

netmap: fast and safe access to network adapters for user programs

Luigi Rizzo*, Università di Pisa (31 may 2011)
<http://info.iet.unipi.it/~luigi/netmap/>

ABSTRACT

Many applications (software switches, traffic monitors and generators, firewalls, etc.) need to send and receive packets at line rate on very fast links. Running them on commodity operating systems has severe performance issues. One of the culprits is the mechanism normally used to access network device drivers, which has huge per-packet overheads, and is a poor match for the millions of packets per seconds traversing 1..10 Gbit/s links. A number of past proposals have suggested more efficient mechanisms, but all these solutions are either limited to a specific application (e.g. packet capture), or tied to a specific hardware device, or potentially unsafe for the operating system.

In this work we present **netmap**, a system that integrates the strengths of existing proposals and addresses their weaknesses. **netmap** uses memory mapped packet buffers, but enforces operating system checks on critical operations such as programming the hardware or passing pointers to the kernel, and is smoothly integrated with the synchronization system calls provided by the OS.

netmap has been implemented in FreeBSD for several 1 and 10 Gbit/s network adapters. In our prototype, it takes only 90 CPU clock cycles to move one packet between the wire and the application – about 10 times less than using the conventional APIs. With **netmap**, a single core running at 1.33 GHz achieves 14.88 Mpps (the peak packet rate on 10Gbit/s links). This level of efficiency is an enabling factor to the use of commodity operating systems for a wide range of high performance network applications.

1. INTRODUCTION

General purpose operating systems provide a rich and flexible environment for running all sorts of applications, including those related to network operation (routing, firewall, intrusion detection), monitoring and

testing. A peculiarity of these applications is the need to send or receive raw network packets without interference from the host stack. This is not the intended target of general purpose OS, which usually limit the access to raw packet access only to the kernel, to protect the system from untrusted user programs. The kernel itself has to access the network adapter (NIC) through an API (based on mbufs/skbufs/NdisPacket) which is extremely flexible, but heavyweight, as it requires every packet to be accompanied by a large amount of metadata, expensive to generate.

Over time, several mechanisms have been proposed to address the application requirements for direct access to the network. Raw sockets, the Berkeley Packet Filter (BPF), the AF_SOCKET family, and equivalent APIs are used to build all sorts of network monitors, traffic generators, and generic routing systems. Performance, however, is not adequate for millions of packets per second, which is what we can find on 1..10 Gbit/s links. The cost of crossing the kernel-userspace boundary is only part of the reason for this lack of performance. Experiments have shown that even remaining in the kernel does not permit reaching the peak PPS rates at 10 Gbit/s, because of the already mentioned metadata management issues.

In search of better performance, some systems expose, in various way, the network adapter's data structures to user space applications. We will provide a survey of these proposals in Section 3. Efficient as this may be, many of these systems have drawbacks: some are bound to one specific type of hardware; some bypass operating system protections, putting system stability at risk; some have a poor integration with OS synchronization mechanisms, so that their use from applications is inefficient.

Nevertheless, there are good ideas in many of these proposals that deserve to be considered, and this is what we do in this work. Our system, called **netmap**, integrates the strengths of existing proposals while at the same time addressing their limitations. The result is that **netmap** provides extremely fast access to raw network packets, both for transmission and reception,

*Copyright 2011 Luigi Rizzo, Università di Pisa. Work partially supported by Intel Research Berkeley, and by EU FP7 project CHANGE. This paper is a preliminary version of a conference submission, please do not redistribute directly, instead refer interested readers to the URL in the title where they can find updated versions and other material.

while at the same time keeping the operating system safe from disruption that an application might cause, and using APIs that permit easy and efficient porting of applications.

One metric to evaluate our system is performance: in our implementation, moving one packet between the wire and the userspace application takes only 90 CPU clock cycles, which is one order of magnitude less than in standard drivers. In other words, a single core running at 1.33 GHz can source or sink the 14.88 Mpps achievable on a 10 Gbit/s link, and the same core running at 150 MHz saturates a 1 Gbit/s link.

Other, equally important, metrics are safety of operation and ease of use. **netmap** clients cannot possibly crash the system, because all memory accesses are protected, and applications cannot inject bogus memory pointers in the system. At the same time, **netmap** uses an extremely simple data model well suited to zero-copy packet forwarding; supports multi-queue adapters; and uses standard system calls (such as `select()/poll()`) to achieve synchronization between the hardware and the software. All this makes it very easy to port existing applications to the new mechanism, and to write new ones that exploit the **netmap** API.

The paper is structured as follows. In Section 2 we discuss the motivation of our work, and show how network adapters are normally managed in operating systems. Section 3 presents related work, presenting some of the techniques that **netmap** integrates and extends to provide a safe and fast architecture for raw packet access. Section 4 describes in detail our system, motivating our the design choices. Applications of **netmap** are discussed in Section 5, while performance data are presented in Section 6, which explores the behaviour of the system in terms of throughput, CPU load and also energy use. Finally we conclude the paper discussing the current status and availability of the system, and plans for future work.

2. MOTIVATIONS AND BACKGROUND

There has always been interest in running, on general purpose operating systems, applications such as software switches [14] or routers [7, 4, 6], various types of firewall, traffic monitors, intrusion detection systems, and even traffic generators.

There is no doubt that a general purpose OS provides a convenient, flexible, rich software environment for running such applications. The software provides all sorts of functionalities so that applications requiring memory, databases, presentation capabilities find all they need at no cost. Additionally, the hardware where these OS run tends to have the best price/performance ratios, and decent I/O capabilities, especially for applications that require a modest number of network ports.

It is unfortunate that general purpose OS normally do

not provide efficient mechanisms to access raw packet data at high packet rates. Thus, the focus of this paper is to address this limitation.

As a matter of fact, the problem of coping with high packet rates is not limited to applications running in user space. There has always been a tension between the nominal speed of communication links, and the ability of systems to cope with their maximum packet rates. Link speeds grow over time in steps of one order of magnitude, with 10 Gbit/s and 14.88 Mpps being increasingly available nowadays. The speed of I/O and memory buses (other potential bottlenecks) also improve in large steps, not necessarily synchronized with link speeds. As of this writing, a PCI-Express bus reaches 4 GBytes/s, whereas the memory bandwidth is between 6 and 10 Gbytes/s on modern systems. CPU speeds increase much more gradually, with clock speeds topping around 3.5..4 GHz while the number of cores grows, sometimes to two digit figures.

