netmap: memory mapped access to network devices

Luigi Rizzo, *Matteo Landi*, Università di Pisa, Italy
`rizzo@iet.unipi.it - http://info.iet.unipi.it/~luigi/netmap/`

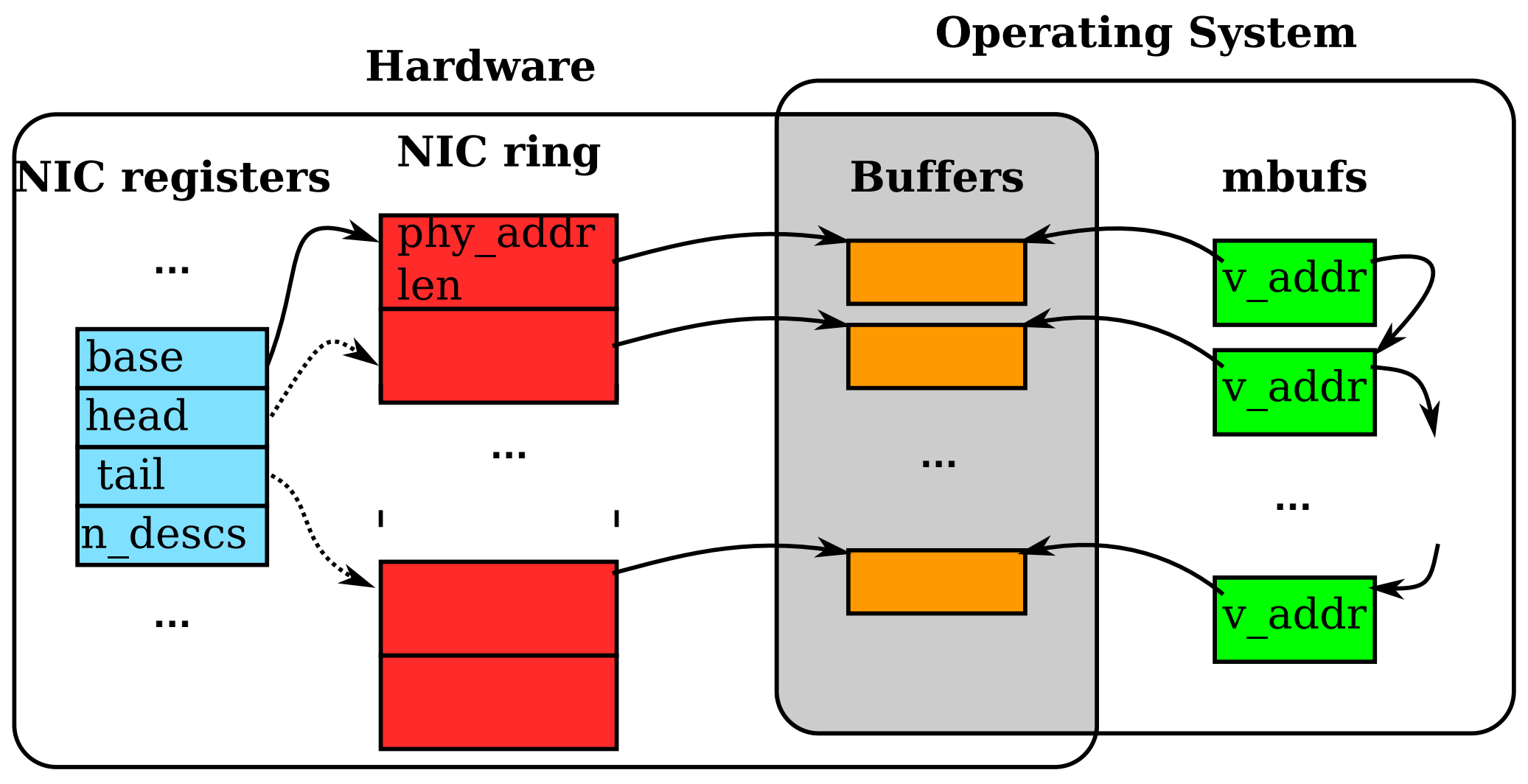
Moving packets quickly between the wire and the application is a must for systems such as software routers, switches, firewalls, traffic generators and monitors. But at 10 Gbit/s line rate equals to 14.88 Mpps per port, or 67.2 ns per packet. Sure, there are a few custom solutions [1, 2, 3, 4, 5] that run really fast. But how do we achieve such speeds (and remain compatible with applications written in the past 20 years) on general purpose OS designed when "fast" was 2-3 orders of magnitude lower ?

