netmap: a novel framework for fast packet I/O

Luigi Rizzo[*] *Università di Pisa, Italy*

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 and Linux 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 OSes 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 [14] (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 inadequate for 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 FreeBSD and Linux 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 has an amortized cost of 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,

[*]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.)

and they cannot inject bogus memory pointers in the kernel (these are often vulnerabilities of other schemes based 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.

In this paper we will focus on the architecture and features of *netmap*, and on its core performance. In a related Infocom paper [19] we address a different problem: (how) can applications make good use of a fast I/O subsystem such as netmap ? [19] shows that significant performance bottlenecks may emerge in the applications themselves, although in some cases we can remove them and make good use of the new infrastructure.

In the rest of this paper, Section 2 gives some background on current network stack architecture and performance. 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 Background

There has always been interest in using general purpose hardware and Operating Systems to run applications such as software switches [15], 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. This Section illustrates the organization of the network stack in general purpose OSes and shows the processing costs of the various stages.

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

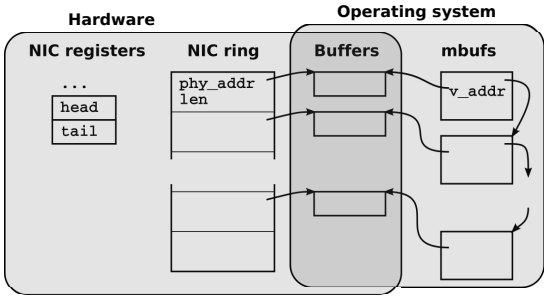

Figure 1: Typical NIC's data structures and their relation with the OS data structures.

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" [16], or inability to perform any useful work above a certain load. Polling device drivers [10, 16, 18] 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` [22] or equivalent structures such as `sk_buffs` and `NdisPackets`). These containers include large amounts of metadata about each packet: size, source or destination interface, and attributes and flags 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; 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 overhead at runtime.

These API contracts, perhaps appropriate 20-30 years ago when they were designed, are far too expensive 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 [14] device, and doing I/O through it using `send()/recv()` or specialized `ioctl()` functions.

2.3 Case study: FreeBSD sendto()

To evaluate how time is spent in the processing of a packet, we have instrumented the `sendto()` system call in FreeBSD[2] so that we can force an early return from the system call at different depths, and estimate the time spent in the various layers of the network stack. Figure 2 shows the results when a test program loops around a `sendto()` on a bound UDP socket. In the table, "time" is the average time per packet when the return point is at the beginning of the function listed on the row; "delta" is the difference between adjacent rows, and indicates the time spent at each stage of the processing chain. As an example, the userspace code takes 8 ns per iteration, entering the kernel consumes an extra 96 ns, and so on.

As we can see, we find several functions at all levels in the stack consuming a significant share of the total execution time. Any network I/O (be it through a TCP or raw socket, or a BPF writer) has to go through several expensive layers. Of course we cannot avoid the system call; the initial mbuf construction/data copy is expensive, and so are the route and header setup, and (surprisingly) the MAC header setup. Finally, it takes a long time to translate mbufs and metadata into the NIC format. Local optimizations (e.g. caching routes and headers instead of rebuilding them every time) can give modest improvements, but we need radical changes at all layers to gain the tenfold speedup necessary to work at line rate on 10 Gbit/s interfaces.

What we show in this paper is how fast can we become if we take such a radical approach, while still enforcing safety checks on user supplied data through a system call, and providing a libpcap-compatible API.

File	Function/description	time ns	delta ns
user program	`sendto` system call	8	96
uipc_syscalls.c	`sys_sendto`	104	
uipc_syscalls.c	`sendit`	111	
uipc_syscalls.c	`kern_sendit`	118	
uipc_socket.c	`sosend`	—	
uipc_socket.c	`sosend_dgram` sockbuf locking, mbuf allocation, copyin	146	137
udp_usrreq.c	`udp_send`	273	
udp_usrreq.c	`udp_output`	273	57
ip_output.c	`ip_output` route lookup, ip header setup	330	198
if_ethersubr.c	`ether_output` MAC header lookup and copy, loopback	528	162
if_ethersubr.c	`ether_output_frame`	690	
ixgbe.c	`ixgbe_mq_start`	698	
ixgbe.c	`ixgbe_mq_start_locked`	720	
ixgbe.c	`ixgbe_xmit` mbuf mangling, device programming	730	220
–	on wire	950	

Figure 2: The path and execution times for `sendto()` on a recent FreeBSD HEAD 64-bit, i7-870 at 2.93 GHz + Turboboost, Intel 10 Gbit NIC and ixgbe driver. Measurements done with a single process issuing sendto() calls. Values have a 5% tolerance and are averaged over multiple 5s tests.