As a result of all these factors, the ability to move packets between the network and applications changes over time, and depends critically on the relative speeds of the subsystems involved.

Hardware is not the only cause of performance problems. In this respect, the design of the software interface between network device drivers and the OS is in many cases over 20 years old, and the next subsections will describe it so that we can understand its limitations.

2.1 NIC data structures and operation

Network adapters (NICs) normally manage incoming and outgoing packets through circular queues (*rings*) of buffer descriptors, or slots, as in Figure 1. Each slot includes length and physical address of the buffer, and some flags to control the buffer’s handling by the NIC. CPU-accessible registers in the NIC indicate the active portion of the ring, indicating the buffers available for transmission or reception.

On reception, incoming packets are stored in the next available buffer (possibly split on multiple buffers if they

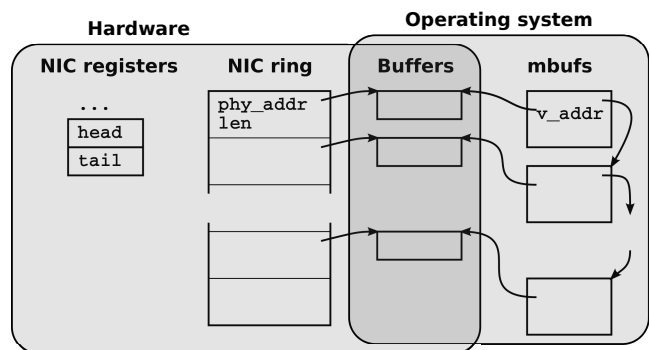


Figure 1: Typical NIC’s data structures and their relation with the OS data structures.

exceed the available buffer size), and length/status information is written back to the slot to indicate the availability of new data. Interrupts notify the CPU of these events.

On the transmit side, the NIC expects buffers filled with data to transmit. The request to send new packets is issued writing to a register in the NIC, which in turn starts sending packets marked as available in the TX ring. This arrangement is almost universal among modern NICs; variations are mostly related to ring size, the format of the slots, and restrictions on buffer sizes and alignments.

Some high speed NICs support multiple transmit and receive rings. This helps spreading the load on multiple CPU cores, eases on-NIC traffic filtering, and also improves decoupling virtual machines sharing the same hardware.

2.2 Interface between the NIC and the OS

The OS maintains shadow copies of the NIC’s data structures. Buffers are linked to OS-specific, but device-independent containers (`mbufs` [17] or equivalent structures such as `skbufs` and `NdisPackets`), transporting large amounts of metadata about each packet: size, source or destination interface, and all sort of attributes and command/status bits to indicate how the buffers should be processed by the NIC and the OS. The software interface assumes that packets in both directions can be split into an arbitrary number of fragments, so both the device drivers and the host stack must be prepared to handle the fragmentation.

This assumption makes the interface very expensive, because fragment descriptors must be allocated, filled and deallocated individually. Some producers do indeed generate fragmented packets (e.g. TCP when prepending headers to data from the socket buffers, or certain NICs when the packet size exceed the buffer size), but the cost of carrying these buffer chains all the way up or down the protocol stack often exceeds the cost of enforcing the use of linearized data.

Similarly, handling the variety of requests (checksum offloading, vlan tag handling, TCP segmentation) from the OS to the driver might at times exceed the cost of doing the same operation on the host CPU.

2.3 Access from user programs

The standard APIs to read/write raw packets on standard operating systems require at least one memory copy to move data and metadata between kernel and user space, and one system call per packet (or, in the best cases, per batch of packets). As discussed in Section 2, copy costs are becoming smaller and smaller over time as clock speeds and bus widths increase. System calls, on the other hand, do not improve so fast because they are constrained by latencies more than

clock frequencies. Typical approaches involve opening a socket, or a Berkeley Packet Filter [13] device, and doing regular I/O through it.

3. RELATED WORK

In this section we describe several options designed to give user programs efficient access to packet data. To the best of our knowledge, however, none of the existing proposal has the same set of features as the `netmap` system which is the main contribution of this paper.

The Berkeley Packet Filter, or BPF [13], is one of the most popular systems to support direct access to raw packet data. It works by tapping into the data path of a network device driver, and dispatching a copy of each received or transmitted packet to a special device, from which userspace processes can read or write. Linux has a similar mechanism through the `AF_PACKET` socket family. BPF can coexist with regular traffic from/to the system, but usually BPF clients need to put the card in promiscuous mode, causing large amounts of traffic to be delivered to the host stack (and immediately dropped). At high packet rates this overhead can be significant.

Another cost element, at least in the past and at high packet rates, was related to interrupt processing. Early hardware generated interrupts at high rates [15], which caused a number of undesirable phenomena ranging from high per-packet processing costs to the so-called “receive livelock”, or inability to do any useful work under extreme load. To overcome this problem, some systems use polling device drivers [9, 15, 16], and modern NICs implement interrupt mitigation directly in the hardware.

3.1 Packet filter hooks

While BPF was designed to capture and generate packets without altering regular traffic from/to the system, this is not always a requirement or even a desire. As an example, monitoring tools might want to keep the monitoring interface well separated from the rest of the system; firewalls and NATs *must* filter or modify traffic before it gets to the host stack; and in software routers or switches, most of the traffic does not need to be passed to the host stack at all. In these cases, a more efficient mechanisms is to implement packet hooks such as Netgraph (FreeBSD), Netfilter (Linux), Ndis Miniport drivers (Windows). These hooks permit to intercept traffic from/to the driver and pass it to processing modules without additional data copies. Note however that even the packet filter hooks rely on the standard mbuf/skbuf based packet representation, so the cost of metadata management (Section 2.2) still remains.

Netslice [12] and PacketShader [6] remove this overhead by binding a NIC ring to a device descriptors, from which userspace applications can read/write the

content of the packets buffers without creating metadata. The cost of data copying and of system calls can be amortized on multiple packets, so they do not impact performance too much.

