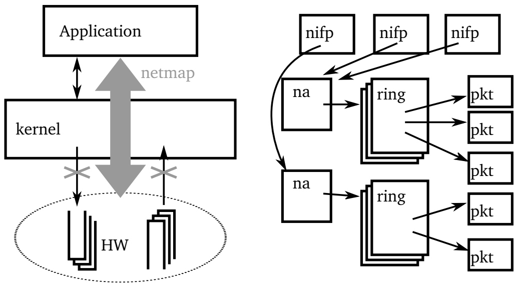

**Figure 2: Netmap Memory Layout.**

ring. The slot in turn contains the index of the buffer associated to the slot, the length of the packet stored in it, and some flags. The translation between the slot index and its address in the shared memory is done thanks to another field of the netmap ring, which indicates the offset (*buf_ofs*) between the address of the netmap ring (`ring_addr`) and the start of the buffer area in the shared memory region. With these information, the address of a buffer can be simply computed as
`ring_addr + buf_ofs + index * buf_size`
and the computation is *position independent*, i.e. does not depend on where, in the process address space, the shared memory is mapped. Finally, all shadow rings and other parameters of an interface are described by a *netmap_if*, which contains the number of rings and an array of offsets between the `netmap_if` and each of the rings. Once again, the use of offsets permits to reach the rings in a position-independent way. The switching of a NIC in **netmap** mode happens by opening the special device `/dev/netmap` and invoking the `NIOCREG` ioctl, passing the interface name as a parameter. On return the ioctl provides the size of the shared memory region where all data structures reside, and the offset in this area of the `netmap_if`. The memory area can be subsequently mapped into the process' address space through an `mmap()` call.

At this point, processes can issue non-blocking `ioctl()` calls to synchronize the state of the netmap rings with those of the hardware rings. In particular, on the transmit side, a process can fill the buffers (starting from the current write position, and up to the number of available ones) with the content of packets to send, and call the NIOCTXSYNC ioctl() to tell the kernel to actually schedule the transmission. The call validates the parameters (buffer indexes and lengths) in the netmap ring, updates the hardware rings accordingly, and possibly starts the NIC for the new transmissions. On return, the shadow ring will contain updated values for the current insert position and number of free slots, also taking into accounts previous transmissions that have completed. On the receive path, the application should first call the NIOCRXSYNC ioctl(), which checks whether there are newly received packets, copies the length into the shadow rings, and updates the state of the NIC, this time reusing any buffers that the application has declared as newly available. Besides being non blocking, none of this calls involves any data copying (with the exception of the length field), because the buffers are shared between the the netmap and the hardware rings. At the same time, a misbehaving application cannot cause corruptions in the system because the content of the rings is checked before being used to update the state of the NIC.

Blocking until I/O is possible is done through the `poll()` system call. In particular, each file descriptor returned by opening /dev/netmap is associated to one (or all) queue pair in the NIC – the association is controlled by one field in the *netmap_if.* This field also determines which queues are used by the NIOCTXSYNC and NIOCRXSYNC system calls. In the case of poll(), the file descriptor unblocks when the number of available descriptor becomes non zero. A blocking descriptor also causes the execution of the body of a TX/RX SYNC. This further reduces the system call overhead of a process using netmap: the eventloop requires just a single system call for each batch of packets to be sent or received on the various netmap-enable interfaces.

In addition to those associated to the adapter's queues, two more netmap rings per interface are used to communicate with the host stack. Packets coming from the host stack are made visible in an "RX" netmap ring, whereas packets written to the extra "TX" netmap ring are encapsulated and passed up to the host stack.

**netmap** is especially useful for applications (such as traffic generators, monitors, firewalls, etc.) that need to deal to traffic on a packet-by-packet basis. Such applications can set the interface in **netmap** mode, and use the netmap rings to read incoming traffic or send locally generated traffic. Passing packets through interfaces (or from/to the host stack) can be done by copying the buffers' contents from one ring to another. In many cases, zero-copy operation is possible by just swapping buffer pointers between rings, so that received buffers can be sent out on a different interface, while the other buffer is made available for incoming packets.

At least three factors contribute to the performance achieved by **netmap**: i) no overhead for encapsulation and metadata management; ii) no per-packet system calls and data copying (`ioctl()`s are still required, but involve no copying and their cost is amortized over a batch of packets); iii) much simpler device driver operation, because now all buffers have a plain and simple format that requires no run-time decisions.

The first prototype of **netmap**, developed on FreeBSD, consists of about 2000 lines of code for device functions (ioctl, select/poll) and driver support, and individual driver modifications (mostly mechanical, about 500 lines each) to interact with the netmap rings. The most tedious parts of the driver (initializing, the PHY interface, etc.) do not need changes. To date, **netmap** support is available for the Intel 10G and 1G, and for the RealTek 8169-based adapters. Support for other devices is coming.