netmap [6] addresses the problem by making the datapath between the wire and userspace applications as fast as possible, but otherwise leaving unchanged the rest of the OS, and most/all user APIs. As a result we achieved 10..20 times faster I/O rates with minimal modifications to the operating system and applications.

Why current APIs are slow ?

Raw packet access normally uses a socket API or `libpcap`. This involves syscall and memory copy costs (often per packet) just to enter the kernel. Within the kernel, device drivers encapsulate packets into containers (mbuf/skbuf/NdisPacket) adding even more overhead due to allocations, copies, buffer sharing, synchronization.

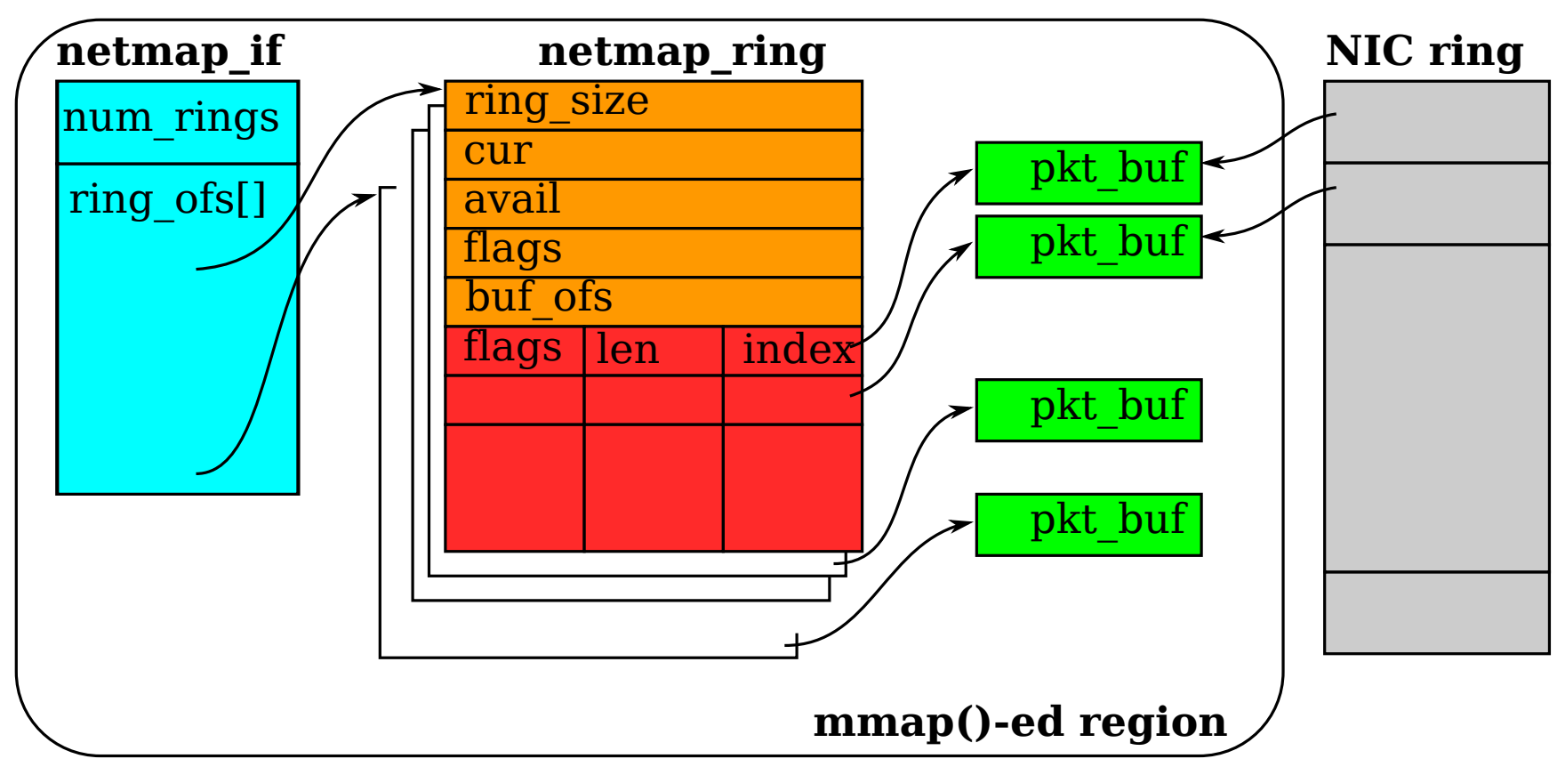
As a result, current per-core performance on standard OS is 0.5 Mpps for userspace apps, up to 1..2 Mpps for in-kernel apps, and poor scalability with number of cores. Custom systems (in-kernel Click [1]) reach about 4 Mpps, and scale slightly better with cores.