3.2 Direct buffer access

Transferring information between kernel and user space involves system calls and data copies. Even on a fast CPU, a system call easily take between 200 to 500 ns. Data copying is inversely proportional to memory bandwidth, so it becomes less and less important as system performance increases.

One way to remove the above costs is to run the application code directly within the kernel. Systems based on kernel-mode Click [9, 4] follow this approach. Click permits an easy construction of packet processing chains through the composition of modules, some of which support fast access to the NIC bypassing the OS.

The kernel environment is however much more limited than the one available in user space, so this approach is not always feasible. As a consequence, a number of recent proposals try to remove the system call overhead by exposing to user space the data structures managed by the NIC. This approach generally requires modified device drivers, and its use poses some stability risks at runtime, because the NIC has a DMA engine that can write to arbitrary memory addresses, and potentially trash data anywhere in the system if the wrong values are written to the NIC's data structures.

UIO-IXGBE [10] implements exactly what we have described above: buffers, hardware rings and NIC registers (see Figure 1) are directly accessible to user programs, with obvious risks for the stability of the system.

PF_RING [2] is a packet-capture solution somewhat related to direct buffer access. In PF_RING, user processes have access, through memory mapping, to a ring of pre-allocated buffers which contain packet data. The buffers are filled in software by copying the content of the mbufs/skbufs returned by the device driver. No individual driver modifications are needed. Compared to BPF, the fact that buffers are preallocated saves the per-packet allocation costs.

An evolution of PF_RING called DNA [3] avoids the copy because the memory mapped ring buffers are directly accessed by the NIC. Same as UIO-IXGBE, DNA clients have direct access to the registers and rings of the NIC.

Ringmap [5] is a packet capture solution with some similarities with the work presented here. It exports packet buffers to and a shadow ring to userspace programs using mmap. Unlike our work, it does not support the transmit side or communication with the host stack.

3.3 Hardware solutions

We conclude our survey mentioning a number of hardware cards designed specifically to support high speed packet capture, or possibly generation, together with special features such as timestamping. Usually these cards come with custom device drivers and user libraries to access the hardware. As an example, DAG [1, 8] cards are FPGA-based devices designed to achieve wire-rate packet capture and precise timestamping. These cards use on-board memory to avoid depending on the performance of the bus (of course, access from the CPU will still have to go through the system bus). NetFPGA [11] is another example of an FPGA-based card where the firmware of the card can be programmed to implement specific functions directly in the NIC, offloading the main CPU from some work.

4. NETMAP

From the discussion in the previous Sections it appears clearly that that high cost operations in accessing packet data include data copying, metadata management, and system call overhead. Removing metadata and exposing the NIC's packet buffers to user applications is certainly useful to remove or reduce the cost of the above operations.

Our proposal, called **netmap**, is a system to give user space applications very fast access to network packets, both on the receive and the transmit side, and including those from/to the host stack. Efficiency does not come at the expense of safety of operation: potentially dangerous actions such as programming the NIC are validated by the OS, which also enforces memory protection.

netmap achieves its high performance through several techniques:

- a lightweight metadata format;
- linear, fixed size buffers;
- avoid copy costs granting applications direct, protected access to the packet buffers;
- support zero-copy transfer of packets between interfaces;
- efficient system calls to handle batches of packets with a single system call;
- support multiple hardware queues when available.

Overall, we use each part of the system for the task it is best suited to:

- the NIC moves data quickly between the network and memory;
- the OS enforces protection and provides support for synchronization.

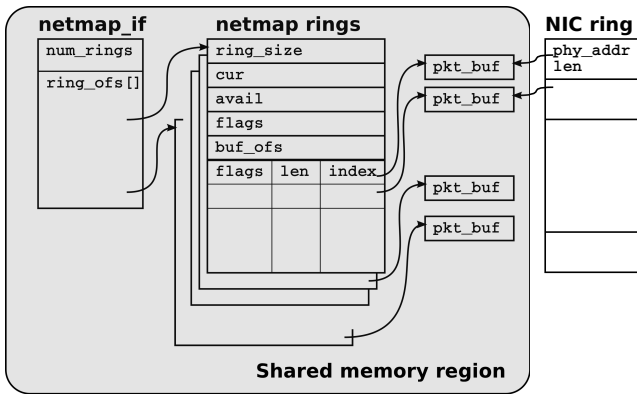


Figure 2: User view of the shared memory area exported by netmap.

A NIC in **netmap** mode is intentionally disconnected from the host protocol stack. The disconnection is only partial, so that management functions (to bring interfaces up and down, set speed, assign addresses) continue to work as usual; it is instead left to the application, if needed, to restore the flow of data packets between the network and the host (see Section 5).

4.1 Shared data structures

The data structures made available by **netmap** to applications are shown in Figure 2. These structures are designed to support efficiently the various functions that we are interested in, namely:

- reduce/amortize per-packet overheads;
- efficient forwarding between interfaces;
- efficient communication between the NIC and the host stack;
- support multi-queue adapters and multi core systems;
- provide effective event notification to processes.

netmap supports these features by associating, to each interface, three types of user-visible objects: *netmap_if* descriptors, *netmap rings*, and *packet buffers*.

All objects for all netmap-enabled interfaces in the system reside in one large memory region, allocated by the kernel in a non-pageable area, and shared by all user processes. Netmap rings are also accessed by the kernel while executing the system calls, and packet buffers are directly read/written by the NIC’s DMA engine.

Since the memory region is mapped by processes and kernel threads in different virtual address spaces, any memory reference contained in there must use *relative* addresses, so that pointers can be calculated in a position-independent way. The solution to this problem is to implement references as offsets between the parent and child data structures.

Packet buffers are just fixed-size buffers (2 Kbytes in the current implementation) used by the NICs and user processes to exchange packet data. Each buffer is identified by a unique *index*, which can be easily translated into a virtual address by user processes or by the kernel, and into a physical address to be passed to the NIC to indicate where to read/write packet data. Buffers are preallocated for all netmap rings when the interface is put into netmap mode, so that during network I/O there is never need to allocate memory, or risk to run out of resources. The metadata describing the buffer (index, data length, some flags) are stored into *slots* which are part of the netmap rings described next. Each buffer is referenced by a netmap ring and by the corresponding hardware ring.

