Revisiting Network I/O APIs: The netmap Framework

It is possible to achieve huge performance improvements in the way packet processing is done on modern operating systems.

Luigi Rizzo, Università di Pisa

Today 10-gigabit interfaces are used more and more in datacenters and servers. On these links, packets flow as fast as one every 67.2 nanoseconds, yet modern operating systems can take 10-20 times longer just to move one packet between the wire and the application. We can do much better, not with more powerful hardware but by revising architectural decisions made long ago regarding the design of device drivers and network stacks.

The netmap framework is a promising step in this direction. Thanks to a careful design and the engineering of a new packet I/O API, netmap eliminates much unnecessary overhead and moves traffic up to 40 times faster than existing operating systems. Most importantly, netmap is largely compatible with existing applications, so it can be incrementally deployed.

A BIT OF HISTORY

In current mainstream operating systems (Windows, Linux, BSD and its derivatives), the architecture of the networking code and device drivers is heavily influenced by design decisions made almost 30 years ago. At the time, memory was a scarce resource; links operated at low (by today's standards) speeds; parallel processing was an advanced research topic; and the ability to work at line rate in all possible conditions was compromised by hardware limitations in the NIC (network interface controller) even before the software was involved.

In such an environment, designers adopted a tradeoff between convenience of use, performance, and lean memory usage. Packets are represented by descriptors (named `mbuf`,[5] `skbuf`,[9] or `NDISbuffer`, depending on the operating system) linked to chains of fixed-size buffers. Mbufs and buffers are dynamically allocated from a common pool, as their lifetime exceeds the scope of individual functions. Buffers are also reference-counted, so they can be shared by multiple consumers. Ultimately, this representation of packets implements a message-passing interface among all layers of the network stack.

Mbufs contain metadata (packet size, flags, references to interfaces, sockets, credentials, and packet buffers), while the buffers contain the packet's payload. The use of fixed-size buffers simplifies memory allocators, even though it requires chaining when data exceeds the size of a single buffer. Allowing buffers to be shared can save some data copying (hence, time and space) in many common operations on the network stack. For example, when transmitting a TCP packet, the protocol stack must keep a copy of the packet in case the transmission gets lost, and sharing the buffer saves the copying cost.

The design of modern NICs is based on this data representation. They can send or receive packets split into multiple memory buffers, as implemented by the operating system. NICs use their own descriptors, much simpler than those used by the operating system, and typically arranged into a circular array, called a *NIC ring* (see figure 1). The NIC ring is statically allocated, and its slots point to the buffers that are part of the mbuf chains.

PACKET-HANDLING COSTS

Network I/O has two main cost components. The *per-byte cost* comes from data manipulation (copying, checksum computation, encryption) and is proportional to the amount of traffic processed. The *per-packet cost* comes from the manipulation of descriptors (allocation and destruction, metadata management) and the execution of system calls, interrupts, and device-driver functions. Per-packet cost depends on how the data stream is split into packets: the larger the packet, the smaller the component.