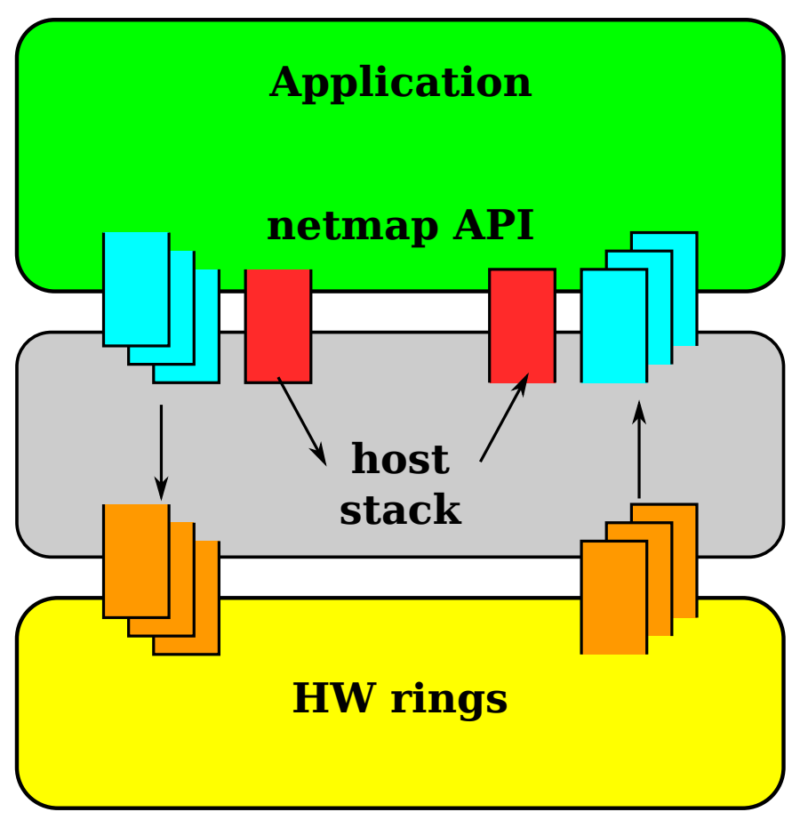


The NIC would be able to manage circular lists of buffers with little/no CPU intervention, but the OS does not make good use of these features.

netmap's key ideas

- a shadow copy of the NIC's ring (**netmap ring**) supports batching of requests and removes the need for mbufs/skbufs;
- efficient synchronization using `poll()`;
- carefully designed API, event loops need only one syscall per iteration;
- full support for multicore and multiqueue NICs through `setaffinity()`;





