University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

High Performance Networking

Advanced Operating Systems

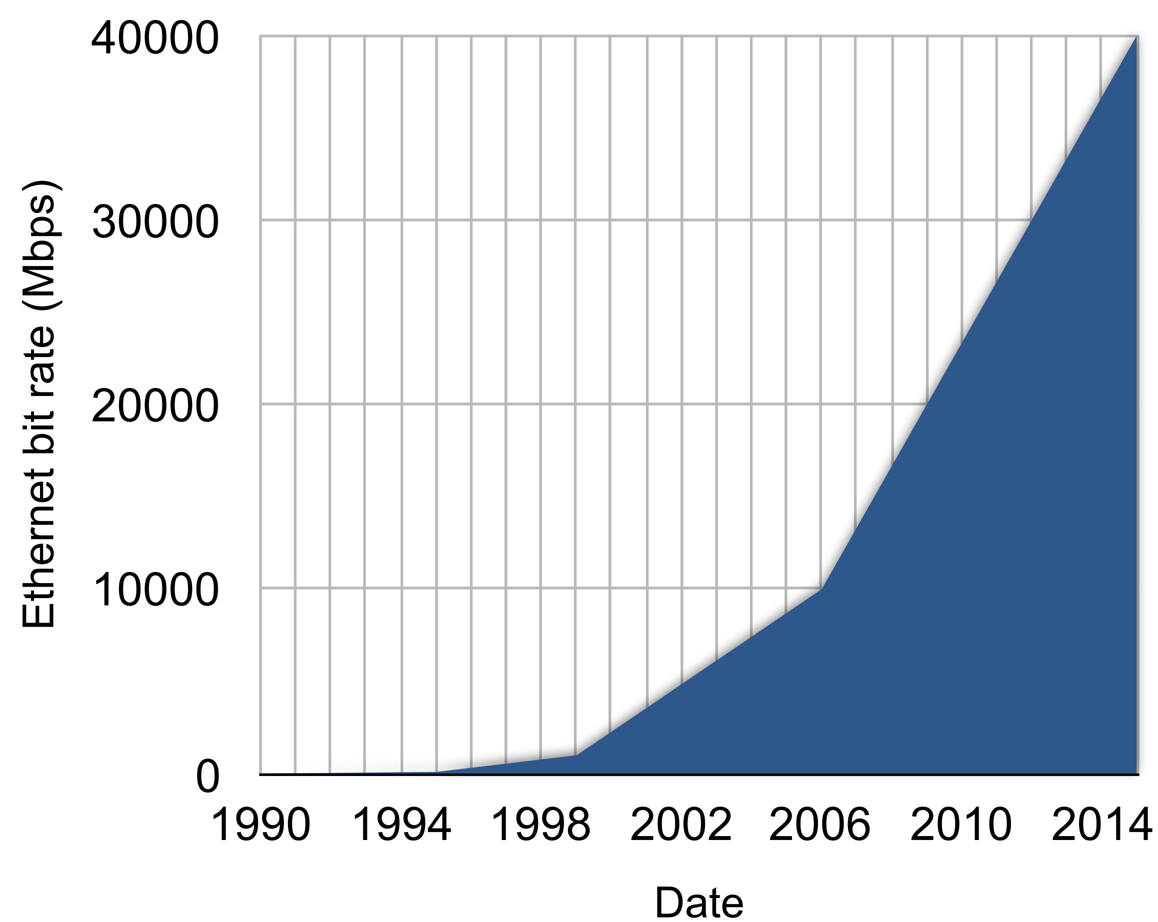
Lecture 14

Lecture Outline

- Limitations of the kernel protocol stack
- Alternative network stacks
- Accelerating TCP via API improvements
- Some cautionary remarks

Network Performance Growth

- Network performance is increasing faster than CPU performance
  - Chart shows Ethernet bit rate over time – wireless links follow a similar curve
  - Closely tracking exponential growth over time – unlike CPU speed, which stopped growing significantly mid-2000s
- MTU remains constant → packet rate increases
  - Maximum 14,880,952 packets/second on 10Gbps Ethernet (scales linearly with link rate)
  - Minimum size packet is 60 bytes data, with 8 byte preamble, 4 byte CRC; 12 byte inter-frame gap (silent period) between packets
- CPU cycles available to process each packet decreasing



Limitations of the Kernel Protocol Stack

- Why does the traditional kernel protocol stack offer sub-optimal performance?
- Designed when CPUs were faster than networks
  - Allocates memory for buffers on per-packet basis
  - Copies data multiple times, from NIC ("network interface card") to kernel to application
  - System call to send/receive each packet
    - Layered architecture offers clean design, but not efficient packet processing
    - Example on right: timing of a sendto() system call on FreeBSD: 950ns total; overheads at each layer boundary due to system call, copies, etc.
- How to redesign the protocol stack to reduce overheads?