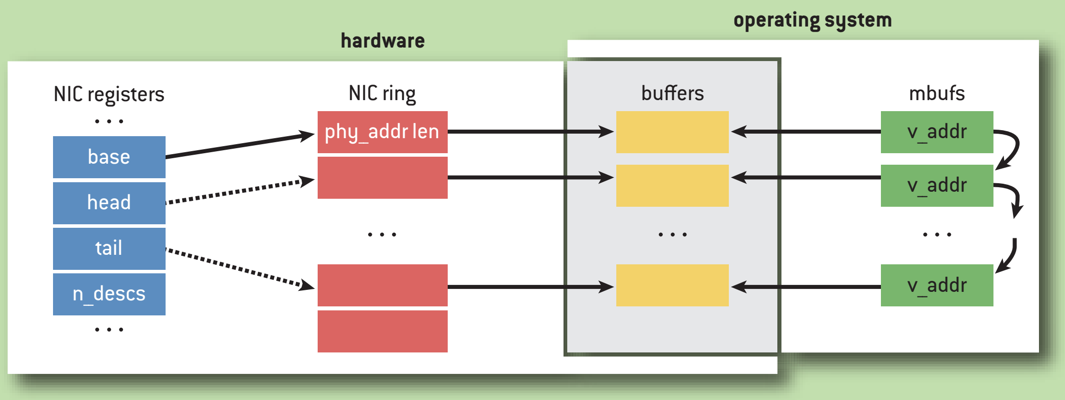
To get the idea of the speed constraints, consider a 10-Gbit/s Ethernet interface, which will be the point of reference throughout this article. The minimum packet size is 64 bytes or 512 bits, surrounded by an additional 160 bits of inter-packet gap and preambles. At 10 Gbit/s, this translates into one packet every 67.2 nanoseconds, for a worst-case rate of 14.88 Mpps (million packets per second). At the maximum Ethernet frame size (1,518 bytes plus framing), the transmission time becomes 1.23 microseconds, for a frame rate of about 812 Kpps. This is about 20 times lower than the peak rate, but still quite challenging, and it is a regimen that needs to be sustained if TCP is to saturate a 10-Gbit/s link.

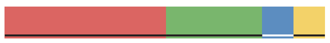
The split of the packet-processing time in a typical operating system is useful information to have in order to understand how to improve it. Consider the following two situations.

Applications using UDP (User Datagram Protocol) or raw sockets normally issue one system call per packet. This results in the allocation of an mbuf chain, which is initialized with metadata and a copy of the user-supplied payload. Then the NIC is programmed to transmit the packet, and eventually the mbuf chain is reclaimed. A similar course of action occurs on reception. Out of the total processing time, about 50 percent goes on the system call: 30 percent for mbuf allocation and reclaiming, and the remaining part equally split between memory copy and programming the hardware. A graphical representation of the split is shown here:

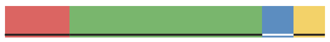
FIGURE 1

Mbufs and NIC Rings in Conventional Operating Systems





TCP senders and receivers are in a slightly different situation: system calls can move large segments from/to the kernel, and segmentation occurs within the kernel. In this case the impact of system calls on individual packets is reduced; on the other hand, the in-kernel processing is more expensive, also because of the need to handle acknowledgments flowing in the reverse direction. This results in a different split of work, shown graphically as:



The overall packet-processing capacity is similar in both cases, however: a state-of-the-art system can process approximately 1 Mpps per core. This is much less than the peak speed of a 10-Gbit/s link and barely sufficient to saturate the link with 1,500-byte packets.

Why is mbuf handling so time consuming? In the uniprocessor systems on the scene 30 years ago, memory allocators and reference counting were relatively inexpensive, especially in the presence of fixed-size objects. These days things are different: in the presence of multiple cores, allocations contend for global locks, and reference counts operate on shared memory variables; both operations may easily result in uncached DRAM accesses, which take up to 50-100 nanoseconds.[4]

DEALING WITH LATENCY

Processing costs are only one part of the equation when trying to saturate high-speed links. Latency also has a severe impact on performance. Sensitivity to latency is not unique to 10-Gbit/s networking, but the phenomenon becomes particularly significant when the link speed ceases to be the bottleneck in the system.

High memory read latencies—the tens of nanoseconds mentioned earlier—often occur when reading from the NIC's registers or NIC rings. These memory areas are updated by the NIC, thus invalidating CPU caches. To avoid stalls (which would eat all of the available packet-processing time) and achieve the desired throughput, the packet-processing code has to issue `prefetch` instructions well before data is actually needed. In turn, this might require significant code restructuring: instead of processing one packet at a time, the code should work on batches, with a first round of prefetches (to keep the CPU busy while waiting for reads to complete), followed by the actual packet processing.

