netmap: a novel framework for fast packet I/O

Luigi Rizzo,[*] *Università di Pisa*
http://info.iet.unipi.it/∼luigi/netmap/

Abstract

Many applications (routers, traffic monitors, firewalls, etc.) need to send and receive packets at line rate even on very fast links. In this paper we present *netmap*, a novel framework that enables commodity operating systems to handle the millions of packets per seconds traversing 1..10 Gbit/s links, without requiring custom hardware or changes to applications.

In building *netmap*, we identified and successfully reduced or removed three main packet processing costs: per-packet dynamic memory allocations, removed by preallocating resources; system call overheads, amortized over large batches; and memory copies, eliminated by sharing buffers and metadata between kernel and userspace, while still protecting access to device registers and other kernel memory areas. Separately, some of these techniques have been used in the past. The novelty in our proposal is not only that we exceed the performance of most of previous work, but also that we provide an architecture that is tightly integrated with existing operating system primitives, not tied to specific hardware, and easy to use and maintain.

netmap has been implemented in FreeBSD for several 1 and 10 Gbit/s network adapters. In our prototype, a single core running at 900 MHz can send or receive 14.88 Mpps (the peak packet rate on 10 Gbit/s links). This is more than 20 times faster than conventional APIs. Large speedups (5x and more) are also achieved on user-space Click and other packet forwarding applications using a libpcap emulation library running on top of *netmap*.

1 Introduction

General purpose OS provide a rich and flexible environment for running, among others, many packet processing and network monitoring and testing tasks. The high rate raw packet I/O required by these applications is not the intended target of general purpose OSes. Raw sockets, the Berkeley Packet Filter [12] (BPF), the AF_SOCKET family, and equivalent APIs have been used to build all sorts of network monitors, traffic generators, and generic routing systems. Performance, however, is not adequate the millions of packets per second (*pps*) that can be present on 1..10 Gbit/s links. In search of better performance, some systems (see Section 3) either run completely in the kernel, or bypass the device driver and the entire network stack by exposing the NIC's data structures to user space applications. Efficient as they may be, many of these approaches depend on specific hardware features, give unprotected access to hardware, or are poorly integrated with the existing OS primitives.

The *netmap* framework presented in this paper combines and extends some of the ideas presented in the past trying to address their shortcomings. Besides giving huge speed improvements, *netmap* does not depend on specific hardware[1], has been fully integrated in the OS (FreeBSD) with minimal modifications, and supports unmodified libpcap clients, through a compatibility library.

One metric to evaluate our framework is performance: in our implementation, moving one packet between the wire and the userspace application takes less than 70 CPU clock cycles, which is at least one order of magnitude faster than standard APIs. In other words, a single core running at 900 MHz can source or sink the 14.88 Mpps achievable on a 10 Gbit/s link. The same core running at 150 MHz is well above the capacity of 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 device registers and critical kernel memory regions are not exposed to clients, and they cannot inject bogus memory pointers in the kernel (these are often vulnerabilities of schemes based

[*] This work was funded by the EU FP7 project CHANGE (257422).

[1] *netmap* can give isolation even without hardware mechanisms such as IOMMU or VMDq, and is orthogonal to hardware offloading and virtualization mechanisms (checksum, TSO, LRO, VMDc, etc.)

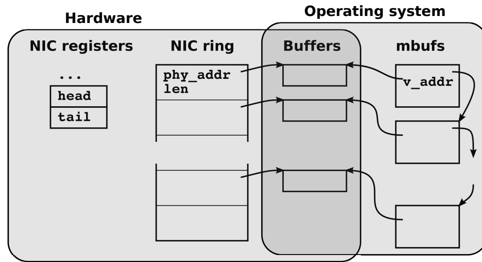

Figure 1: Typical NIC's data structures and their relation with the OS data structures.

on shared memory). 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 (`select()/poll()`) for event notification. All this makes it very easy to port existing applications to the new mechanism, and to write new ones that make effective use of the *netmap* API.

The rest of the paper is structured as follows. Section 2 completes the discussion of the motivations for this work, and gives some background on how network adapters are normally managed in operating systems. Section 3 presents related work, illustrating some of the techniques that *netmap* integrates and extends. Section 4 describes *netmap* in detail, Performance data are presented in Section 5. Finally, Section 6 discusses open issues and our plans for future work.

2 Motivations and background

There has always been interest in using general purpose hardware and Operating Systems to run applications such as software switches [13], routers [6, 4, 5], firewalls, traffic monitors, intrusion detection systems, or traffic generators. While providing a convenient development and runtime environment, such OSes normally do not offer efficient mechanisms to access raw packet data at high packet rates. Thus, the focus of this paper is to address this limitation.

2.1 NIC data structures and operation

Network adapters (NICs) normally manage incoming and outgoing packets through circular queues (*rings*) of buffer descriptors, as in Figure 1. Each slot in the ring contains the length and physical address of the buffer. CPU-accessible registers in the NIC indicate the portion of the ring available for transmission or reception.

On reception, incoming packets are stored in the next available buffer (possibly split in multiple fragments), 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 the OS to fill buffers with data to be sent. The request to send new packets is issued by writing into the registers of the NIC, which in turn starts sending packets marked as available in the TX ring.

At high packet rates, interrupt processing can be expensive and possibly lead to the so-called "receive livelock" [14], or inability to perform any useful work above a certain load. Polling device drivers [8, 14, 16] and the hardware interrupt mitigation implemented in recent NICs solve this problem.