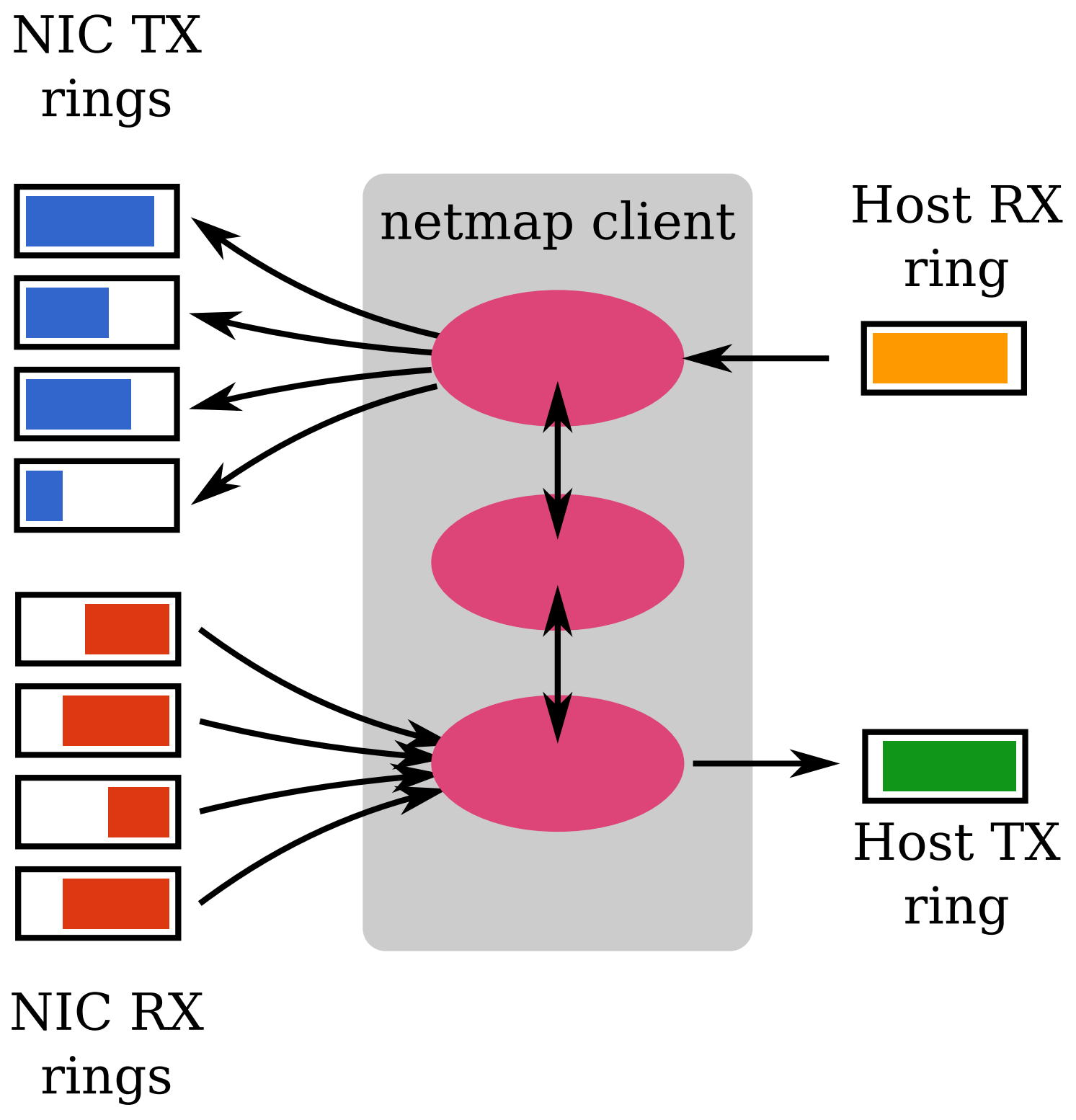
Expensive software modifications are minimized as follows:

- device independent API, does not rely on specific hw features;
- minimal, mostly mechanical modifications to existing device drivers;
- packets from/to the host stack can still use the NIC;
- we provide an efficient libpcap emulation library on top of the native API.