Memory writes are less affected by latency because caches and write buffers can absorb many outstanding write requests without blocking. On the other hand, application stalls waiting for writes to complete may occur at the transport-protocol level. It is well known that in any communication protocol, when the amount of data in transit ("window") is limited by some maximum value W, the maximum throughput is min$(B, W/RTT)$ where B is the bottleneck bandwidth and RTT is the round-trip time of the system. Even for a very optimistic 100-microsecond RTT, it takes 1 Mbit (125 KB) of outstanding data to exploit the link's capacity. If the communicating entities do not have enough data to send before waiting for a response, achieving full speed might be impossible or require significant restructuring of the applications themselves.

HIGH-SPEED TRANSPORT PROTOCOLS

For paths with realistic RTT values (in the 10- to 100-ms range), window sizes become huge and put the entire system under pressure. Large windows interact poorly with conventional TCP congestion-control algorithms,[1] which react very conservatively to packet losses. An experimental RFC (request for comments) proposes a more aggressive window increase policy after a packet loss, thus reducing the time to reach full speed. [2] A second-order effect of the use of large windows is the increase in memory footprint and cache usage, potentially overflowing the available cache space and slowing down data access. Finally, the data structures (linear lists) normally used to store packets introduce linear costs in the presence of packet losses or reordering.

Remember that protocols sometimes deal with not only protocol headers, but also access to the payload of packets. For example, TCP checksum computation requires reading packet data (both on the transmit and receive sides), and this consumes memory bandwidth and pollutes the cache. In some cases (e.g., with locally generated data), this operation can be optimized by merging the checksum computation with the copy done to bring data into the kernel. The same is not possible for the receive path, because acknowledgments (which depend on checksum verification) cannot be delayed until the user-space process reads data. This explains why checksum computations are a natural candidate for hardware offloading.

PERFORMANCE-ENHANCEMENT TECHNIQUES

You can improve the efficiency of the packet I/O mechanisms. Enabling system calls to handle multiple packets per call amortizes their cost, which is a significant fraction of the total. Some high-performance packet-capture APIs adopt this technique. Another common option is to use packets larger than the default 1,500 bytes used on most Ethernet interfaces. A large MTU (maximum transmission unit) reduces the per-packet costs; and in the case of TCP traffic, it also speeds up the window increase process after packet losses. A large MTU is effective, however, only if it is allowed on the entire path between source and destination; otherwise, the MTU is trimmed down by path MTU discovery, or even worse, the MTU mismatch may lead to IP-level fragmentation.

Outsourcing some tasks to the hardware is another popular option to increase throughput. Typical examples are IP and TCP checksum computation and VLAN (virtual LAN) tag addition/removal. These tasks, which can be done in the NIC with only a modest amount of additional circuitry, may save some data accesses or copies. Two popular mechanisms, specifically in the context of TCP, are TSO (TCP segmentation offload) and LRO (large receive offload). TSO means that the NIC can split a single large packet into multiple MTU-size TCP segments (not IP fragments). The saving is that the protocol stack is traversed only once for the entire packet, instead of once for each MTU bytes of data. LRO acts on the receive side, merging multiple incoming segments (for the same flow) into one that is then delivered to the network stack.

Modern NICs also support some forms of packet filtering and crypto acceleration. These are more specialized features that find application in specific cases.

The usefulness of hardware acceleration has been frequently debated over the years. At every jump in link speeds (usually by a factor of 10), systems find themselves unable to cope with line rates, and vendors add hardware acceleration features to fill the gap. Then over time CPUs and memories become faster, and general-purpose processors may reach and even exceed the speed of hardware accelerators.

While there might be a point to having hardware checksums (they cost almost nothing in the hardware and can save a significant amount of time), features such as TSO and LRO are relatively inexpensive even when implemented in software.