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 helps decoupling virtual machines sharing the same hardware.

2.2 Kernel and user APIs

The OS maintains shadow copies of the NIC's data structures. Buffers are linked to OS-specific, device-independent containers (`mbufs` [18] or equivalent structures such as `skbufs` and `NdisPackets`). These containers include large amounts of metadata about each packet: size, source or destination interface, and attributes and command/status bits to indicate how the buffers should be processed by the NIC and the OS.

Driver/OS: The software interface between device drivers and the OS usually 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. The same API also expects that subsystems may retain packets for deferred processing, hence buffers and metadata cannot be simply passed by reference during function calls, but they must be copied or reference-counted. This flexibility is paid with a significant (and often unnecessary) overhead at runtime.

These API contracts, perhaps appropriate 20-30 years ago when they were designed, are extremely inadequate for today's systems. The cost of allocating, managing and navigating through buffer chains often exceeds that of linearizing their content, even when producers do indeed generate fragmented packets (e.g. TCP when prepending headers to data from the socket buffers).

Raw packet I/O: The standard APIs to read/write raw packets for user programs 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). Typical approaches involve opening a socket, or a Berkeley Packet Filter [12] device, and doing regular I/O through it using `read()/write()` or specialized `ioctl()` functions. System calls are the dominant source of overhead in these APIs.

In summary, the result of these hardware and soft-

ware architectures is that most systems barely reach 0.5..1 Mpps per core from userspace, and even remaining in the kernel yields only modest speed improvements, usually within a factor of 2.

3 Related (and unrelated) work

It is useful at this point to present some techniques proposed in the literature, or used in commercial systems, to improve packet processing speeds. This will be instrumental in understanding their advantages and limitations, and how our *netmap* framework can make use of them.

Socket APIs: The Berkeley Packet Filter, or BPF [12], is one of the most popular systems for direct access to raw packet data. BPF taps into the data path of a network device driver, and dispatches a copy of each sent or received packet to a file descriptor, 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).

Packet filter hooks: Netgraph (FreeBSD), Netfilter (Linux), Ndis Miniport drivers (Windows) are in-kernel mechanisms used when packet duplication (as in BPF) is not necessary, or the application (e.g. a firewall, or an IDS) manipulates traffic as part of a packet processing chain. 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 [11] is an example of a system that uses the netfilter hooks to export traffic to userspace processes through a suitable kernel module.

Direct buffer access: One easy way to remove the data copies involved in the kernel-userland transition is to run the application code directly within the kernel. Systems based on kernel-mode Click [8, 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 (even though they retain an skbuf-based packet representation).

The kernel environment is much more constrained than the one available in user space, so a number of recent proposals try a different approach: instead of running the application in the kernel, they remove the system call overhead by exposing NIC registers and data structures to user space. This approach generally requires modified device drivers, and poses some risks at runtime, because the NIC's DMA engine can write to arbitrary memory addresses (unless limited by hardware mechanisms such as IOMMUs), and a misbehaving client can potentially trash data anywhere in the system.

UIO-IXGBE [9] 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] exports to userspace clients a shared memory region containing a ring of pre-allocated packet buffers. The kernel is in charge of copying data between skbufs and the shared buffers, so the system is safe and no driver modifications are needed. This approach amortizes the system call costs over batches of packets, but retains the data copy and skbuf management overhead. 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 NIC's registers and rings.

The PacketShader [5] I/O engine (PSIOE) is one of the closest relatives to our proposals. PSIOE uses a custom device driver that replaces the skbuf-based API with a simpler one, using preallocated buffers. Custom `ioctl()`s are used to synchronize the kernel with userspace applications, and multiple packets are passed up and down through a memory area shared between the kernel and the application. The kernel is in charge of copying packet data between the shared memory and packet buffers. Unlike *netmap*, PSIOE only supports one specific network card, and does not support `select()/poll()`, requiring modifications to applications in order to let them use the new API.

Hardware solutions: Some hardware has been designed specifically to support high speed packet capture, or possibly generation, together with special features such as timestamping, filtering, forwarding. Usually these cards come with custom device drivers and user libraries to access the hardware. As an example, DAG [1, 7] cards are FPGA-based devices for wire-rate packet capture and precise timestamping, using fast onboard memory for the capture buffers (at the time, the I/O bus was unable to sustain line rate). NetFPGA [10] 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.

3.1 Unrelated work

A lot of commercial interest, in high speed networking, goes to TCP acceleration and hardware virtualization, so it is important to clarify where *netmap* stands in this respect. *netmap* **is a framework to reduce the cost of moving traffic between the hardware and the host stack**. Popular hardware features related to TCP acceleration, such as hardware checksumming or even

encryption, Tx Segmentation Offloading, Large Receive Offloading, are completely orthogonal to our proposal: they reduce some processing in the host stack but do not address the communication with the device. Similarly orthogonal are the features relates to virtualization, such as support for multiple hardware queues and the ability to assign traffic to specific queues (VMDq) and/or queues to specific virtual machines (VMDc, SR-IOV). We expect to run *netmap* within virtual machines, although it might be worthwhile (but not the focus of this paper) to explore how the ideas used in *netmap* could be used within a hypervisor to help the virtualization of network hardware.

4 Netmap

The previous survey shows that most related proposals have identified, and tried to remove, the following high cost operations in packet processing: data copying, metadata management, and system call overhead.