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 to show how our framework can use them.

Socket APIs: The Berkeley Packet Filter, or BPF [14], 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, although usually BPF clients 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), and Ndis Miniport drivers (Windows) are in-kernel mechanisms used when packet duplication is not necessary, and instead the application (e.g. a firewall) must be interposed in the packet processing chain. These hooks intercept traffic from/to the driver and pass it to processing modules without additional data copies. The packet filter hooks rely on the standard mbuf/sk_buff based packet representation.

Direct buffer access: One easy way to remove the data copies involved in the kernel-userland transition

[2] We expect similar numbers on Linux and Windows.

is to run the application code directly within the kernel. Kernel-mode Click [10] supports this approach [4]. 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 sk_buff-based packet representation).

The kernel environment is however very constrained and fragile, so a better choice is to expose packet buffers to userspace. Examples include PF_RING [2] and Linux PACKET_MMAP, which export to userspace clients a shared memory region containing multiple pre-allocated packet buffers. The kernel is in charge of copying data between sk_buffs and the shared buffers, so that no custom device drivers are needed. This amortizes the system call costs over batches of packets, but retains the data copy and sk_buff management overhead. Possibly (but we do not have detailed documentation) this is also how the "Windows Registered I/O API" (RIO) [20] works.

Better performance can be achieved by running the full stack, down to NIC access, in userspace. This requires custom device drivers, and poses some risks because the NIC's DMA engine can write to arbitrary memory addresses (unless limited by hardware mechanisms such as IOMMUs), so a misbehaving client can potentially trash data anywhere in the system. Examples in this category include UIO-IXGBE [11], PF_RING-DNA [3], and commercial solutions including Intel's DPDK [8] and SolarFlare's OpenOnload [21].

Van Jacobson's NetChannels [9] have some similarities to our work, at least on the techniques used to accelerate performance: remove sk_bufs, avoid packet processing in interrupt handlers, and map buffers to userspace where suitable libraries implement the whole protocol processing. The only documentation available [9] shows interesting speedups, though subsequent attempts to implement the same ideas in Linux (see [13]) were considered unsatisfactory, presumably because of additional constraints introduced trying to remain 100% compatible with the existing kernel network architecture.

The PacketShader [5] I/O engine (PSIOE) is another close relative to our proposal, especially in terms of performance. PSIOE uses a custom device driver that replaces the sk_buff-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 NIC, 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 they were introduced, typical I/O buses were unable to sustain line rate at 1 and 10 Gbit/s). NetFPGA [12] 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 some work from the CPU.

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.

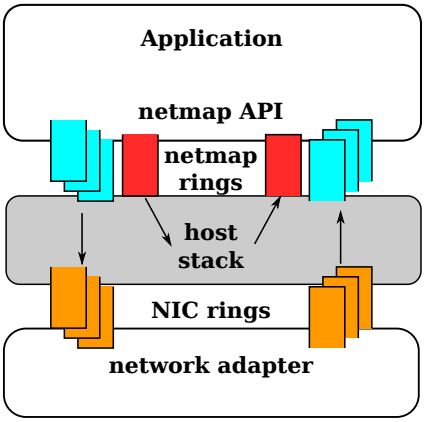

Figure 3: In netmap mode, the NIC rings are disconnected from the host network stack, and exchange packets through the netmap API. Two additional netmap rings let the application talk to the host stack.

At a very high level, when a program requests to put an interface in *netmap* mode, the NIC is partially disconnected (see Figure 3) 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.

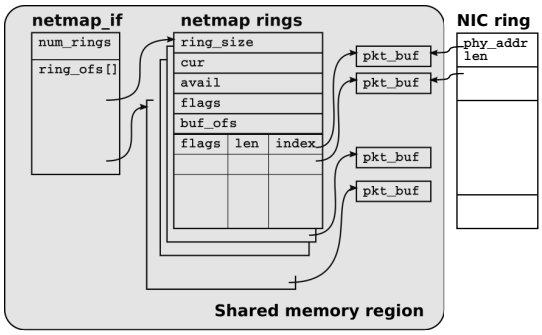

Figure 4: User view of the shared memory area exported by netmap.