MULTICORE SUPPORT

The almost ubiquitous availability of multiple CPU cores these days can be exploited to increase the throughput of a packet-processing system. Modern NICs support multiple transmit and receive queues, which different cores can use independently without need for coordination, at least in terms of accessing NIC registers and rings. Internally, the NIC schedules packets from the transmit queues into the output link and provides some form of demultiplexing so that incoming traffic is delivered to the receive queues according to some useful key (such as MAC addresses or 5-tuples).

NETMAP

Returning to the initial goal of improving packet-processing performance, we are looking for a large factor: from 1 Mpps to 14.88 Mpps and above, in order to reach line rate. According to Amdahl's law, such large speedups can be achieved only by subsequently shaving off the largest cost factors in the task. In this respect, none of the techniques shown so far has the potential to solve the problem. Using large packets is not always an option; hardware offloading does not help system calls and mbuf management, and recourse to parallelism is a brute-force approach to the problem and does not scale if other bottlenecks are not removed.

Netmap[7] is a novel framework that employs some known techniques to reduce packet-processing costs. Its key feature, apart from performance, is that it integrates smoothly with existing operating-system internals and applications. This makes it possible to achieve great speedups with just a limited amount of new code, while building a robust and maintainable system.

Netmap defines an API that supports sending and receiving large numbers of packets with each system call, hence making system calls (the largest cost component for the UDP) almost negligible. Mbuf handling costs (the next most important component) are completely removed because buffers and descriptors are allocated only once, when a network device is initialized. The sharing of buffers between kernel and user space in many cases saves memory copies and reduces cache pollution.

DATA STRUCTURES

Describing the netmap architecture is easier when looking at the data structures used to represent packets and support communication between applications and the kernel. The framework is built around a shared memory region—accessible to the kernel and user-space applications—which contains buffers and descriptors for all packets managed by an interface. Packet buffers have a fixed size, sufficient to store a maximum-size packet. This implies no fragmentation and a fixed and simple packet format. Descriptors—one per buffer—are extremely compact (eight bytes each) and stored in a circular array that maps one-to-one to the NIC ring. They are part of a data structure called `netmap_ring`, which also contains some additional fields, including the index (`cur`) of the first buffer to send or receive, and the number (`avail`) of buffers available for transmission or reception.

Netmap buffers and descriptors are allocated only once—when the interface is brought up—and remain bound to the interface. Netmap has a simple rule to arbitrate access to the shared data structures: the netmap ring is always owned by the application except during the execution of a system call, when the application is blocked and the operating system is free to access the structure without conflicts. The buffers between `cur` and `cur+avail` follow the same rule: they belong to user space except during a system call. Remaining buffers, if any, are instead owned by the kernel.

USER API

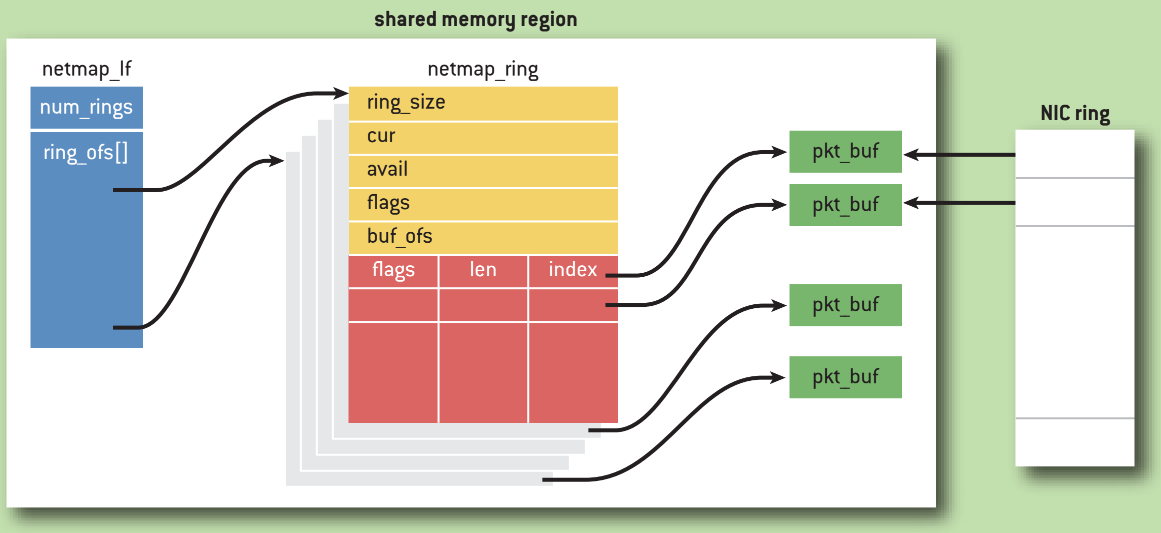
Using netmap is extremely simple and intuitive for programmers. First, an unbound file descriptor (akin to a socket) is created by calling `open("/dev/netmap")`. The descriptor is then bound to a given interface using an `ioctl()`, passing the interface name as part of the argument. On return, the argument indicates the size of the shared memory area and the position of the netmap ring(s) in that area. A subsequent `mmap()` will make the shared memory area accessible to the process.