Our framework, 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. Also, a distinctive feature of *netmap* is the attempt to design and implement an API that is simple to use, tightly integrated with existing OS mechanisms, and not tied to a specific device or hardware features.

netmap achieves its high performance through several techniques:

- a lightweight metadata representation which is compact, easy to use, and hides device-specific features. Also, the representation supports processing of large number of packets in each system call, thus amortizing its cost;
- linear, fixed size packet buffers that are preallocated when the device is opened, thus saving the cost of per-packet allocations and deallocations;
- removal of data-copy costs by granting applications direct, protected access to the packet buffers. The same mechanism also supports zero-copy transfer of packets between interfaces;
- support of useful hardware features (such as multiple hardware queues).

Overall, we use each part of the system for the task it is best suited to: the NIC to move data quickly between the network and memory, and the OS to enforce protection and provide support for synchronization.

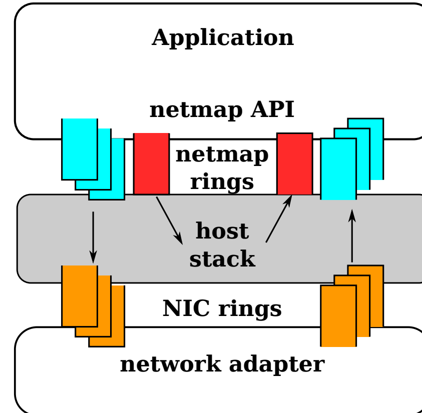

Figure 2: In netmap mode, the NIC rings are disconnected from the host network stack, and exchange packets through the netmap API. Two additional netmap rings let the application talk to the host stack.

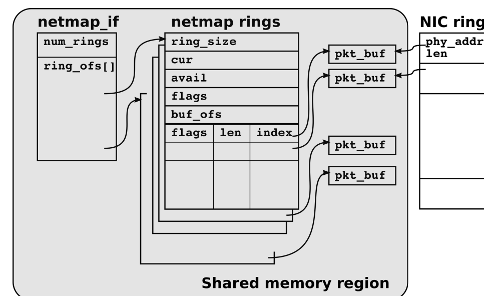

Figure 3: User view of the shared memory area exported by netmap.

At a very high level, when a program requests to put an interface in *netmap* mode, the NIC is partially disconnected (see Figure 2) from the host protocol stack. The program gains the ability to exchange packets with the NIC and (separately) with the host stack, through circular queues of buffers (*netmap rings*) implemented in shared memory. Traditional OS primitives such as `select()/poll()` are used for synchronization. Apart from the disconnection in the data path, the operating system is unaware of the change so it still continues to use and manage the interface as during regular operation.

4.1 Data structures

The key component in the *netmap* architecture is the set of data structures represented in Figure 3. These are designed to support efficiently following features: 1) reduced/amortized per-packet overheads; 2) efficient forwarding between interfaces; 3) efficient communication between the NIC and the host stack; and 4) support for multi-queue adapters and multi core systems.

netmap supports these features by associating, to each interface, three types of user-visible objects, shown in Figure 3: *packet buffers*, *netmap rings*, and *netmap_if* descriptors. 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. The use of a single buffer pool is just for convenience and not a strict requirement of the architecture. With little effort and almost negligible runtime overhead we can easily modify the API to have separate memory regions for different interfaces, or even introduce shadow copies of the data structures to reduce data sharing.

Since the memory region is mapped by processes and kernel threads in different virtual address spaces, any memory reference contained in that region 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 have fixed-size (2 Kbytes in the current implementation) and are shared by the NICs and user processes. 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 used by the NIC's DMA engines. Buffers for all netmap rings are preallocated when the interface is put into netmap mode, so that during network I/O there is never the need to allocate memory, or a 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 replica of the circular queue implemented by the NIC, and includes:

- `ring_size`, the number of slots in the ring;
- `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`, the offset between the ring and the beginning of the array of (fixed-size) packet buffers;
- `slots[]`, an array with `ring_size` entries. Each slot contains the index of the corresponding packet buffer, the length of the packet, and some flags used to request special operations on the buffer.

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).

4.1.1 Data ownership and access rules

The *netmap* data structures are shared between the kernel and userspace, but the ownership of the various data areas is well defined, so that there are no races. In particular, the *netmap_ring* is always owned by the userspace application except during the execution of a system call, when it is updated by the upper half of the kernel. The lower half of the kernel, which may be called by an interrupt, never touches a netmap ring.

Packet buffers between `cur` and `cur+avail-1` are owned by the userspace application, whereas the remaining buffers are owned by the kernel. The boundaries between these two regions are updated during system calls.

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 4.2.2). 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 (except from the synchronization of the slots in the netmap and hardware rings), 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, using 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, both in the netmap and hardware rings);
- synchronizes the content of the slots between the netmap and the hardware rings, 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 minimal, and the checks performed make sure that any value written to the shared data structure cannot cause system crashes.

4.2.1 Blocking primitives

Blocking I/O is supported through the `select()` and `poll()` system calls. Netmap file descriptors can be passed to these functions, and are reported as ready (waking up the caller) when `avail` > 0. Before returning from a `select()/poll()`, the system updates the status of the rings, same as in the NIOC*SYNC ioctls. This way, applications spinning on an eventloop require only one system call per iteration.