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	

Source: L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012.

An Alternative Network Stack: netmap

- Changes network API – a mechanism to give an application dedicated control of a NIC
    - A replacement for `libpcap`, not the Sockets API
    - Basis for fast packet capture applications; software router; user-space protocol stack – not general purpose
- Pre-allocate buffers, that are shared between OS and user application; coordinate buffer ownership
    - No memory allocation at time the packets are sent/received
    - No data copies – DMA direct to application accessible memory
    - Fewer system calls – one system call can transfer ownership of multiple buffers between application and kernel
- NIC uses efficient DMA to transfer packets to and from memory; kernel manages synchronisation and memory protection
    - Kernel is the control plane
    - NIC and DMA transfer manage the data plane

netmap: a novel framework for fast packet I/O

Luigi Rizzo, *Università di Pisa, Italy*[*]
Proceedings of the 2012 USENIX Annual Technical Conference, June 2012.
https://www.usenix.org/conference/atc12/ [†]

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

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

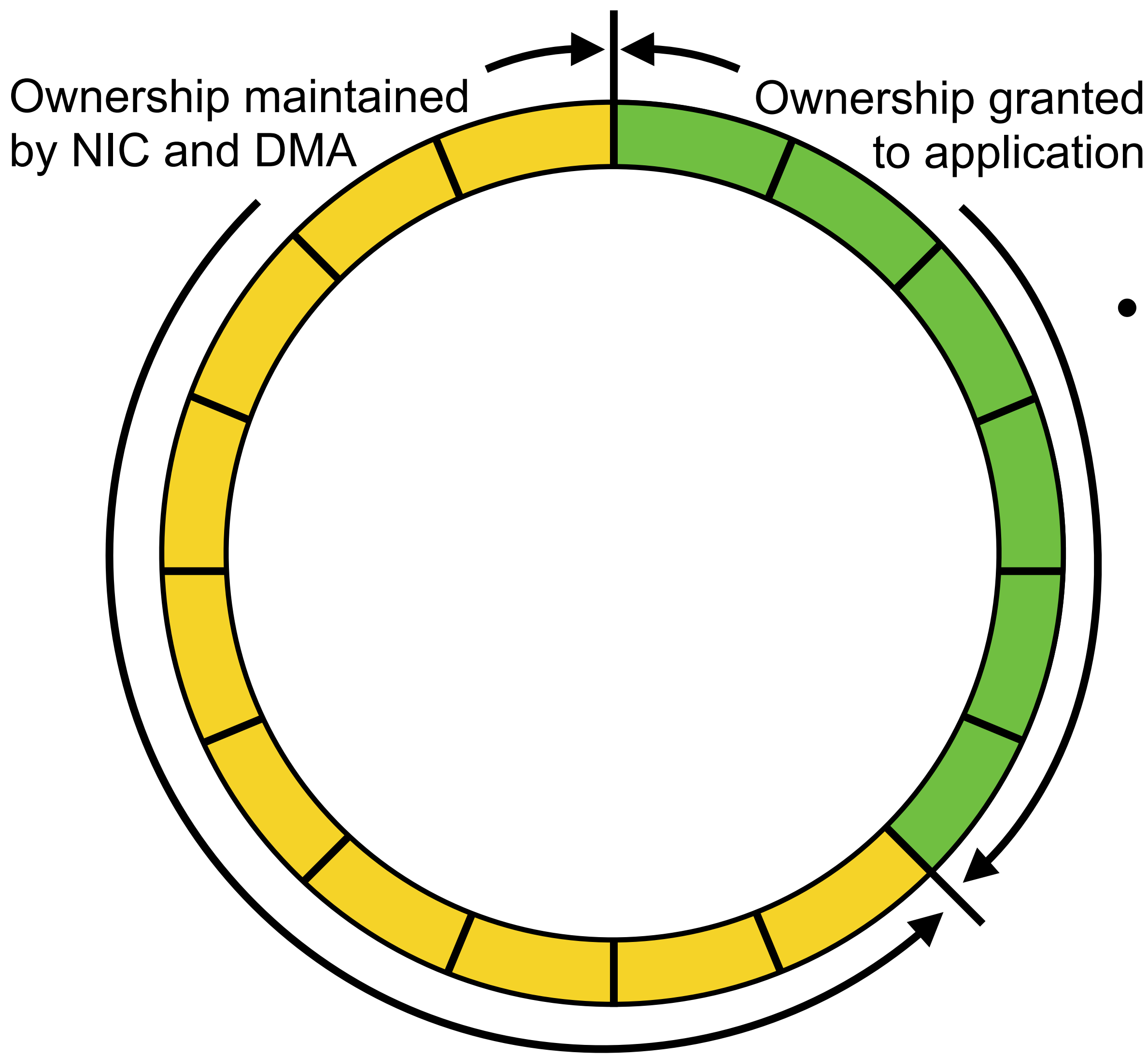
[†] USENIX plans to publish this paper on the *Proceedings of the 2012 USENIX Annual Technical Conference,* which will be available at this URL after June 13, 2012. You may also find this paper, with related material, on the author's site, http://info.iet.unipi.it/ luigi/netmap/