A *netmap ring* is a device-independent representation of the circular queue implemented by the hardware. This data structure contains the following fields:

- **ring_size**, the size of the ring. This is used to determine when indexes need to wrap;
- **cur**, the index of the current read or write position in the ring;
- **avail**, the number of available buffers, representing received packets in RX rings, or empty slots in TX rings;
- **flags**, to indicate special conditions such as empty TX ring or errors;
- **buf_ofs**, indicating the memory offset between the ring and the beginning of the memory area containing buffers;
- **slots[]**, an array with **ring_size** entries. Each slot contains the index of the corresponding packet buffer, the length of the packet stored in there, and again some flags used to request special operations on the buffer.

The **buf_ofs** field is especially important because using this information and the ring address, processes can locate any packet buffer in the system as follows:

```
(char *)ring + ring->buf_ofs + i * buf_len
```

Finally, a *netmap_if* contains read-only information describing the interface, such as the number of rings and an array with the memory offsets between the **netmap_if** and each netmap ring associated to the interface (once again, offsets are used to make addressing position-independent).

4.2 The netmap API

Programs put an interface in **netmap** mode by opening the special device `/dev/netmap` and issuing an

```
ioctl(.., NIOCREG, arg)
```

on the file descriptor. The argument contains the interface name, and optionally the indication of which rings

we want to control through this file descriptor (see Section 4.4). On success, the function returns the size of the shared memory region where all data structures are located, and the offset of the `netmap_if` within the region. The shared memory region can be mapped in the process' address space using the `mmap()` system call.

Once the file descriptor is associated to an interface, two more `ioctl()`s support the transmission and reception of packets. In particular, transmissions require the program to fill up to `avail` buffers in the TX ring, starting from slot `cur` (packet lengths are written to the `len` field of the slot), and then issue an

```
ioctl(.., NIOCTXSYNC)
```

to tell the system that there are new packets to transmit. This system call passes the information to the kernel, and on return it updates the `avail` field in the netmap ring reporting slots that have become available due to the completion of previous transmissions.

On the receive side, programs should first issue an

```
ioctl(.., NIOCRXSYNC)
```

to ask the system how many packets are available for reading; then their lengths and payload are immediately available through the content of the slots (starting from `cur`) in the netmap ring.

Both `NIOC*SYNC ioctl()`s are non blocking, involve no data copying, and can deal with multiple packets at once. These features are essential to reduce the per-packet overhead to very small values. The in-kernel part of these system calls does the following:

- locates the interface and rings involved. This is done through a private kernel copy of the `netmap_if` which cannot be modified by user processes;
- validates the `cur` field and the content of the slots involved (lengths and buffer indexes);
- updates the values of in the hardware rings accordingly, and issues commands to the NIC to advertise new packets to send or newly receive buffers;
- updates the `avail` field in the netmap ring.

The amount of work in the kernel is kept to a minimum, and the checks performed make sure that any value written to the shared data structure cannot cause system crashes.

4.3 Blocking primitives

Support for blocking operation is provided through the well known `select()` and `poll()` system calls. Netmap file descriptors can be passed to such functions, and are reported as ready (causing the process to be woken up) when `avail > 0`. Before returning from a potentially blocking `select()/poll()`, the system executes the kernel part of the `NIOC*SYNC` functions.

This way, applications spinning on an eventloop require only one system call per iteration.

4.4 Multi-queue interfaces

On cards with a single TX/RX ring, a file descriptor operates on the only ring pair available. For cards with multiple ring pairs, file descriptors (and the related `ioctl()` and `poll()`) can be configured in two ways: i) act on all rings; ii) act on a single TX/RX ring pair. The former mode (which is the default), is convenient when we want to use a single thread to manage one interface. In this case, the kernel checks for available space/data on all rings¹. The second mode associates each ring pair to a different file descriptor. This can be useful to split the workload between multiple processes/threads, possibly bound to different CPU cores.

The operating mode is chosen by setting the `ring_id` field in the argument of the `NIOCREG ioctl()`. Setting the value to -1 (default) uses all rings at once, setting it to $0..N - 1$ (where N is the number of ring pairs) uses a single pair.

Note that the binding of a file descriptor to interfaces and ring pairs only affects the system calls issued through that file descriptor. Apart from that, any process that has `mmapped` the `/dev/netmap` memory region and knows where the data structures are located can update any of the rings or buffers. This permits very useful forms of cooperation and efficient data management, provided of course that the processes act with due coordination. This requirement is inherent in all applications based on shared memory.

4.5 Talking to the host stack

In addition to the ring pairs associated to the N hardware ring pairs, each interface in `netmap` mode exports an additional ring pair, used to handle packets to/from the host stack. This ring pair is selected by setting `ring_id = -2` when calling `NIOCREG`. When operating on the “host stack” netmap rings, `NIOCTXSYNC` encapsulates buffers into mbufs and then passes them to the host stack, whereas packets coming from the host stack are copied to the shared buffers so that subsequent `NIOCRXSYNC` will report them as “received” packets.

4.6 Comparison with existing proposals

`netmap` has some similarities with several of the solutions presented in Section 3, but the overlap with individual proposals is only partial.

In some respects, the closest relative to `netmap` is `RINGMAP` [5], though it only supports packet cap-

¹In the current implementation, the multiple rings are exposed to the user space application even in this mode, so the application has to scan the rings for data. We are considering a modification of the API that, when using a single file descriptor per NIC, manipulates the rings so that user programs only see one “virtual” netmap ring.

ture. Packetshader [6] has some other similarities to our work, namely the removal of driver’s metadata and support for batched I/O, but relies on copying to transfer packet data. PF_RING uses the idea of “shadow rings” and memory mapped buffers, but it only supports packet capture. DNA and UIO-IXGBE expose the NIC’s registers and rings directly to the user space process, which is efficient but extremely dangerous for system stability. Note that both these proposals do not work with packets from/to the host stack, and use a metadata format which is tied to a specific NIC.