4.2.2 Multi-queue interfaces

For cards with multiple ring pairs, file descriptors (and the related `ioctl()` and `poll()`) can be configured in one of two modes, chosen through the `ring_id` field in the argument of the NIOCREG `ioctl()`. In the default mode, the file descriptor controls all rings, causing the kernel to check for available buffers on any of the rings. In the alternate mode, a file descriptor is associated to a single TX/RX ring pair. This allows multiple threads/processes to create separate file descriptors, bind them to different ring pairs, and to operate independently on the card without interference or need for synchronization. Binding a thread to a specific core just requires a standard OS system call, `setaffinity()`, without the need of any new mechanism.

4.2.3 Talking to the host stack

In addition to the netmap rings associated to the hardware ring pairs, each interface in *netmap* mode exports an additional ring pair, used to handle packets to/from the host stack. A file descriptor can be associated to the "host stack" netmap rings using a special value for the `ring_id` field in the NIOCREG call. In this mode, NIOCTXSYNC encapsulates buffers into mbufs and then passes them to the host stack. Packets coming from the host stack are instead copied to the buffers in the netmap rings so that subsequent NIOCRXSYNC will report them as "received" packets.

Note that the network stack in the OS believes it has full control and access to a network interface even when this operates in netmap mode. As a consequence, it will happily try to communicate over that interface with external systems. It is then a responsibility of the netmap client to make sure that packets are properly passed between the rings connected to the host stack and those connected to the NIC. Implementing this feature is straightforward, possibly even using the zero-copy technique shown in Section 4.5. This is also an ideal opportunity to implement functions such as firewalls, traffic shapers and NAT boxes, which are normally attached to packet filter hooks.

4.3 Safety considerations

The sharing of memory between the kernel and the multiple user processes who can open `/dev/netmap` poses the question of what safety implications exist in the usage of *netmap*.

Processes using *netmap*, even if misbehaving, *cannot cause the kernel to crash*, unlike many other high-performance packet I/O systems (e.g. UIO-IXGBE, PF_RING-DNA, in-kernel Click). In fact, the shared memory area does not contain critical kernel memory regions, and buffer indexes and lengths are always validated by the kernel before being used.

A misbehaving process can however corrupt someone else's netmap rings or packet buffers. The easy cure for this problem is to implement a separate memory region for each ring, so clients cannot interfere. This is straightforward in case of hardware multiqueues, and can be simulated in software without data copies when the number of clients exceeds the number of queues. These solutions will be explored in future work.

4.4 Example of use

The real-life example below (which is the core of the packet generator used in Section 5) shows the simplicity of use of the *netmap* API. Apart from a few macros

used to navigate through the data structures in the shared memory region, netmap clients do not need any library to use the system, and the code is extremely compact and readable.

```
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(nmr.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &nmr);
p = mmap(0, nmr.memsize, fds.fd);
nifp = NETMAP_IF(p, nmr.offset);
fds.events = POLLOUT;
for (;;) {
  poll(fds, 1, -1);
  for (r = 0; r < nmr.num_queues; r++) {
    ring = NETMAP_TXRING(nifp, r);
    while (ring->avail-- > 0) {
      i = ring->cur;
      buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
      ... store the payload into buf ...
      ring->slot[i].len =  ... // set packet length
      ring->cur = NETMAP_NEXT(ring, i);
    }
  }
}
```

4.5 Zero-copy packet forwarding

Having all buffers for all interfaces in the same memory area permits zero-copy packet forwarding between interfaces in a very simple way. It suffices to swap the buffers indexes between the receive slot on the incoming interface and the transmit slot on the outgoing interface, and update the length and flags fields accordingly. The swap at the same time enqueues the packet on the output interface, and replenishes the input ring with an empty buffer without the need of resorting to a memory allocator. The relevant code is below:

```
  ...
  src = &src_nifp->slot[i]; /* locate src and dst slots */
  dst = &dst_nifp->slot[j];
  /* swap the buffers */
  tmp = dst->buf_index;
  dst->buf_index = src->buf_index;
  src->buf_index = tmp;
  /* update length and flags */
  dst->len = src->len;
  /* tell kernel to update addresses in the NIC rings */
  dst->flags = src->flags = BUF_CHANGED;
  ...
```

4.6 libpcap compatibility

An API is worth little if there are no applications that use it, and a significant obstacle to the deployment of new APIs is the need to adapt existing code to them.

Following a common approach to address compatibility problems, one of the first things we wrote on top of netmap was a small library that maps `libpcap` calls into *netmap* calls. The task was heavily simplified by the fact that *netmap* uses standard synchronization primitives, so we just needed to map the read/write functions (`pcap_dispatch()/pcap_inject()` into equivalent netmap functions. The whole code for these functions is below:

```
int pcap_inject(pcap_t *p, void *buf, size_t size)
{
  int si, idx;

  for (si = p->begin; si < p->end; si++) {
    struct netmap_ring *ring = NETMAP_TXRING(p->nifp, si);

    if (ring->avail == 0)
      continue;
    idx = ring->slot[ring->cur].buf_idx;
    bcopy(buf, NETMAP_BUF(ring, idx), size);
    ring->cur = NETMAP_RING_NEXT(ring, ring->cur);
    ring->avail--;
    return size;
  }
  return -1;
}

int pcap_dispatch(pcap_t *p, int cnt,
    pcap_handler callback, u_char *user)
{
  int si, ret = 0;

  for (si =p->begin; si < p->end; si++) {
    struct netmap_ring *ring = NETMAP_RXRING(p->nifp, si);

    while ((cnt == -1 || ret != cnt) && ring->avail > 0) {
      int i = ring->cur;
      int idx = ring->slot[i].buf_idx;

      p->hdr.len = p->hdr.caplen = ring->slot[i].len;
      callback(user, &p->hdr, NETMAP_BUF(ring, idx));
      ring->cur = NETMAP_RING_NEXT(ring, i);
      ring->avail--;
      ret++;
    }
  }
  return ret;
}
```

4.7 Implementation

In the design and development of *netmap*, a fair amount of work has been put into making the system maintainable and performant. The current version, included in FreeBSD, consists of about 2000 lines of code for system call (ioctl, select/poll) and driver support. There is no need for a userspace library: a small C header (200 lines) defines all the structures, prototypes and macros used by *netmap* clients.