To send packets, an application fills up to `avail` buffers with packet data, sets the `len` field in the corresponding slots in `netmap_ring`, and advances the `cur` index by the number of packets to send. After this, a nonblocking `ioctl(fd, NIOCTXSYNC)` tells the kernel to transmit the new packets and reclaims buffers for completed transmissions. The receive side uses similar operations: a nonblocking `ioctl(fd, NIOCRXSYNC)` updates the state of the netmap ring to what is known to the kernel. On return, `cur` indicates the first buffer with data, `avail` tells how many buffers are available, and data and metadata are available in the buffers and slots. The user process signals consumed packets by advancing `cur`, and at the next `ioctl()` the kernel makes these buffers available for new receptions.

Let's revisit the processing costs in a system using netmap. The system call is still there but now can be amortized over a large number of packets—possibly the entire ring—so its cost can become negligible. The next highest component, mbuf management, goes away completely because buffers and descriptors are now static. Data copies are also removed, and the only remaining operation (implemented by the `ioctl()`s) is to update the NIC ring after validating the information in `netmap_ring` and start transmission by writing to one of the NIC's registers. With these simplifications, it is not surprising that netmap can achieve much higher transmit and receive rates than the standard API.

Many applications need blocking I/O calls. Netmap descriptors can be passed to a `poll()/select()` system call, which is unblocked when the ring has slots available. Using `poll()` on a netmap file

FIGURE 2 **Shared Memory Structures and NIC Rings**



descriptor has the same complexity as the `ioctl()`s shown before, and `poll()` has some additional optimizations to reduce the number of system calls necessary in a typical application. For example, `poll()` can push out any pending transmission even if POLLOUT is not part of the argument; and it can update a timestamp in `netmap_ring`, avoiding the extra `gettimeofday()` call that many applications issue right after a `poll()`.

SUPPORT FOR MULTIPLE RINGS

Figure 2 shows that an interface can have multiple netmap rings. This supports the multi-ring architecture of modern high-speed NICs. When binding a file descriptor to a NIC, applications have a choice of attaching all rings, or just one, to the file descriptor. With the first option, the same code can work for a single- or multiqueue NIC. With the second option, a high-performance system can be built using one process/core per ring, thus exploiting the parallelism available in the system.

A working example of a traffic generator that can handle a multi-ring NIC is shown in figure 3. The device attach sequence follows the structure discussed so far. The code in charge of sending packets loops around a `poll()`, which returns when buffers are available. The code simply fills all available buffers in all rings, and the next `poll()` call will push out packets and return when more buffers are available.