5. APPLICATIONS

Despite its the simplicity, the **netmap** API supports efficiently all sorts of packet manipulation applications. Their development is also made very convenient by the fact that code runs in user space and the system is protected by crashes induced by misbehaving applications. In this Section we provide a brief survey of tools that we have built or are building using **netmap**.

Traffic generators

Line rate traffic generators are normally used to evaluate the performance of packet processing systems under different traffic patterns. It is especially important to be able to generate traffic with variable features (addresses/flow IDs, packet sizes, content) because these often influence the cost of certain operation (lookups, state allocation, etc.) and the scalability of a system.

netmap is very well suited for this kind of application, because creating packets is very simple and efficient. In its simplest form, a traffic generator can simply loop around a poll(), filling available TX rings as needed, as follows.

```
for (;;) {
    poll(fdlist, 1, INFINITE);
    while (ring->avail-- > 0) {
        i = ring->cur;
        buf_addr = NETMAP_BUF(ring, ring->slots[i].index);
        ring->slot[i].len = fill_buffer(buf_addr);
        ring->cur = NETMAP_NEXT(ring, ring->cur);
    }
}
```

As we will see in Section 6, a single core is able to saturate a 10 Gbit/s interface even at the smallest packet sizes. In case the packet generation turns out to be expensive, a generator can spawn multiple processes/threads to handle, independently of each other, the queues of multi queue adapter.

Traffic analyzers

With **netmap**, tools to capture traffic can have the same simple structure shown for generators, replacing generation with packet analysis. Depending on the complexity of the task we might just want to count packets, or perform some analysis on their content, in which

case the processing might become more expensive and require the use of multiple cores. Once again, **netmap** supports this in a straightforward way.

Firewalls, NATs and other packet filters

These applications normally reside between the network interface and the host stack, and filter or manipulate traffic going through the two. With **netmap** we can put a packet filter in place by opening two file descriptors, and attaching one to the host stack (**ring_id** = -2) and the other to the NIC’s rings (**ring_id** = -1). Then we can simply poll the two descriptors to wait for traffic, and process it as requested. In the case of firewalls, we can use zero-copy operation to pass allowed traffic up and down, and simply skip unwanted buffers from the receive rings, with no allocation and deallocation costs. NAT or other manipulations that do not alter the packet size also require zero copy, because the packet buffers can be modified in place to rewrite addresses, checksums and the like.

It is especially interesting that in our tests, putting an interface in netmap mode and connecting it to the host stack with a “pass-all” filter gives slightly better performance than the use of the driver in standard mode.

Software switches/routers

A variant of the previous category, software switches or routers deal with multiple ports and normally can be implemented with zero-copy solutions. At the highest data rates, multiqueue adapters can help build very efficient systems that exploit the presence of multiple cores. A typical arrangement might dedicate one core for each of the M input interfaces, and M output queues for each output interface. With this approach, cores can work independently without contending for the output queues.

Compatibility libraries

A large number of legacy applications uses the libpcap API to access packets. An adaptation library that maps libpcap calls over netmap is straightforward, because **netmap** already provides the poll()-able file descriptor used by pcap clients, and the read routine can be implemented on top of **netmap** with almost no cost. With this approach, applications can use the improved performance with no modifications.

6. PERFORMANCE ANALYSIS

netmap dramatically reduces the per-packet overheads compared to the ordinary host stack. As an example, Fig.2 shows the send rate of a packet generator using an Intel 10G card, and an i7-870 CPU at different clock frequencies from 150 MHz to 2.93 GHz. One core achieves line rate (14.88Mpps) at just 1.3GHz, and even at 150 MHz can push out 1.76 Mpps. These num-

bers correspond to roughly 90 clock cycles/packet. The receive side gives similar numbers.

Before discussing the performance numbers, it is important to define the test conditions in detail.

6.1 Performance metrics

The measurement of the performance of a packet processing system must take into account that there are two cost components (*per-packet* and *per-byte*). That these costs must be further decomposed into *system* and *application* costs, where the former indicate the work spent to move data to/from the application, and the other part measures the application-specific processing costs.

Given that **netmap** is a mechanism to transport packets to/from applications, we focus our attention on the *system* costs, at least if the way we make data available to applications does not impose limitations on their use². As a consequence, in our tests we measure the performance of a very simple packet generator, and of a receiver which just counts packets: these experiments reduce application costs to a minimum, allowing us to focus on the system costs.

Regarding the per-packet and per-byte costs, the latter are usually proportional to the memory bandwidth available in the system, which, as a rule of thumb, are in the range of 1/4 to 2 clock cycles per byte (taking into account the width of memory buses and the ratio between CPU and memory bus speed). The most challenging situation is usually with the shortest packet sizes, where per-packet costs are almost unavoidably dominant over the memory copy costs. This justifies why we will run most of our tests with 64 byte packets (60 + 4 for the Ethernet CRC).

In **netmap**, in particular, the per-byte system CPU cost is exactly zero because all data transfers are performed by the NIC, and it is up to applications to access memory, if it really needs to.

As a final remark on testing, we should note that there are other system components (I/O and memory buses and related interface logic) which sometimes can act as bottlenecks. An example will be given in Section 6.6.

We should also remember that at the speeds at which we are operating, sometimes even minor issues such as memory alignment and data layout can have a significant impact on the absolute throughput.

6.2 Test equipment (hardware and software)

We have run most of our tests on a system equipped with an i7-870 4-core CPU running at a top speed of ²as an example, it would be incorrect to disregard application costs if moving data from one interface to another required multiple additional data copies; but in our case, **netmap** even permits zero-copy packet forwarding so it is correct to follow this approach.

2.93GHz, memory running at 1.33 GHz, and a dual port 10 Gbit/s card based on the Intel 82599 chip mounted on a x16 PCI-Express slot (only 8 lanes are used by the card). The same machine hosts two 1 Gbit cards, mounted on different PCI-Express buses. The operating system is FreeBSD/i386 as of May 2011 (both HEAD and RELENG_8 exhibit similar performance). Tests have been run using directly connected cards on two similar systems. The results were highly repeatable (within 0.5% or less) so we do not report variance in the tables and graphs.

Our system can be clocked at different frequencies, which is useful to determine the limits of the system. As we will see, **netmap** is extremely efficient so the CPU is mostly idle even when the interface is running at the maximum packet rates. Running the system at reduced clock speeds also helps determining the effect of small performance improvements.