To keep device drivers modifications small (a must, if we want the API to be implemented on new hardware), most of the functionalities are implemented in common code, and each driver only needs to implement two functions for the core of the NIOC*SYNC routines, one function to reinitialize the rings in netmap mode, and one function to export device driver locks to the common code. This reduces individual driver changes, mostly mechanical, to about 500 lines each, (a typical device driver has 4k .. 10k lines of code).

In the *netmap* architecture, device drivers do most of their work in the context of the userspace process. This simplifies resource management (e.g. binding processes to specific cores), and makes the system more controllable and robust, as we do not need to worry of executing too much code in non-interruptible contexts.

We generally modify drivers so that the interrupt service routine does no work except from waking up any sleeping process. This means that interrupt mitigation delays are directly passed to user processes. Synchronization between the NIC and netmap rings is done in the upper half of the system calls, and in a way that avoids expensive operations. As an example, we don't reclaim transmitted buffers or look for more incoming packets if a system call is invoked with `avail` > 0. This provides huge speedups for applications that unnecessarily invoke system calls on every packet.

Two more optimizations (pushing out any packets queued for transmission even if POLLOUT is not specified; and updating a timestamp within the netmap ring before `poll()` returns) reduce from 3 to 1 the number of system calls in each iteration of the typical event loop – once again a significant performance enhancement for certain applications.

To date we have not tried to add special performance optimizations to the code, such as aggressive use of `prefetch` instructions, or data placements to improve cache behaviour. *netmap* support is currently available for the Intel 10 Gbit/s adapters (ixgbe driver), and for various 1 Gbit/s adapters (Intel, RealTek).

5 Performance analysis

We discuss the performance of our framework by first analysing its behaviour for simple I/O functions, and then looking at more complex applications running on top of *netmap*. Before presenting our results, it is important to define the test conditions in detail.

5.1 Performance metrics

The processing of a packet involves multiple subsystems: CPU pipelines, caches, memory and I/O buses. Interesting applications are CPU-bound, so we will focus our measurements on the CPU costs. Specifically, we will measure the work (*system costs*) performed to move packets between the application and the network card. This is precisely the task that *netmap* or other packet-I/O APIs are in charge of. We can split these cycles in two components:

i) Per-byte costs are the CPU cycles consumed to move data from/to the NIC's buffers (for reading or writing a packet). This component can be equal to zero in some cases: as an example, *netmap* exports NIC buffers to the application, so it has no per-byte system costs. Other APIs, such as the socket API, impose a data copy to move traffic from/to userspace, and this has a per-byte CPU cost that, taking into account the width of memory buses and the ratio between CPU and memory bus clocks, can be in the range of 0.25 to 2 clock cycles per byte.

ii) Per-packet costs have multiple origins. At the very least, the CPU must update a slot in the NIC ring for each packet. Additionally, depending on the software architecture, each packet might require additional work, such as memory allocations, system calls, programming the NIC's registers, updating statistics and the like. In some cases, part of the operations in the second set can be removed or amortized over multiple packets.

Given that in most cases (and certainly this is true for *netmap*) *per-packet* costs are the dominating component, the most challenging situation in terms of system load is when the link is traversed by the smallest possible packets. This explains why we run most of our tests with 64 byte packets (60+4 CRC)[2].

Of course, in order to exercise the system and measure its performance we need to run some test code, but we want it to be as simple as possible in order to reduce the interference on the measurement. Our initial tests then use two very simple programs that make application costs almost negligible: a packet generator which streams pre-generated packets, similar to the one presented in Section 4.4, and a packet receiver which just counts incoming packets.

5.2 Test equipment

We have run most of our experiments on systems equipped with an i7-870 4-core CPU at 2.93GHz (3.2 GHz with turbo-boost), memory running at 1.33 GHz, and a dual port 10 Gbit/s card based on the Intel 82599 chip[3]. The operating system is FreeBSD 9/i386 as of late 2011 (other versions exhibit similar performance). Experiments have been run using directly connected cards on two similar systems. For a given version of the *netmap* software, the experimental results are highly repeatable (within 2% or less) so we do not report confidence intervals in the tables and graphs.

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. Our system can be clocked at different frequencies, taken from a discrete set of values. Nominally, most of them are multiples of 150 MHz, but we don't know how precise are the clock speeds, nor the relation between CPU and memory/bus clock speeds.

The transmit speed (in packets per second) has been measured with a packet generator similar to the one in

[2] The 10 G NIC used in our tests is unable to receive at line rate when memory writes are not multiple of 64 bytes. In the default configuration, the received CRC is not written to memory so it takes 64+4 byte packets to reach the peak rx rate of 14.20 Mpps.

[3] We have also run the tests with some 1 Gbit/s cards, but we generally reach the limits of the card (sometimes much lower than line rate) at minimum CPU speed.