4.1 Data structures

The key component in the *netmap* architecture are the data structures shown in Figure 4. They are designed to provide the 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 4: *packet buffers*, *netmap rings*, and *netmap_if* descriptors. All objects for all netmap-enabled interfaces in the system reside in the same memory region, allocated by the kernel in a non-pageable area, and shared by all user processes. The use of a single region is convenient to support zero-copy forwarding between interfaces, but it is trivial to modify the code so that different interfaces or groups of interfaces use separate memory regions, gaining better isolation between clients.

Since the shared memory 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 a 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*, that 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 them. The metadata describing the buffer (index, data length, some flags) are stored into *slots* that 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 current read or write position in the ring;
- `avail`, the number of available buffers (received packets in RX rings, empty slots in TX rings);
- `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 kernel code still in the context of the user process. Interrupt handlers and other kernel threads never touch 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 (actually, only the NIC accesses these buffers). The boundary between these two regions is 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. A subsequent `mmap()` on the file descriptor makes the memory accessible in the process' address space.

Once the file descriptor is bound to one interface and its ring(s), 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 OS about the new packets to send. 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 OS how many packets are available for reading; then their lengths and payloads are immediately available through 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:

- validates the `cur/avail` fields 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 available 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 the user-supplied data in the shared data structure do not 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 them. In the alternate mode, a file descriptor is associated to a single TX/RX ring pair. This way multiple threads/processes can create separate file descriptors, bind them to different ring pairs, and 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 Example of use

The example below (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);
    }
  }
}
```

4.3 Talking to the host stack

Even in netmap mode, the network stack in the OS is still in charge of controlling the interface (through ifconfig and other functions), and will generate (and expect) traffic to/from the interface. This traffic is handled with an additional pair of netmap rings, which can be bound to a netmap file descriptor with a NIOCREG call.

An NIOCTXSYNC on one of these rings encapsulates buffers into mbufs and then passes them to the host stack, as if they were coming from the physical interface. Packets coming from the host stack instead are queued to the "host stack" netmap ring, and made available to the netmap client on subsequent NIOCRXSYNCs.

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.

4.4 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 the framework.

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, or it can be trivially simulated in software without data copies. These solutions will be explored in future work.

4.5 Zero-copy packet forwarding

Having all buffers for all interfaces in the same memory region, zero-copy packet forwarding between interfaces only requires 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:

```
...
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;
...
```

The swap enqueues the packet on the output interface, and at the same time refills the input ring with an empty buffer without the need to involve the memory allocator.

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 calls – about 20 lines of code in total.

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. We have recently completed a Linux version, which uses the same code plus a small wrapper to map certain FreeBSD kernel functions into their Linux equivalents.

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). *netmap* support is currently available for the Intel 10 Gbit/s adapters (ixgbe driver), and for various 1 Gbit/s adapters (Intel, RealTek, nvidia).

In the *netmap* architecture, device drivers do most of their work (which boils down to synchronizing the NIC and netmap rings) in the context of the userspace process, during the execution of a system call. This improves cache locality, 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 NIC 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.

Some trivial optimizations also have huge returns in terms of performance. 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 helps 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 optimizations related to the use of `prefetch` instructions, or data placements to improve cache behaviour.

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 costs 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 sources. 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. For this reason, we run most of our tests with 64 byte packets (60+4 CRC).

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, 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 2.93 GHz (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 NIC. The numbers reported in this paper refer to the netmap version in FreeBSD HEAD/amd64 as of April 2012. Experiments have been run using directly connected cards on two similar systems. 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 it saturates a 10 Gbit/s interface even at the maximum packet rate, and we need to run the system at reduced clock speeds to determine the performance limits and the effect of code changes. Our systems can be clocked at different frequencies, taken from a discrete set of values. Nominally, most of them are multiples of 150 MHz, but we do not know how precise the clock speeds are, 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 Section 4.2.3. The packet size can be configured at runtime, as well as the number of queues and threads/cores used to send/receive 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. By lowering the clock frequency we can determine the point where the system becomes CPU bound, and estimate the (amortized) number of cycles spent for each packet.

Figure 5 show the results using 1..4 cores and an equivalent number of rings, with 64-byte packets. Throughput scales quite well with clock speed, reaching the maximum line rate near 900 MHz with 1 core. 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

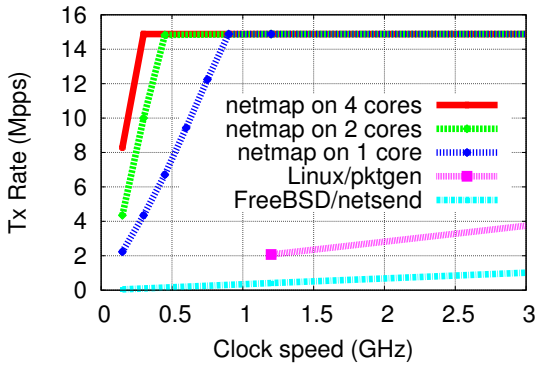

Figure 5: Netmap transmit performance with 64-byte packets, variable clock rates and number of cores, compared to `pktgen` (a specialised, in-kernel generator available on linux, peaking at about 4 Mpps) and a `netsend` (FreeBSD userspace, peaking at 1.05 Mpps).

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.

Once the system reaches line rate, increasing the clock speed reduces the total CPU usage because the generator sleeps until an interrupt from the NIC reports the availability of new buffers. The phenomenon is not linear and depends on the duration of the interrupt mitigation intervals. With one core we measured 100% CPU load at 900 MHz, 80% at 1.2 GHz and 55% at full speed.

Scaling with multiple cores is reasonably good, but the numbers are not particularly interesting because there are no significant contention points in this type of experiment, and we only had a small number of operating points (1..4 cores, 150,300, 450 Mhz) before reaching link saturation.

Just for reference, Figure 5 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 1.05 Mpps at the highest clock speed. Figure 2 details how the 950 ns/pkt are spent.

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). Here we do not have a detailed profile of how time is spent, but the similarity of the device drivers and the architecture of the application suggest that most of the cost is in the device driver itself.

The speed vs. clock results for receive are similar to the transmit ones. netmap can do line rate with 1 core at 900 MHz, at least for packet sizes multiple of 64 bytes.

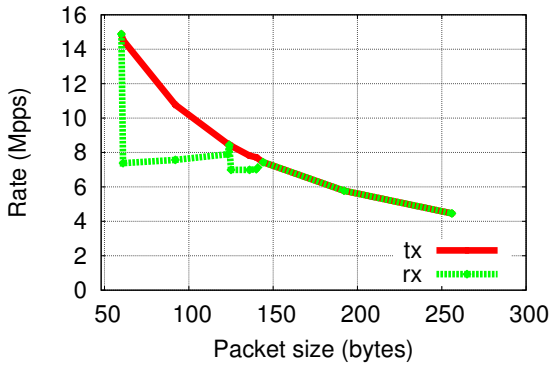

Figure 6: Actual transmit and receive speed with variable packet sizes (excluding Ethernet CRC). The top curve is the transmit rate, the bottom curve is the receive rate. See Section 5.4 for explanations.

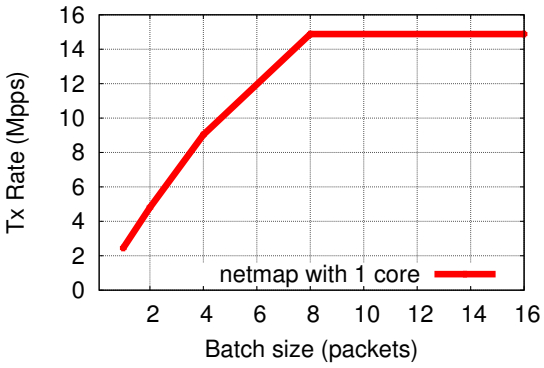

Figure 7: Transmit performance with 1 core, 2.93 GHz, 64-byte packets, and different batch size.

5.4 Speed versus packet size

The previous experiments 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 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 6. The receive side, instead, shows some surprises as indicated by the bottom curve in Figure 6. The maximum rate, irrespective of CPU speed, is achieved only for packet sizes multiples of 64 (or large enough, so that the total data rate is low). At other sizes, receive performance drops (e.g. on Intel CPUs flattens around 7.5 Mpps between 65 and 127 bytes; on AMD CPUs the value is slightly higher). Investigation suggests that the NIC and/or the I/O bridge are issuing read-modify-write cycles for writes that are not a full cache line. Changing the operating mode to remove the CRC from received packets moves the "sweet spots" by 4 bytes (i.e. 64+4, 128+4 etc. achieve line rate, others do not).