[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.)

1

L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo (paper and video of presentation)

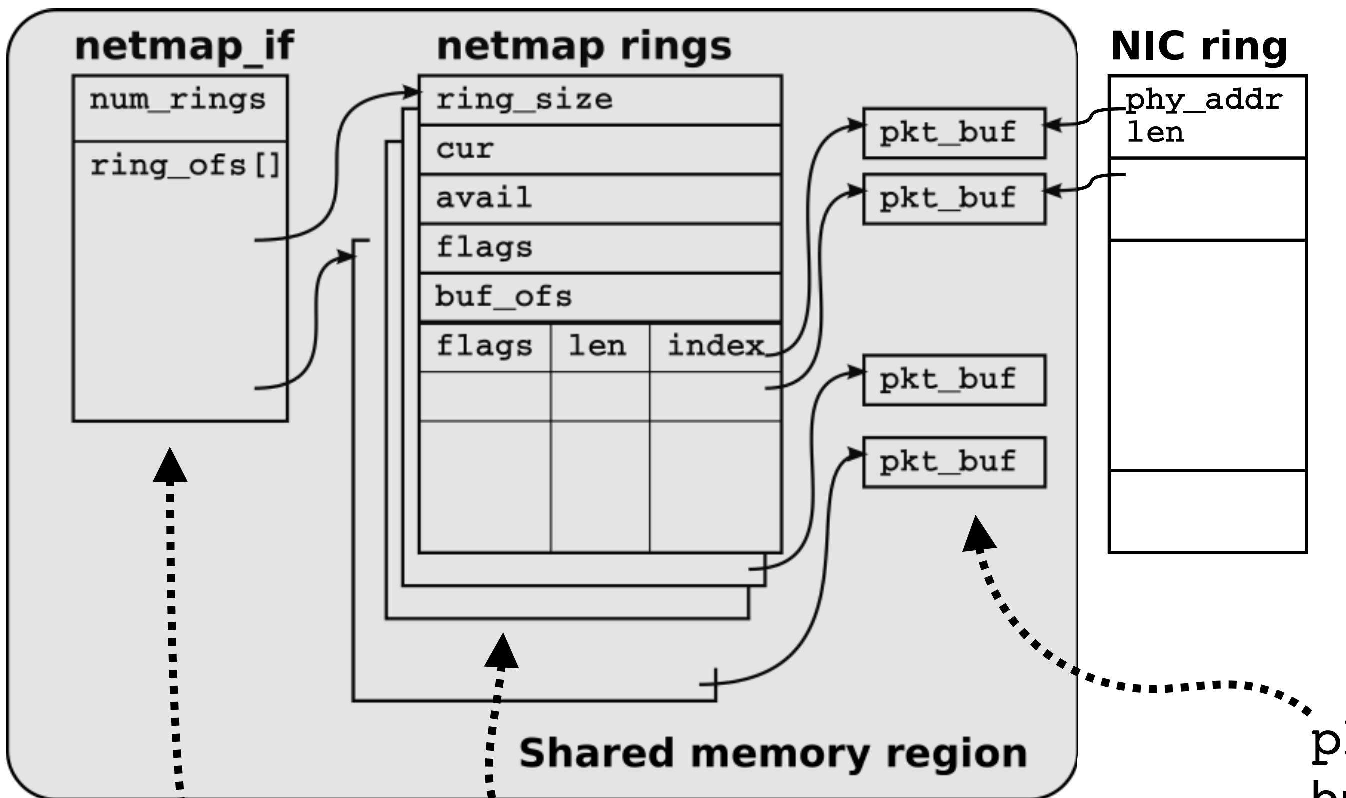
Shared Ownership of NIC Buffers (1)