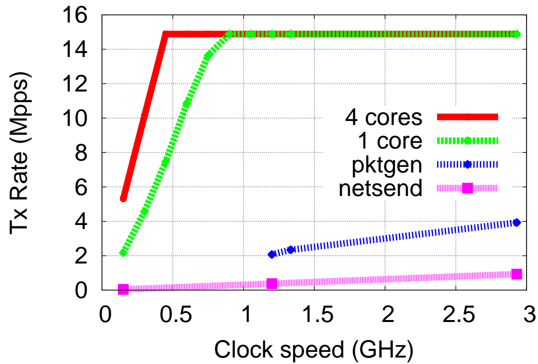

Figure 4: Transmit performance with 64-byte packets, variable clock rates and number of cores. *netmap* reaches line rate near 900 MHz. The two lines at the bottom represent `pktgen` (a specialised, in-kernel generator available on linux, peaking at about 4 Mpps at 2.93 GHz) and a `netsend` (a FreeBSD userspace generator, peaking at 790 Kpps).

Section 4.4. The packet size can be configured at runtime, as well as the number of queues and threads/cores used to send traffic. Packets are prepared in advance so that we can run the tests with close to zero per-byte costs. The test program loops around a poll(), sending at most B packets (*batch size*) per ring at each round. On the receive side we use a similar program, except that this time we poll for read events and only count packets.

5.3 Transmit speed versus clock rate

As a first experiment we ran the generator with variable clock speeds and number of cores, using a large batch size so that the system call cost is almost negligible, and throughput is maximized. *netmap* saturates the link even with just one core. By lowering the clock frequency we can determine the point where the CPU actually becomes the bottleneck for performance, and estimate the (amortized) number of cycles spent for each packet.

Figure 4 shows a subset of our results, using 4 rings (the NIC we used, as shown in Section 5.5, cannot send at line rate with 1 ring and 64-byte packets). In one of the experiments, a single thread is sending to all rings, while in another we assign each ring to a separate thread.

The data show that the throughput scales quite well with clock speed, reaching the maximum line rate near 900 MHz. This corresponds to 60-65 cycles/packet, a value which is reasonably in line with our expectations. In fact, in this particular test, the per-packet work is limited to validating the content of the slot in the netmap ring and updating the corresponding slot in the NIC ring. The cost of cache misses (which do exist, especially on the NIC ring) is amortized among all descriptors that fit into a cache line, and other costs (such as reading/writing the NIC's registers) are amortized over the entire batch.

The measurements exhibit a slightly superlinear behaviour (e.g. when doubling the clock speed between 150 and 300 MHz), though in absolute terms the deviation is very small (3-4 clock cycles per packet), and could be explained by different rations between CPU, memory and I/O bus speeds at different clock rates. Not shown by the graphs, but measured in the experiments, is the fact that once the system reaches line rate, increasing the clock speed reduces the CPU usage. The generator in fact sleeps until an interrupt from the NIC reports the availability of new buffers, and wakes up the process.

Experiments using 4 cores show a speedup of about 2.5 times over the 1-core case. The speedup is modest, but likely because the clock reduction also affects shared components (caches, bus) which are heavily exercised by these tests.

It is useful to compare the performance of our *netmap*-based generator with similar applications using conventional APIs. Figure 4 also reports the maximum throughput of two packet generators representative of the performance achievable using standard APIs. The line at the bottom represents `netsend`, a FreeBSD userspace application running on top of a raw socket. `netsend` peaks at 790 Kpps at the highest clock speed (2.93 GHz plus turbo boost), and does 390 Kpps at 1.2 GHz. The 40x difference in speed can be explained with the many additional operations that the raw socket API requires: data copies, one system call per packet, in-kernel buffer allocations, and no chance to amortize the cost of accessing the NIC's registers.

The other line in the graph is `pktgen`, an in-kernel packet generator available in Linux, which reaches almost 4 Mpps at maximum clock speed, and 2 Mpps at 1.2 GHz (the minimum speed we could set in Linux). In this case the system call and memory copy costs are removed, but there are still device driver overheads (skbuf allocations and deallocations, NIC programming) that make the system at least 7 times slower than *netmap*.

The speed vs. clock rate experiments on the receive path give results similar to the transmit section. The system can do line rate even below 1 GHz and just one receive ring, at least for packet sizes multiple of 64 bytes.

5.4 Speed versus packet size

The experiments reported so far used 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

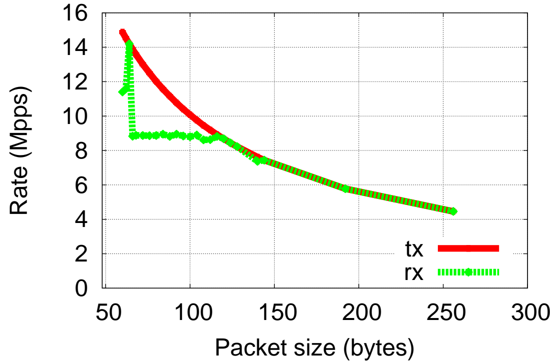

Figure 5: Actual transmit and receive speed with variable packet sizes (excluding Ethernet CRC). The top curve is the transmit rate, the bottom curve is the receive rate. See Section 5.4 for explanations.