HOST STACK ACCESS AND ZERO-COPY FORWARDING

In netmap mode, the NIC is disconnected from the host stack and made directly accessible to applications. The operating system, however, still believes that the NIC is present and available, so it will try to send and receive traffic from it. Netmap attaches two *software* netmap rings to the host stack, making it accessible using the netmap API. Packets generated by the host stack are extracted from the mbufs and stored in the slots of an *input* ring, similar to those used for traffic coming from the network. Packets destined to the host stack are queued by the netmap client into an *output* netmap ring, and from there encapsulated into mbufs and passed to the host stack as if they were

FIGURE 3

Code for a packet generator using netmap

```
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);
        }
    }
}
```

coming from the corresponding netmap-enabled NIC. This approach provides an ideal solution for building traffic filters: a netmap client can bind one descriptor to the host rings and one to the device rings, and decide which traffic should be forwarded between the two.

Because of the way netmap buffers are implemented (all mapped in the same shared region). building true zero-copy forwarding applications is easy. An application just needs to swap buffer pointers between the receive and transmit rings, queueing one buffer for transmission while replenishing the receive ring with a fresh buffer. This works between the NIC rings and the host rings, as well as between different interfaces.

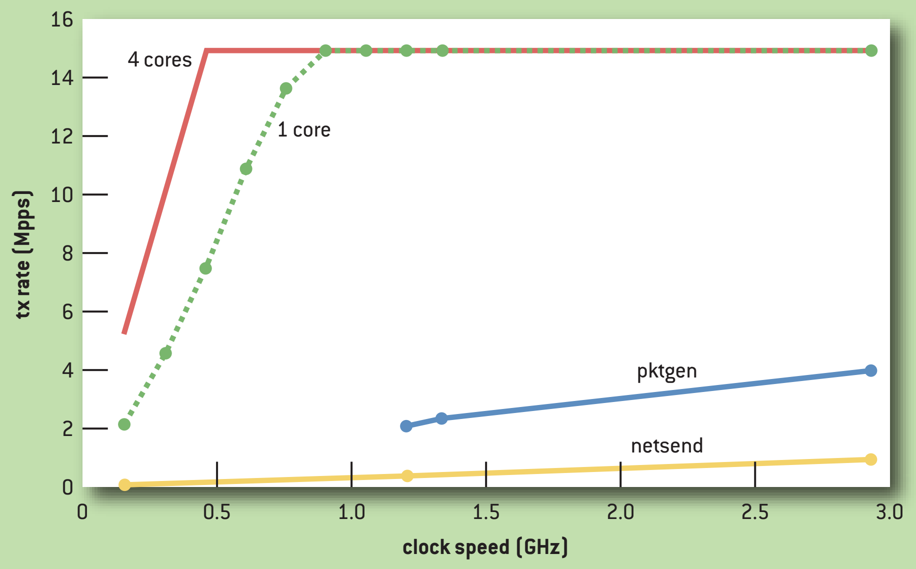
PERFORMANCE

The netmap API mainly provides a shortcut for sending and receiving raw packets. A class of applications (firewalls, traffic analyzers and generators, bridges, and routers) can easily exploit the performance benefits by using the native API. This requires changes, albeit small, in applications, which is usually undesirable. It is trivial to build a `libpcap`-compatible API on top of netmap, however, and this means that you can run a large set of unmodified applications on top of netmap.

The improvements of netmap are best seen on simple applications such as traffic generators or receivers. These applications spend the majority of their time doing raw packet I/O. Figure 4 compares the performance of various traffic generators, using netmap or standard APIs, with various clock speeds and numbers of cores. The bottom curve shows that `netsend`, a packet generator using the traditional socket API, can barely reach 1 Mpps at full clock with one core. Next on the performance graph is `pktgen`, a specialized Linux application that implements packet generation entirely within the kernel. Even in this case, the peak speed is about 4 Mpps, way below the maximum rate achievable on the 10-Gbit/s interface.