**netmap** dramatically reduces the per-packet overheads compared to the ordinary host stack. One core achieves line rate (14.86Mpps) at just 1.3GHz, and even at 150 MHz can push out 1.76 Mpps. These numbers correspond to roughly 90 clock cycles/packet. The receive side gives similar numbers. Using 2 or 4 cores incurs a modest reduction of efficiency (going from 90 to 100-110 clocks/packet) probably due to memory and bus contention.

The ability to send or receive traffic bypassing the protocol stack is very interesting for certain applications. Finally, but not surprisingly, driver simplifications are so large that putting an interface in **netmap** mode and then using a userspace process to move packets between the host stack and the device resulted in a 15% increase compared to using the driver in standard mode.

## 7. PF_RING DNA

In 2003 PF_RING [5], a framework for accelerating packet capture and processing, has been released. PF_RING has been designed to be feature rich and not just accelerate packet capture. In particular it supports advanced packet filtering in addition to legacy BPF, in-kernel packet processing by means of loadable kernel plugins, load-balancing across multiple sockets, and ability to receive-modify-transmit selected packets by combining filtering rules with kernel plugins.

Contrary to similar solutions available at that time, PF_RING implements a memory-mapped memory buffer allocated at socket creation, on which incoming packets are copied. User-space applications can read packets simply accessing the mapped memory and incrementing the offset of the last read packet. This solution avoids system calls for reading packets and also uses a statically allocated memory without any per-packet memory allocation. In order to further enhance PF_RING performance, in 2005 a variant of PF_RING named PF_RING DNA (Direct NIC Access) [6] has been introduced. Similar to FPGA-based NICs, in DNA mode the per-socket PF_RING circular buffer, on which incoming packets are copied, has been replaced with the memory ring allocated by the device driver to host pointers to incoming packets. As the memory ring needs to be mapped to user-space, a modified network driver allocates it using continuous non-swappable memory. Similar to netmap, incoming packets are copied in the ring by the network adapter, and the user-space application that manages the buffer is responsible for reading packets and updating the index of the next slot that will host incoming packets. All the communications with the network adapter happen in DMA, as PF_RING maps in user-space both the packet memory ring and card registers. Contrary to netmap that does not allow user-space applications to manipulate card registers but rather update the card status by means of system calls, the DNA approach maximizes performance by avoiding un-necessary system calls even if misbehaved application could potentially set the registers to invalid values that migh prevent applications from receiving packets. This is considered a minor issue, as this practice does not result in system crashes or memory corruption, when compared to the benefit of completely bypassing the kernel during packet processing. The DNA implementation details are hidden to user-space applications as they read packets using the PF_RING API that is responsible for packet manipulation and card registers udpdate. Initially available for the Intel 1 Gbit/s adapters, recently we have added support for 10 Gbit/s adapters. As modern adapters feature multiple RX queues, the 10 Gbit version of DNA, creates a memory ring per queue, so that applications can read packets from individual rings. Intel adapters do not have really independent RX queues, thus the DNA network driver has to consume packets for those RX queues on which there is no active application. The drawback is that the operating system has to perform minimal packet housekeeping even if not application is receiving packets. Due to space constraints we have decided not to present detailed testign reports, as 10 Gbit/s DNA performance figures are similar to netmap and other memory-mapped solutions, being their design conceptually similar. Thus even with a low-end dual-core server it is possible to capture packets at 10 Gbit/s wire-speed using a limited numer of CPU cycles, or use a dual-port 10 Gbit card and thus capture about 20 Mpps. What makes DNA unique with respect to other available solutions is:

- The DNA driver is a pluggable solution that can be used seamlessly by legacy applications such as those based on pcap and PF_RING (non DNA).
- Most PF_RING features are available in DNA. The only difference is that DNA is much faster but with the limitation that an RX queue cannot be shared by multiple applications. Multiple threads living in the same application can instead concurrently access the queue as the user-space library is thread-safe.
- In order to both read packets and update the card registers for notifying that packets have been read, DNA uses memory mapping with no need of issue system call as required by netmap. The complete kernel bypass makes DNA virtually the fastest available solution, limiting its speed just on the network adapter being used.
- The memory buffers are allocated per queue, respecting the memory locality principle, and thus making it NUMA friendly.
- In DNA, when no applications are reading packets from queues, the kernel is not doing any housekeeping activity including buffers cleanup, and notification about packet being received. The outcome is that every CPU cycle is used wisely and not wasted in activities that could be potentially avoided. This includes interrupts processing that is disabled in DNA, and enabled only during `poll()` calls that are issued only when no packet is available for processing.
- As queues are completely independent and managed in user-space with no kernel housekeeping, applications sitting on top of them are fully independent can be mapped on different CPU cores with no cross-dependency. The outcome is that the system scales linearly with the number of cores, as each couple {queue, application} is independent from the others. Tests have validated this claim when both increasing the number of queues and ethernet ports.
- PF_RING DNA is not yet another packet processing accelerator, but a comprehensive framework that simplifies the design of networking applications.