We have found that inability to achieve line rate for certain packet size and in transmit or receive mode is present in several NICs, including 1 Gbit/s ones.

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, such as accessing the NIC's registers. 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 and minimum-size packets (64 bytes, including the Ethernet CRC), trying to determine how throughput is affected by the batch size. In this particular test we only used one core, and variable number of queues.

Results are shown in Figure 7: throughput starts at about 2.45 Mpps (408 ns/pkt) with a batch size of 1, and grows quickly with the batch size, reaching line rate (14.88 Mpps) with a batch of 8 packets. The overhead of a standard FreeBSD `poll()` without calling the netmap-specific poll handlers (`netmap_poll()` and `ixgbe_txsync()`) is about 250 ns, so the handling of multiple packets per call is absolutely necessary if we want to reach line rate on 10 Gbit/s and faster interfaces.

5.6 Packet forwarding performance

So far we have measured the costs of moving packets between the wire and the application. 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.

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;
- `netmap-fwd + pcap`, as above but using the libpcap emulation instead of the zero-copy code;
- `click-fwd`, a simple Click [10] configuration 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 8 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 1.733 GHz.

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 bidirectional operation is within reach even for a single core;
- the libpcap emulation library adds a significant overhead to the previous case (7.5 Mpps at full clock vs. 14.88 Mpps at 1.733 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 using prefetching);
- replacing the system's libpcap 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.

5.7 Discussion

In presence of huge performance improvements such as those presented in Figure 5 and Figure 8, which show that *netmap* is 4 to 40 times faster than similar applications using the standard APIs, one might wonder i) how fair is the comparison, and ii) what is the contribution of the various mechanisms to the performance improvement.

Configuration	Mpps
netmap-fwd (1.733 GHz)	14.88
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 8: Forwarding performance of our test hardware with various software configurations.

The answer to the first question is that the comparison is indeed fair. All traffic generators in Figure 5 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 8, 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).

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 observation 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 [19]).

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. Click userspace is now actually faster than the in-kernel version, and the reason is the expensive device driver and sk_buff management overhead discussed in Section 2.3.

6 Conclusions and future work

We have presented *netmap*, a framework that gives userspace applications a very fast channel to exchange raw packets with the network adapter. *netmap* is not dependent on special hardware features, and its design makes very reasonable assumptions on the capabilities of the NICs. Our measurements show that *netmap* can give huge performance improvements to a wide range of applications using low/level packet I/O (packet capture and generation tools, software routers, firewalls). The existence of FreeBSD and Linux versions, the limited OS dependencies, and the availability of a libpcap emulation library makes us confident that netmap can become a useful and popular tool for the development of low level, high performance network software.

An interesting couple of open problems, and the subject of future work, are how the features of the netmap architecture can be exploited by the host transport/network stack, and how can they help in building efficient network support in virtualized platforms.

Further information on this work, including source code and updates on future developments, can be found on the project's page [17].

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. *CAIA, Tech. Rep. 080507A, May 2008.[Online]. Available: http://caia. swin. edu. au/reports/080507A/CAIA-TR-080507A. pdf* (2008).

[8] INTEL. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378* (2012).

[9] JACOBSON, V., AND FELDERMAN, B. Speeding up networking. *Linux Conference Au, http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf*.

[10] 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.

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

[12] 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).

[13] LWN.NET ARTICLE 192767. Reconsidering network channels. *http://lwn.net/Articles/192767/*.

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

[15] 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.

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

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

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

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

[20] SCHUTZ, B., BRIGGS, E., AND ERTUGAY, O. New techniques to develop low-latency network apps. *http://channel9.msdn.com/Events/BUILD/BUILD2011/SAC-593T*.

[21] SOLARFLARE. Openonload. *http://www.openonload.org/* (2008).

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