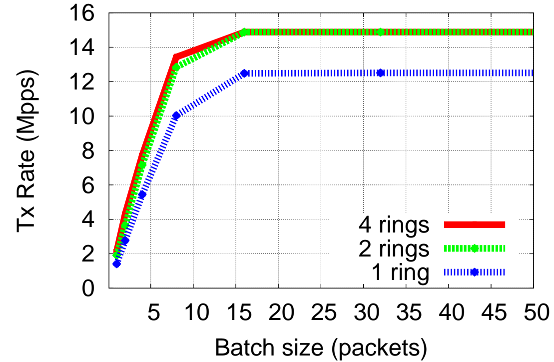

Figure 6: Transmit performance with 1 core, 2.93 GHz, 64-byte packets, and varying rings and burst size.

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 indicated 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 are 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). Investigation suggests that the NIC is issuing additional read cycles to preserve the content of remaining parts of 64-byte blocks, preventing reaching line rate. We have encountered several similar hardware bugs while testing netmap on a variety of network adapters.

5.5 Transmit speed versus batch size

Operating with large batches enhances the throughput of the system as it amortizes the cost of system calls and other potentially expensive operations. But not all applications have this luxury, and in some cases they are forced to operate in regimes where a system call is issued on each/every few packets.

We then ran another set of experiments using different batch sizes, this time running the system at minimum-size packets (64 bytes, including the Ethernet CRC). The goal of this experiment was to determine the system call overhead, and what kind of batch sizes permit reaching line rate. In this particular test we only used one core, and variable number of queues. Results are summarized in Figure 6.

As we see from the data, performance grows almost linearly with the batch size, up to the maximum value, which is approximately 12.5 Mpps with one queue, and 14.88 Mpps with 2 or more queues. The low speed achieved with a batch size of 1 shows that the cost of the `poll()` system call is much larger than the per-packet cost measured in Section 5.3, and so it is important to amortize it over large batches.

With some surprise, we found out that using a single queue (an experiment which was not done previously) we could not reach the line rate but saturated near 12.5 Mpps. Further tests with lower clock speeds reached the same maximum rate, suggesting that the bottleneck is not the CPU but must be searched in the NIC and its interface to the PCI-Express bus. This throughput limitation is not present with 2 and 4 queues, which allow the generator to reach the maximum packet rate achievable on the link.

5.6 Packet forwarding performance

The experiments described 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, as well as any data touching operation.

It is then interesting to measure the benefit of the *netmap* API when used by more CPU-intensive tasks. Packet forwarding is one of the main applications of packet processing systems, and a good test case for our framework. In fact it involves simultaneous reception and transmission (thus potentially causing memory and bus contention), and may involve some packet processing that consumes CPU cycles, and causes pipeline stalls and cache conflicts. All these phenomena will likely reduce the benefits of using a fast packet I/O mechanism, compared to the simple applications used so far.

Configuration	Mpps
netmap-fwd (1.6 GHz)	14.20
netmap-fwd + pcap	7.50
click-fwd + netmap	3.95
click-etherswitch + netmap	3.10
click-fwd + native pcap	0.49
openvswitch + netmap	3.00
openvswitch + native pcap	0.78
bsd-bridge	0.75

Figure 7: Forwarding performance of our test hardware with various software configurations.

We have then explored how a few packet forwarding applications behave when using the new API, either directly or through the `libpcap` compatibility library described in Section 4.6. The test cases are the following:

- `netmap-fwd`, a simple application that forwards packets between interfaces using the zero-copy technique shown in Section 4.5. In this case we reach line rate (14.2 Mpps, due to the limitations in the NIC) at just 1.6 GHz.
- `netmap-fwd + pcap`, as above but using the libpcap emulation (Section 4.6) instead of the zero-copy code;
- `click-fwd`, a simple Click [8] configuration shown below, that passes packets between interfaces:

```
FromDevice(ix0) -> Queue -> ToDevice(ix1)
FromDevice(ix1) -> Queue -> ToDevice(ix0)
```

  The experiment has been run using Click userspace with the system's libpcap, and on top of netmap with the libpcap emulation library;
- `click-etherswitch`, as above but replacing the two queues with an EtherSwitch element;
- `openvswitch`, the OpenvSwitch software with userspace forwarding, both with the system's libpcap and on top of netmap;
- `bsd-bridge`, in-kernel FreeBSD bridging, using the mbuf-based device driver.

Figure 7 reports the measured performance. All experiments have been run on a single core with two 10 Gbit/s interfaces, and maximum clock rate except for the first case where we saturated the link at just 1.6 GHz[4].

From the experiment we can draw a number of interesting observations:

- native netmap forwarding with no data touching operation easily reaches line rate. This is interesting because it means that full rate bidirectiona operation is possible;
- the libpcap emulation library adds a significant overhead to the previous case (7.5 Mpps at full clock vs. 14.2 Mpps at 1.6 GHz means a difference of about 80-100 ns per packet). We have not yet investigated whether/how this can be improved (e.g. by reducing stalls, optimizing the scan of rings, etc.);
- replacing the native API with the netmap-based libpcap emulation gives a speedup between 4 and 8 times for OpenvSwitch and Click, despite the fact that `pcap_inject()` does use data copies. This is also an important result because it means that real-life applications can actually benefit from our API.

[4] Previous revision of this paper reported a lower value, but we recently removed a useless and expensive function call in the critical path of packet forwarding.

5.7 Discussion

In presence of huge performance improvements such as those presented in Figure 4 and Figure 7, which show that *netmap* is 4 to 40 times faster than similar applications using the standard APIs, one might wonder about two things: i) are we doing a fair comparison, and ii) what is the contribution of the various mechanisms to the performance improvement.

