netmap: Memory Mapped Access To Network Devices*

Luigi Rizzo and Matteo Landi, Università di Pisa
http://info.iet.unipi.it/~luigi/netmap/

ABSTRACT

Recent papers have shown that wire-speed packet processing is feasible in software even at 10 Gbit/s, but the result has been achieved taking direct control of the network controllers to cut down OS and device driver overheads.

In this paper we show how to achieve similar performance in safer conditions on standard operating systems. As in some other proposals, our framework, called **netmap**, maps packet buffers into the process' memory space; but unlike other proposals, any operation that may affect the state of the hardware is filtered by the OS. This protects the system from crashes induced by misbehaving programs, and simplifies the use of the API.

Our tests show that **netmap** takes less than 90 clock cycles to move one packet between the wire and the application, almost one order of magnitude less than using the standard OS path. One core at 1.33 GHz can send or receive packets at wire speed on 10 Gbit/s links (14.88 Mpps), with very good scalability in the number of cores and clock speed.

At least three factors contribute to this performance: A i) no overhead for encapsulation and metadata management; ii) no per-packet system calls and data copying; iii) much simpler device driver operation, because buffers have a plain and simple format that requires no run-time decisions.

Categories and Subject Descriptors: C.2.1 [Network Architecture and Design]: Network communications

General Terms: Design, experimentation, performance

Keywords: Software packet processing, operating systems.

1. INTRODUCTION

Moving packets quickly between the wire and the application is a must for systems such as software routers, switches, firewalls, traffic generators and monitors.

The task involves three components: Network Interface Controllers (*NICs*), device drivers, and the Operating System (*OS*). NICs typically handle packets through circular arrays of buffers (*NIC rings*) managed directly by the hardware. Device drivers wrap the buffers in containers (`mbuf, skbuf, NdisPacket`) to exchange information with the OS about data fragmentation, buffer sharing, offloading of tasks to the NIC, etc.. The OS in turn introduces further encapsulations and data copies in the path to/from the application. This layering imposes large per-packet overheads, impacting performance in many ways, including energy efficiency and the ability to work at very high packets-per-second rates.

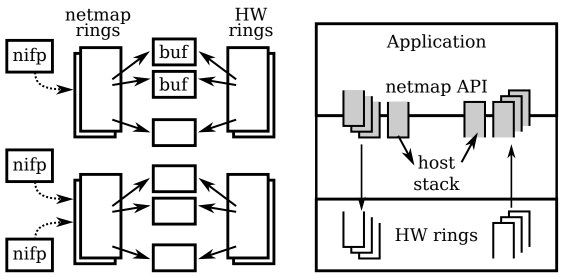

Figure 1: netmap disconnects the adapter's datapath from the host stack, and makes buffers and rings visible to applications through a shared memory area (right).

Dedicated appliances address the problem by taking direct control of the hardware, and removing unnecessary software layers. Researchers have followed a similar approach, using modified Click drivers [1, 2], or exporting packet buffers to user space [3], to reach processing speeds of millions of packets per second per core. But giving applications direct access to the hardware (even if mediated by libraries), makes the system extremely vulnerable in case of a crash of the application itself, and often prevents the use of some convenient OS primitives (e.g. select/poll).

The **netmap** framework [4] presented in this paper supports wire-speed packet processing for user-space applications in a conventional OS, while at the same time preserving the safety and convenience of a rich and portable execution environment. **netmap** is mostly meant for trusted applications in charge of special tasks (software router/switches, firewalls, traffic generator and analyzers). Existing applications can benefit of the performance improvements even without using the native **netmap** API, thanks to wrapper libraries (e.g. mapping libpcap calls onto **netmap** calls).

2. ARCHITECTURE

To achieve its performance, our framework changes the way applications talk to the NIC. A program using the **netmap** API opens the special file `/dev/netmap` and issues an `ioctl()` to switch the NIC to "netmap" mode. This *partially* disconnects the NIC from the host stack: standard

*Work supported by EC FP7-ICT Project CHANGE 257422. Thanks to Intel Research Berkeley for funding the hardware used to run the experiments.

`ioctl()`s are still operational to let the operating system configure the interface, but packets are no more exchanged with the host. Instead, they are stored into statically allocated buffers accessible in the program's address space (Fig. 1), with an `mmap()` call. Structures called *netmap rings*, also residing in the `mmap`-ed memory, replicate, in a device-independent way, the essential content of the NIC rings: size, `cur`-rent read/write position, number of `avail`-able slots (representing received packets on the RX side, empty slots on the TX side), and for each slot, the corresponding buffer index and payload size. This change, as we will see, greatly reduces the packet processing costs.