To test the transmit speed (in packets per second), we built a simplified packet generator that transmits variable-size IP packets. The packet size is set as a command line option. The (negligible) application cost to prepare each packet is limited to copying the Ethernet and IP headers. Another configurable parameter is the number of rings used by the NIC, and the number of threads running in user space to drive the rings. This gives some indication on the improvements that can be achieved using multiple cores, and on the interference between threads.

The test program loops around a poll(), sending at most B packets per ring at each round, where B is the burst size. The use of different burst sizes lets us evaluate the system call overhead, which is important to know when we cannot freely decide the size of the bursts. On the receive side we use a similar program, except that this time we poll for read events and only count packets.

6.3 Transmit speed

The first test for transmit speed looks at how fast we can generate minimum-size packets (64 bytes, including the Ethernet CRC) at maximum clock rate, with variable burst size and number of rings. In this particular test we only use one core, and the goals are to measure the system call and synchronization overhead, and whether the hardware of the system is able to cope with the highest data rates.

Results are summarized in Figure 3. Even with modest burst sizes, using at least 2 rings we can easily send packets at line rate, and low CPU occupation. The cost of the poll/NIOCTXSYNC function can be inferred looking at the experiment with a burst size of 1 packet. With one ring, we have 2.24 million of packets (i.e. iterations) per second, so one iteration of the send loop takes approximately $1/2.24 = 446$ ns. This includes the

Burst size	1 ring	2 rings	4 rings
1	2.24	3.62	5.71
2	4.28	6.55	9.60
4	7.41	10.06	13.22
8	11.75	13.03	14.88
16	12.48	14.88	14.88
32	12.53	14.88	14.88
64	12.63	14.88	14.88
128	12.67	14.88	14.88
256	12.69	14.88	14.88
512	12.71	14.88	14.88

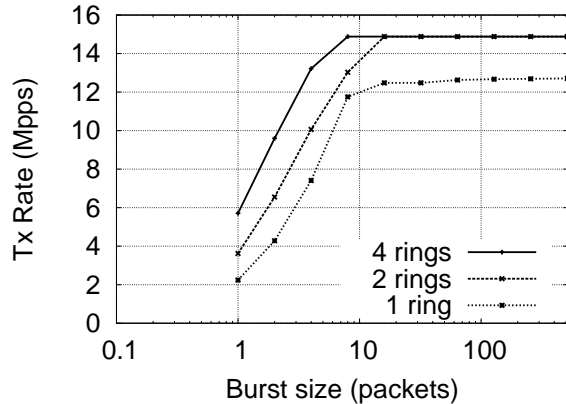


Figure 3: Transmit performance with 1 core at 2.93 GHz, 64-byte packets, and variable number of rings and burst size. Rates are in Million of packets per second (Mpps).

system call overhead and locking costs. With multiple rings used through a single file descriptor, both the user program and the kernel have to loop over all rings, and use an additional lock to synchronize. This means that the poll/NIOCTXSYNC cost becomes larger. With 2 and 4 rings the number of iterations per second is 3.62/2 and 5.71/4, respectively, translating in 552 and 700 ns per iteration (part of this time is spent in the user program).

The system call cost can be easily amortized by using larger batch sizes. The tests with 2 and more rings show that the hardware is capable of sustaining line rate. The behaviour with one ring shows saturation at around 12.7 Mpps, even if there is plenty of CPU cycles available. The reasons are not known precisely, but we suspect this to be a limitation of the hardware, which is unable to make full use of the bus capacity.

Also note that these results have been achieved disabling interrupts and error reporting on most slots. With error reporting on all slots, the throughput with one ring tops around 6 Mpps; once again, this is probably due to limitations of the NIC’s hardware dealing

Clock (GHz)	1 core 4 rings	2 cores 2 rings	4 cores 4 rings
0.150	1.67	2.77	5.34
0.300	3.39	5.79	10.90
0.450	5.22	8.80	14.88
0.600	6.85	11.34	14.88
0.750	8.34	14.04	14.88
0.900	10.38	14.88	14.88
1.050	12.11	14.88	14.88
1.200	13.95	14.88	14.88
1.333	14.88	14.88	14.88

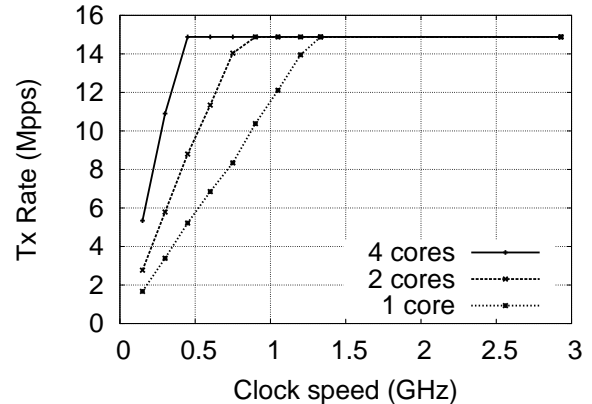


Figure 4: Transmit performance with 64-byte packets, variable clock rates and number of cores. Above 1.33 GHz, the system reaches line rate.

with bus traffic, because the CPU has spare cycles. Note that minimizing error reporting and interrupts on the transmit side is extremely beneficial for reducing the system’s load, and has no undesirable side effects (the NIC’s hardware has internal counters for errors and other statistics, so no information is lost).

6.4 Transmit speed versus clock rate

A better indication of the transmit performance is given running the experiments at reduced clock speeds (and a suitably large burst size). By lowering the clock frequency we can determine the point where the CPU actually becomes the bottleneck, and estimate an (amortized) number of cycles spent for each packet. Experimental results are in Figure 4. We tested the configuration with 1 core/4 rings (to overcome the 12 Mpps limit with 1 ring), and then 2 and 4 cores with an equal number of rings.

We see that the throughput scales quite linearly with clock speed, reaching the maximum line rate between 1.2 and 1.33 GHz (we only have discrete frequency levels). Dividing the clock rate by the packet rate we find an average of 90 cycles/packet with 1 core, and

marginally higher values (between 105 and 110) with 2 and 4 cores.