The answer to the first question is that the comparison is indeed fair. All the various generators in Figure 4 do exactly the same thing and each one tries to do it in the most efficient way, constrained only by the underlying APIs they use. The answer is even more obvious for Figure 7, where in many cases we just use the same unmodified binary on top of two different `libpcap` implementations.

The results measured in different configurations also let us answer the second question – evaluate the impact of different optimizations on the *netmap*'s performance.

Data copies, as shown in Section 5.6, are moderately expensive, but they do not prevent significant speedups (such as the 7.5 Mpps achieved forwarding packets on top of libpcap+netmap). This is an interesting result, in light of the fact that we may want to reintroduce data copies to reduce the interference among netmap clients (see Section 4.3).

Per-packet system calls certainly play a major role, as witnessed by the difference between netsend and pktgen (albeit on different platforms), or by the low performance of the packet generator when using small batch sizes.

Finally, an interesting information on the cost of the skbuf/mbuf-based API comes from the comparison of pktgen (taking about 250 ns/pkt) and the netmap-based packet generator, which only takes 20-30 ns per packet which are spent in programming the NIC). These two application essentially differ only on the way packet buffers are managed, because the amortized cost of system calls and memory copies is negligible in both cases.

5.8 Application porting

We conclude with a brief discussion of the issues encountered in adapting existing applications to *netmap*. Our libpcap emulation library is a drop-in replacement for the standard one, but other performance bottlenecks in the applications may prevent the exploitation of the faster I/O subsystem that we provide. This is exactly the case we encountered with two applications, OpenvSwitch and Click (full details are described in [17]).

In the case of OpenvSwitch, the original code (with the userspace/libpcap forwarding module) had a very expensive event loop, and could only do less than 70 Kpps. Replacing the native libpcap with the netmap-based version gave almost no measurable improvement. After restructuring the event loop and splitting the system in two processes, the native performance went up to 780 Kpps, and the netmap based libpcap further raised the forwarding performance to almost 3 Mpps.

In the case of Click, the culprit was the C++ allocator, significantly more expensive than managing a private pool of fixed-size packet buffers. Replacing the memory allocator brought the forwarding performance from 1.3 Mpps to 3.95 Mpps when run over netmap, and from 0.40 to 0.495 Mpps when run on the standard libpcap.

6 Conclusions and future work

In this paper we have presented *netmap*, a framework that gives userspace applications a fast channel to exchange raw packets with the network adapter. As shown by our measurements, *netmap* can give huge performance improvements to a wide range of applications. Thanks to a libpcap compatibility library, existing applications may be able to run much faster even with no source or binary modifications. *netmap* is ideal to build packet capture and generation tools, but also useful for all packet processing tasks, including high performance software routers or switches, firewalls. Although it requires device driver modifications, *netmap* is not dependent on specific hardware features, and its design makes very reasonable assumptions on the capabilities of the NICs.

We are still exploring some aspects of the performance of *netmap*, such as using the same ideas to accelerate functions such as firewalls, IP forwarding, ACK processing, and study the interaction between *netmap* and the TCP stack. Further information on this work can be found on the project's page [15].

References

[1] The dag project. Tech. rep., University of Waikato, 2001.

[2] DERI, L. Improving passive packet capture:beyond device polling. In *SANE 2004, Amsterdam*.

[3] DERI, L. ncap: Wire-speed packet capture and transmission. In *Workshop on End-to-End Monitoring Techniques and Services* (2005), IEEE, pp. 47–55.

[4] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. Routebricks: Exploiting parallelism to scale software routers. In *ACM SOSP* (2009), pp. 15–28.

[5] HAN, S., JANG, K., PARK, K., AND MOON, S. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 195–206.

[6] HANDLEY, M., HODSON, O., AND KOHLER, E. Xorp: An open platform for network research. *ACM SIGCOMM Computer Communication Review 33*, 1 (2003), 53–57.

[7] HEYDE, A., AND STEWART, L. Using the endace dag 3.7 gf card with freebsd 7.0.

[8] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. The click modular router. *ACM Transactions on Computer Systems (TOCS) 18*, 3 (2000), 263–297.

[9] KRASNYANSKY, M. Uio-ixgbe. *Qualcomm, https://opensource.qualcomm.com/wiki/UIO-IXGBE*.

[10] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., ET AL. Netfpga–an open platform for gigabit-rate network switching and routing. *IEEE Conf. on Microelectronics Systems Education* (2007).

[11] MARIAN, T. Operating systems abstractions for software packet processing in datacenters. *PhD Dissertation, Cornell University* (2010).

[12] MCCANNE, S., AND JACOBSON, V. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter Conference* (1993), USENIX Association.

[13] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38* (March 2008), 69–74.

[14] MOGUL, J., AND RAMAKRISHNAN, K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS) 15*, 3 (1997), 217–252.

[15] RIZZO, L. Netmap home page. *Università di Pisa, http://info.iet.unipi.it/∼luigi/netmap/*.

[16] RIZZO, L. Polling versus interrupts in network device drivers. *BSDConEurope 2001* (2001).

[17] RIZZO, L., CARBONE, M., AND CATALLI, G. Transparent acceleration of software packet forwarding using netmap. *INFOCOM'12, Orlando, FL, March 2012 (to appear), http://info.iet.unipi.it/∼luigi/netmap/*.

[18] STEVENS, W., AND WRIGHT, G. *TCP/IP illustrated (vol. 2): the implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.