Most of these ideas above have been proposed before, but separately. None of the existing systems to date (though [4] comes close) puts all these features together into a high performance and general purpose framework for packet I/O from userspace. A well engineered architecture is much more than a collection of parts.

Using netmap (native API)

Threads open `/dev/netmap` and issue an `ioctl()` to switch the NIC to "netmap" mode, disconnectiong the datapath from the host stack. Data packets and netmap rings are in an `mmap()`'ed region with well defined ownership, so that lock free access is possible. Netmap rings are updated by `poll()` or `ioctl()`, and their content is validated by the kernel so a faulty program cannot crash the system. File descriptors and threads can be associated to individual rings (and cores). A thread can manage multiple interfaces and do zero-copy forwarding to other interfaces or to the host stack.

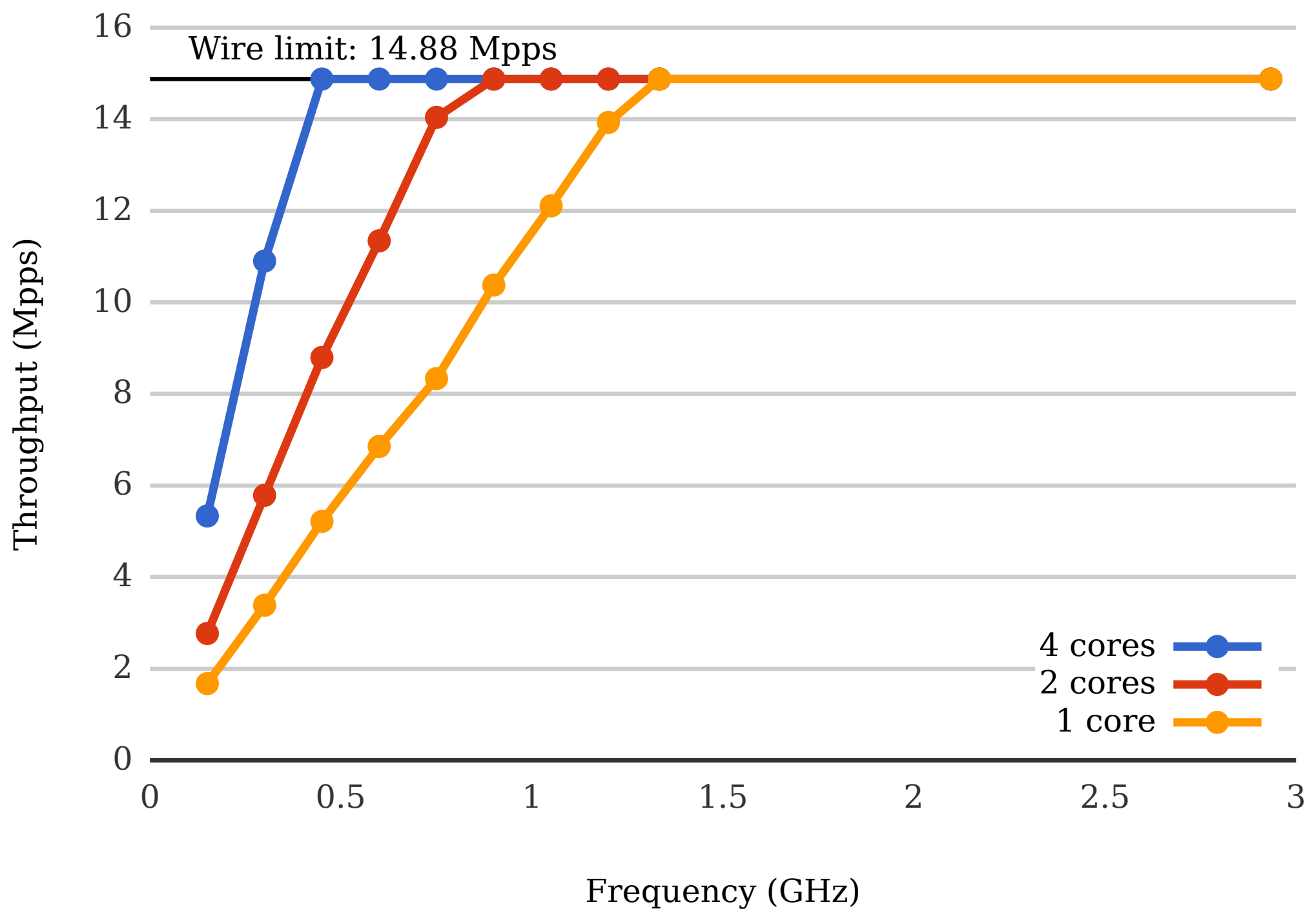


Sample code for a packet generator:

```
struct netmap_if *nifp;
struct nmreq req;
char *mem;
struct pollfd fds;

bzero(&req, sizeof(req));
bzero(&fds, sizeof(fds));
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(req.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &req);
mem = mmap(0, req.memsize, fds.fd);
nifp = NETMAP_IF(mem, req.offset);
fds.events = POLLOUT;
for (;;) {
  poll(fds, 1, -1);
  for (r = 0; r < req.num_queues; r++) {
    struct netmap_ring *ring = NETMAP_TXRING(nifp, r);
    while (ring->avail-- > 0) {
      int i = ring->cur;
      char *buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
      ... prepare packet in buf ...
      ring->slot[i].len = ... packet length ...
      ring->cur = NETMAP_NEXT(ring, i);
    }
  }
}
```

Performance



netmap uses 70..90 CPU clocks to send or receive one packet, 10-20 times less than standard APIs, and has with very good scalability on clock frequency and number of cores.