These numbers are almost one order of magnitude lower than those for the standard drivers, and are of particular interest when we think that a 150 MHz CPU is able to saturate a 1 Gbit/s link.

6.5 Receive speed

The speed vs. clock rate experiments on the receive path give results similar to the transmit section, so we omit the numbers for space reasons. The numbers are actually slightly better in terms of CPU usage, because the receive path only counts packet and does not have to write into the buffers.

Interrupt mitigation is used in the receive experiments to limit the number of receive interrupts to 62500 interrupts/second. Unlike the transmit path, interrupts on the receive side need to be enabled on every packet or at least with a sufficiently short period, or the software would be notified too late of incoming traffic. The values chosen correspond to a worst case latency of about 16 μ s in reporting packet availability.

6.6 Speed versus packet size

The experiment so far dealt with minimum-size packets, which is the most critical situation in terms of per-packet overhead. 64-byte packets match very well the bus widths along the various path in the system and this helps the performance of the system. We then checked whether varying the packet size has an impact on the performance of the system, both on the transmit and the receive side.

Transmit speeds with variable packet sizes exhibit the expected $1/size$ behaviour, as shown by the upper curve in Figure 5.

The receive side, instead, shows some surprises as shown by the bottom curve in Figure 5. The maximum rate, irrespective of CPU speed and number of rings, is achieved only for certain packet sizes such as 64+4, 128+4 and above 144+4 bytes (in all cases, the extra 4 bytes corresponding to the Ethernet CRC is not written to memory). At other packet sizes, performance drops (e.g. we see 11.6 Mpps at 60 bytes, and a plateau around 8.5 Mpps between 65 and 127 bytes). We briefly investigated this behaviour looking at the NIC's statistic counters, and it turns out that the problem is not due to lack of CPU cycles, and the receive rings never run out of buffers. Rather, the NIC seems unable to transfer incoming data to memory, perhaps due to a poor interaction with the PCI-Express bus.

It makes more sense to analyse the performance drop in terms of time. 11.6 Mpps correspond to about 86 ns/packet, compared to the 67 ns/pkt corresponding to 14.8 Mpps. The plateau at 8.5 Mpps corresponds to 117 ns/pkt, to be compared to the 71 ns/pkt that we

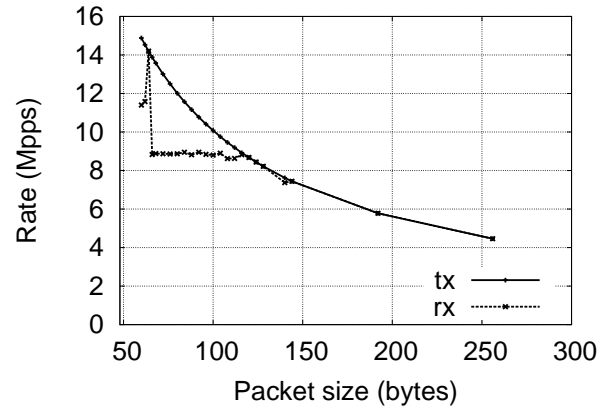


Figure 5: Actual transmit and receive speed with variable packet sizes (excluding Ethernet CRC). The top curve is the transmit rate, the bottom curve is the receive rate. See Section 6.6 for explanations.

should expect at 69 bytes per packet. The differences are small and likely to depend on the use of additional PCI-Express transactions when the packet sizes are not a multiple of 64. We should note that this behaviour has no architectural relation with the way **netmap** works, but it depends purely on the implementation of bus transactions on the controller and on the motherboard. Yet, in a very high speed packet processing system, we should be aware that these phenomena may occur and not immediately blame applications for strange performance numbers.

6.7 Application costs

We should remark that all tests so far measure the system costs involved in moving packets between the wire and memory. This includes the operating systems overhead, but excludes any significant application cost. Our generator simply writes a small memory block for each packet, and writes do not cause any significant CPU stall as they occur in parallel with other operations. Our receiver just counts packets, so once again there are no CPU stalls involved in accessing payload data.

It is expected that when we use **netmap** for specific applications, either the CPU load or the throughput may change significantly, depending on the access pattern and amount of work performed by applications. At very high packet rates (e.g. the 67 ns/pkt corresponding to line rate at 10 Gbit/s) even CPU stalls waiting for data might have a significant impact on the performance of the system. Thus, it is recommended that applications are coded in a way that minimizes these stalls (e.g. by issuing prefetch instructions well before the actual data are needed, or using multiple queues

Clock	Total power (W, +/- 2%)				
	Idle	1 core	4 cores	tx(1)	tx(4)
150	70	71	72	72	75
300	70	72	73	73	76
600	70	72	77	75	81
1050	70	72	79	79	84
1200	71	74	82	79	84
1467	71	76	87	84	86
2133	71	83	102	97	98
2400	71	87	110	101	102
2800	71	92	124	108	110
2934* (Turbo)	71	108	155	128	129

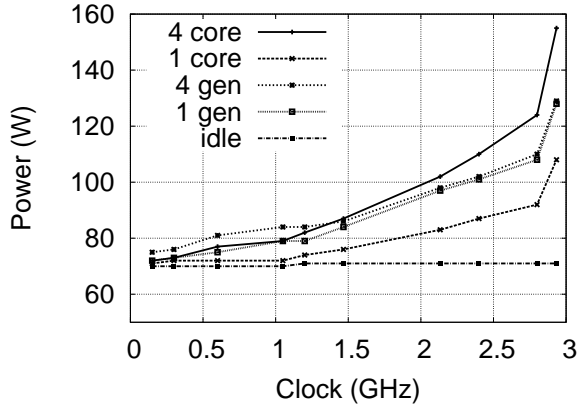


Figure 6: Total system power in various conditions.

and hyperthreading to relax the timing requirements of the system).

6.8 Energy considerations

To complete our evaluation of the system, we have run a set of tests to measure the power consumption of the system in various conditions. In fact, the same performance level in terms of packet rates can be achieved with different approaches (e.g. aggressive device polling or interrupt rates; short bursts; variable tradeoffs between number of cores and clock speed), and it might be interesting to see the impact of the various approaches on energy efficiency.