FIGURE 4

Packet Generation Speed Using Netmap vs Traditional APIs



The next two curves show the performance of the netmap generator of figure 3: not only can it reach line rate (14.88 Mpps with minimum-size frames), but also it does so even at one-third of the maximum clock speed. This is about 40 times faster than using the native API and 10 times faster than the in-kernel Linux tool. The top curve, using four cores, shows that the API scales reasonably well with multiple CPUs.

Figure 5 shows how various packet-forwarding applications benefit from the use of netmap. The two extremes are native bridging (on FreeBSD), reaching 0.69 Mpps, and a custom application that implements the simplest possible packet forwarding across interfaces using the netmap API, reaching more than 10 Mpps. As seen in the figure, the use of a `libpcap` emulation on top of netmap sacrifices only 25 percent of the performance in this case (and much less in more CPU-intensive applications). Similarly, great speedups have been achieved[8] with two popular forwarding applications. Open vSwitch[6] gets a fourfold speedup even after being heavily optimized to remove some performance issues from the original implementation. Click[3] is 10 times faster using netmap compared with the original version. In fact, Click with netmap is much faster than the in-kernel version that has been considered for many years to be one of the most efficient solutions for building software packet-processing systems.

IMPLEMENTATION

One of the key design goals of netmap is to make it easy to integrate into an existing operating system—and equally easy to maintain and port to new (or closed-source, third-party) hardware. Without these features, netmap would be just another research prototype with no hope of being used at large.

Netmap has been recently imported into FreeBSD distribution, and a Linux version is under development. The core consists of fewer than 2,000 lines of heavily documented C code, and it makes no changes to the internal data structures or software interfaces of the operating system. Netmap does require individual device-driver modifications, but these changes are small (about 500 lines of code each, compared with the 3,000 to 10,000 lines that make up a typical device driver) and partitioned so that only a small portion of the original sources is modified.

FIGURE 5

Performance of various applications with and without netmap

Packet forwarding	**Mpps**
FreeBSD bridging	0.690
netmap + libpcap emulation	7.500
netmap, native	10.660
Open vSwitch	**Mpps**
optimized, FreeBSD	0.790
optimized, FreeBSD + netmap	3.050
Click	**Mpps**
user space + libpcap	0.400
linux kernel	2.100
user space + netmap	3.950

CONCLUSIONS

Our experience with netmap has shown that it is possible to achieve huge performance improvements in the way operating systems employ packet processing. This result is possible without special hardware support or deep changes to existing software but, instead, by focusing on the bottlenecks in traditional packet-processing architectures and by revising design decisions in a way that could minimize changes to the system.

REFERENCES

1. Allman, M., Paxson, V., Blanton, E. 2009. RFC 5681: TCP congestion control.
2. Floyd, S. 2003. RFC 3649: High-speed TCP for large congestion windows.
3. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, F. 2000. The Click modular router. *ACM Transactions on Computer Systems* 18(3); http://dl.acm.org/citation.cfm?id=354874.
4. Levinthal, D. 2008-09. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors: 22; http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.
5. McKusick, M. K., Neville-Neil, G. 2004. *The Design and Implementation of the FreeBSD Operating System*. Boston, MA: Addison-Wesley.
6. Open vSwitch; http://openvswitch.org/.
7. Rizzo, L. The netmap home page; http://info.iet.unipi.it/~luigi/netmap.
8. Rizzo, L., Carbone, M., Catalli, G. 2012. Transparent acceleration of software packet forwarding using netmap. Infocom; http://info.iet.unipi.it/~luigi/netmap/20110729-rizzo-infocom.pdf.
9. Rubini, A., Corbet, J. 2001. *Linux Device Drivers*, 2nd ed. Sebastopol, CA: O'Reilly (Chapter 14); http://lwn.net/Kernel/LDD2/ch14.lwn.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

LUIGI RIZZO (rizzo@iet.unipi.it) is an associate professor at the Dipartimento di Ingegneria dell'Informazione of the Università di Pisa, Italy. His research focus is on computer networks, most recently on fast packet processing, packet scheduling, network emulation, and disk scheduling.