With **netmap** we do line rate packet generation at 10 Gbit/s (14.88 Mpps) with a single core running at 1.2 GHz (and low CPU occupation at higher clock rates). Packet reception is equally fast.
Simple packet forwarding runs at over 10 Mpps again using a single core.

In many cases the bottleneck is not any more the CPU, but the NIC itself (most models have limitations on RX or TX rates at some or all packet sizes) and the I/O buses. Other than that, performance usually beats that of the best in-kernel applications.
We have built a **netmapcap**, a libpcap emulation library on top of netmap so that porting applications is straightforward.

Application	Speed (Mpps)
in-kernel bridging	0.69
native netmap forwarding	10.66
netmapcap forwarding	7.50
libpcap OpenvSwitch	0.78
netmapcap OpenvSwitch	2.98
Click userspace	0.40
netmapcap Click	3.95

Applications, especially I/O intensive ones, esperience large speedups. As an example, Click userspace is now competitive or better than the in-kernel version.

netmap (available for 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. 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), more are being added.

Acknowledgements: Work supported by EC FP7-ICT Project "CHANGE" (n. 257422). Thanks to Intel Research Berkeley for funding the hardware used to run the experiments.

References

[1] E.Kohler, R.Morris, B.Chen, J.Jannotti, M.F.Kaashoek, The Click modular router, ACM TOCS, vol.18 n.3, pp.263-297, 2000

[2] M.Dobrescu, N.Egi, K.Argyraki, B.G.Chun, K.Fall, G. Iannaccone, A.Knies, M.Manesh, S.Ratnasamy, RouteBricks: Exploiting parallelism to scale software routers, ACM SOSP, 2009

[3] S. Han, K.Jang, K.Park, S.Moon, PacketShader: a GPU-accelerated software router, Proc. of ACM SIGCOMM 2010, New Delhi, India

[4] Kaist's Packet I/O engine, http://shader.kaist.edu/packetshader/io_engine/

[5] Max Krasnyansky, UIO-IXGBE, Qualcomm, https://opensource.qualcomm.com/wiki/UIO-IXGBE

[6] L. Rizzo, netmap: fast and safe access to network adapters for user programs, Tech. Report, Univ. di Pisa, June 2011, http://info.iet.unipi.it/~luigi/netmap/