In modern CPUs, parts of the chip can be put in a low power state following a HALT instructions. This happens, for instance, when a core has no processes to execute. Also, the power consumption of an active block has the form $P = c + kfV^2$ where the term c represents the “static power”, mostly independent of the system’s clock, and the second terms represents the “dynamic power”, which grows with the switching frequency. In practice, the supply voltage V is also increasing with the operating frequency, so that P grows more than

linearly with frequency.

One way to make use of this relation is to determine, for a given workload, the best tradeoff between clock frequency and number of cores. If we assume that the number of clock cycles consumed for a workload does not depend on the operating frequency, then maximising the number of cores and keeping the clock frequency as low as possible is the choice that improves the energy efficiency of the system.

However, there are other effects that might suggest a different strategy. On the one hand, operating at a lower frequency might also reduce the number of CPU stalls waiting for memory or I/O transactions. On the other hand, spreading a workload among multiple cores might incur some performance penalty due to contention for shared resources (e.g. cache, buses) and arbitration times. We do not want to explore the tradeoffs in this paper, but just provide some initial data on the topic. In Figure 6 we measure the total power consumption of our system in various conditions. We have made no specific effort to minimize the total power or measure individual components. The graph simply plots the reading of a power meter on the AC line driving the power supply of the system.

The lower line at around 70 W corresponds to the system up and running but with no active processes. The only significant activity is represented by timer interrupts, occurring at a frequency of 1 KHz. These task only run for a very short time before executing the HALT instruction. This makes the idle power approximately constant irrespective of the clock frequency.

The next two reference curves are labeled “1 core” and “4 core”, and are generated by running one and four processes spinning on an infinite loop. With only one process, the package activates only one core and the “uncore” blocks (cache and bus controllers, etc.). With four processes, we can see the contribution of all four cores, with a significant increase on the total system power. From the table data we can see that the power versus frequency behaviour is significantly superlinear.

The final two curves represent the total system power when we use our traffic generator running on one and four cores. Compared to the previous cases, now the network card is also active, which causes some extra power to be used. On the other hand, the CPU load is not as high as before, even with four cores, because the system already reaches link rate at 1.33 GHz. As a result, the curves with one and four generator processes show no significant difference once the link is saturated, and the difference with the “1 core” case is mostly due to the power consumed by the network card.

Note that the point labeled 2.93 actually corresponds to Turbo mode, meaning that the system’s clock speed can go up to 3.6 GHz if thermal constraints are satisfied. This explains the steep increase between the last two

points (2.8 GHz and 2.93 GHz).

7. STATUS AND FUTURE WORK

The current prototype of **netmap**, developed on FreeBSD, consists of about 2000 lines of code for device functions (ioctl, select/poll) and driver support, plus individual driver modifications (mostly mechanical, about 500 lines each) to interact with the netmap rings. The most tedious parts of the driver (initialization, controlling the PHY interface, etc.) do not need changes; this significantly simplifies development and support of newer hardware. To date, **netmap** support is available for the Intel 10 Gbit/s adapters (ixgbe driver), and for various 1 Gbit/s adapters (Intel, Realtek, Nvidia). Support for other devices is quickly being added³.

We are working on several of the applications listed in Section 5, including a PCAP library, and porting some third party code, noticeably the OpenVswitch software that implements the OpenFlow [14] protocol. All the code will be made publicly available before the end of the summer under a BSD license, and it is likely to become part of the standard FreeBSD distributions by then.

While it has been developed on FreeBSD, **netmap** has very few system dependencies, so that a port to Linux systems should be feasible with little effort, and in fact it is a planned activity.

8. CONCLUSIONS

In this paper we have presented **netmap**, a system that gives userspace applications a fast and safe channel to exchange raw packets with the network adapter. **netmap** can be used for packet capture (an area which has received a lot of attention in the past), packet generation, and for communication with the host stack. As such, it is usable for all tasks related to networking, including building high performance routers or switches, firewalls, and generic packet processing systems. Although it requires device driver modifications, **netmap** is not bound to any specific piece of hardware, and its design makes very reasonable assumptions on the capabilities of the hardware.

9. REFERENCES

- [1] The dag project. Technical report, University of Waikato, 2001.
- [2] L. Deri. Improving passive packet capture:beyond device polling. In *SANE 2004, Amsterdam*.
- [3] L. Deri. ncap: Wire-speed packet capture and transmission. In *Workshop on End-to-End Monitoring Techniques and Services*, pages 47–55. IEEE, 2005.
- [4] M. Dobrescu, N. Egi, K. Argyraki, B.G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy.

³Support for slower interfaces (e.g. 100 Mbit) is only interesting for embedded platforms, where CPU cycles tend to be scarce even for relatively low data rates.

- Routebricks: Exploiting parallelism to scale software routers. In *ACM SOSP*, pages 15–28, 2009.
- [5] Alexander Fiveg. Ringmap capturing stack for high performance packet capturing in freebsd. http://ringmap.googlecode.com/files/ringmap_slides.pdf, 2010.
 - [6] 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.
 - [7] M. Handley, O. Hodson, and E. Kohler. Xorp: An open platform for network research. *ACM SIGCOMM Computer Communication Review*, 33(1):53–57, 2003.
 - [8] A. Heyde and L. Stewart. Using the endace dag 3.7 gf card with freebsd 7.0. 2008.
 - [9] 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.
 - [10] Max Krasnyansky. Uio-ixgbe. <https://opensource.qualcomm.com/wiki/UIO-IXGBE>.
 - [11] 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. *IEEE Int. Conf. on Microelectronics Systems Education*, 2007.
 - [12] Tudor Marian. Operating systems abstractions for software packet processing in datacenters. *PhD Dissertation, Cornell University*, 2010.
 - [13] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter Conference*. USENIX Association, 1993.
 - [14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38:69–74, March 2008.
 - [15] J.C. Mogul and K.K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)*, 15(3):217–252, 1997.
 - [16] L. Rizzo. Polling versus interrupts in network device drivers. *BSDConEurope 2001*, 2001.
 - [17] W.R. Stevens and G.R. Wright. *TCP/IP illustrated (vol. 2): the implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.