## 8. OPEN ISSUES AND FUTURE WORK

Modern network adapters are becoming increasingly powerful both in terms of speed and features. Intel 82599-based and Neterion X3110 10 Gbit ethernet adapters feature advanced packet steering configuration that, based on hardware filtering rules, allow incoming packets to be diverted to specific RX queues. PF_RING already supports native Intel packet steering facilities [7], and its support is also planned for netmap.

Another area where we are focusing is on effective packet transmission. While TX support is under development for DNA, netmap has demonstrated that 10 Gbit/s wire-rate traffic generation is feasible using entry-level servers. Although open-source software traffic generators have been available since long time, they are focusing more on flexibility rather than speed. Recently the ostinato [17] project has released a flexible software-based traffic generator that can run on a large number of platforms including Linux and FreeBSD. We are planning to modify it so that it can run on top of DNA and netmap, in order to combine both the flexibility of ostinato with high packet generation rates.

As the use of virtual machines is becoming increasingly popular for reducing administrative and operational costs, we are planning to explore how DNA and netmap could be effectively used in virtualized environments. The vPF_RING (Virtual PF_RING) [3] project has demonstrated that the use of memory mapping inside virtual machines increases significatly the packet processing performance with respect to virtualized network adapters. We are currently evaluating how to add DNA support to vPF_RING in order to further reduce packet processing costs, while increasing packet capture rate.

Finally, netmap support of additional network cards and its porting to Linux is under development. In order to make netmap transparent to legacy applications, we are also coding some glue software for porting libpcap on top of netmap, similar to the libpcap PF_RING DNA support. The goal is to speed up legacy applications without the need to modify them.

## 9. FINAL REMARKS

This paper has presented a survey of previous proposals for fast packet capture and generation, followed by two novel systems proposed by the authors and targeted to smart packet capture (exploiting features of the hardware) and packet generation, respectively. Although memory-mapping techniques are not novel in the industry, both netmap and PF_RING DNA have demonstratated that 10 Gbit/s packet processing at wire-rate is feasible using commodity hardware. Our code, working on commodity hardware and operating systems, reaches 14.8Mpps with only a modest CPU usage, leaving most of the CPU cycles available for user operation.

## 10. ACKNOWLEDGEMENTS

We would like to thank Silicom Ltd. that has greatly supported and contributed to the development of 10 Gbit PF_RING DNA.

## 11. REFERENCES

[1] The dag project. Technical report, University of Waikato.

[2] A. Biswas. A high performance real-time packet capturing architecture for network management systems. Technical report, Masters Thesis, Concordia University, 2005.

[3] A. Cardigliano. Towards wire-speed network monitoring using virtual machines. Technical report, Masters Thesis, University of Pisa, 2011.

[4] Microsoft Corporation. Scalable networking: Eliminating the receive processing bottleneck - introducing rss. Technical report, Technical Report, 2004.

[5] L. Deri. Improving passive packet capture:beyond device polling. In *SANE 2004. Workshop on.*

[6] L. Deri. ncap: Wire-speed packet capture and transmission. In *End-to-End Monitoring Techniques and Services, 2005. Workshop on*, pages 47–55. IEEE, 2005.

[7] L. Deri, J. Gasparakis, P. Waskiewicz, and F. Fusco. Wire-speed hardware-assisted traffic filtering with mainstream network adapters. *Advances in Network-Embedded Management and Applications*, pages 71–86, 2011.

[8] 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 *ACM SOSP*, pages 15–28. Citeseer, 2009.

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

[10] A. Heyde and L. Stewart. Using the endace dag 3.7 gf card with freebsd 7.0. 2008.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[12] Max Krasnyansky. Uio-ixgbe. *https://opensource.qualcomm.com/wiki/UIO-IXGBE.*

[13] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. Netfpga–an open platform for gigabit-rate network switching and routing. *Microelectronics Systems Education, IEEE International Conference on/Multimedia Software Engineering, International Symposium on*, 0:160–161, 2007.

[14] E-Con InfoTech Pvt Ltd. Ethernet driver and optimization techniques, revision 2.0. Technical report, Technical Report, 2007.

[15] Tudor Marian. Operating systems abstractions for software packet processing in datacenters. *PhD Dissertation, Cornell University*, 2010.

[16] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[17] ostinato.org. Ostinato: Packet/traffic generator and analyzer. Technical report, 2011.

[18] C. Walravens and B. Gaidioz. Receive descriptor recycling for small packet high speed ethernet traffic. In *MELECON, 2006*, pages 1252–1256. IEEE, 2006.