2.1 Receiving and sending packets

Packet reception is normally preceded by calls to either `ioctl(fd, NIOCRXSYNC)` (non blocking), or `poll()` (blocking). On return, these system calls update the netmap ring to reflect the status of the NIC ring. At this point the program can process some of the available packets, adjusting the `cur` and `avail` fields of the netmap ring accordingly. These values will be used in the subsequent system call to tell the kernel which buffers have been consumed. Packet data is available through the buffer indexed by the slot.

A similar mechanism is used on the transmit side, with the system calls used to wait for available buffers and/or push out newly queued packets.

Buffers and netmap rings reside in shared memory, so the system calls involve no memory copies; besides, their cost can be amortized over potentially large batches of packets.

2.2 Multiple rings, cores and NICs

Recent 1-10 Gbit cards implement multiple NIC rings to spread load to multiple CPU cores without lock contention. **netmap** supports this feature in a very natural way: NIC rings map to an equivalent number of netmap rings, which can be associated to independent file descriptors and possibly used by different processes/threads. The standard `setaffinity()` system call permits the mapping of threads to cores without any new or special mechanism.

netmap exports two more rings per NIC to communicate with the host stack. Packets from the host are made visible in an RX netmap ring; packets written to the extra TX netmap ring are encapsulated and passed to the host.

2.3 Efficient packet forwarding

All buffers and netmap rings for all adapters in netmap mode are in the same memory region, shared by the kernel and all **netmap** clients. Hence, handling multiple NICs within one process (e.g. for routing) is straightforward. Zero copy forwarding between interfaces requires no special tricks, either: swapping the buffer index between the source and destination slot enqueues the received buffer for transmission, and at the same time provides a fresh buffer on the receive ring. A similar approach can be followed to let the host stack use the NIC while in netmap mode.

2.4 Security considerations

netmap exposes the system to the same vulnerabilities as `libpcap`: programs can read all incoming packets, and inject arbitrary traffic into the network. However, they cannot possibly crash the system by writing invalid buffer indexes or packets sizes, because the content of the netmap rings is validated by the system calls before being used.

3. STATUS AND PERFORMANCE

netmap is available on FreeBSD. It consists of about 2000 lines of code for system calls and and general functions, plus individual driver modifications (mostly mechanical, about 500 lines each) currently supporting Intel 10 Gbit and 1 Gbit, and RealTek 1 Gbit adapters. The most tedious parts of the drivers (initialization, the PHY interface, ioctls etc.) do not need changes. Additionally, we have developed some native applications (packet sources and sinks, bridges) and a library to run unmodified libpcap clients on top of **netmap**, often with a 5..20x performance improvement.

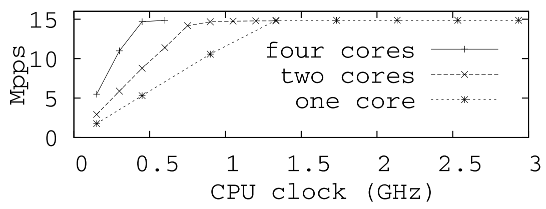

Figure 2: Send rate for a packet generator using netmap with different CPU clock speeds.

netmap dramatically reduces the per-packet overheads compared to the ordinary host stack. As an example, Fig.2 shows the send rate of a packet generator using an Intel 10G card, and an i7-870 CPU at different clock frequencies from 150 MHz to 2.93 GHz. One core achieves line rate (14.88Mpps) at just 1.3GHz, and even at 150 MHz can push out 1.76 Mpps. These numbers correspond to roughly 90 clock cycles/packet. The receive side gives similar numbers. Using 2 or 4 cores is marginally less efficient (going to 100-110 clocks/packet) probably due to memory and bus contention. As a comparison, a native traffic generator on the same system and OS delivers barely 790 Kpps per core at maximum clock speed. A more detailed performance analysis is presented in [4].

4. FUTURE WORK

In addition to porting existing applications, we are working on extending the API to support more features such as better insulation among clients (e.g. providing virtual rings in excess of those supported by the hardware) and a semi-transparent operation mode where the host stack remains connected to the NIC even in netmap mode. These and other extension are likely to expose different tradeoffs between performance and features.

5. 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, ACM, 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] 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/