- Modern NICs maintain circular buffers sized to hold queues of full size packets
  - NIC writes incoming packets direct to one segment of ring buffer via DMA
  - Operating system copies from other segment into lower layer of protocol stack – first of several copies
  - If using netmap → OS disconnected; ownership of ring buffer segment is temporarily granted to application to process packets in place

  - (Analogous for outgoing packets)

Shared Ownership of NIC Buffers (2)



Source: L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012.

`pkt_buf` objects form the circular buffer, and are in memory shared between NIC and application

`netmap_ring` provides index into the circular buffer; tracks current ownership of each `pkt_buf`

`netmap_if` provides metadata about the interface

Sample netmap Code

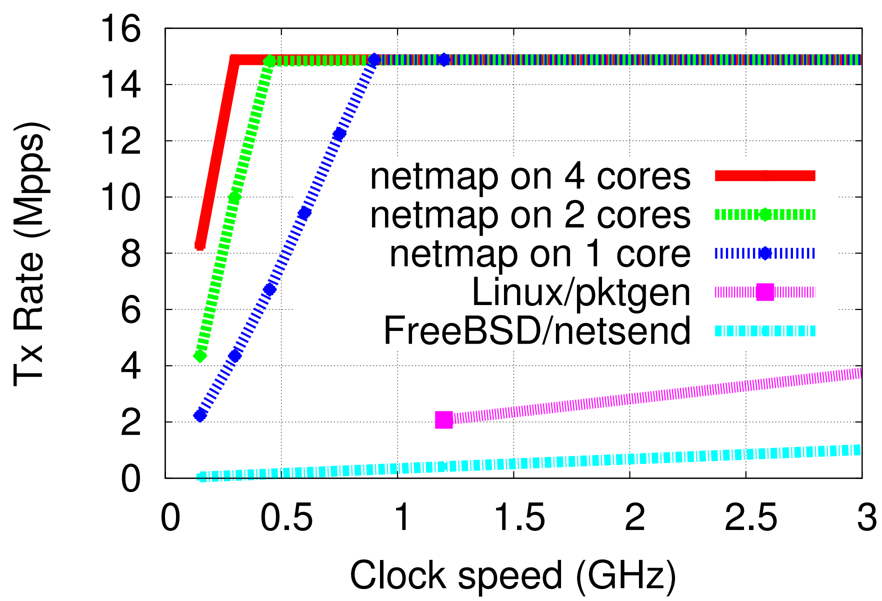
```
struct netmap_if *nifp;
struct nmreq req;
int i, len;
char *buf;

fd = open("/dev/netmap", 0);
strcpy(req.nr_name, "ix0"); // register the interface
ioctl(fd, NIOCREG, &req); // offset of the structure
mem = mmap(NULL, req.nr_memsize, PROT_READ|PROT_WRITE, 0, fd, 0);
nifp = NETMAP_IF(mem, req.nr_offset);
for (;;) {
    struct pollfd x[1];
    struct netmap_ring *ring = NETMAP_RX_RING(nifp, 0);

    x[0].fd = fd;
    x[0].events = POLLIN;
    poll(x, 1, 1000);
    for ( ; ring->avail > 0 ; ring->avail--) {
        i = ring->cur;
        buf = NETMAP_BUF(ring, i);
        use_data(buf, ring->slot[i].len);
        ring->cur = NETMAP_NEXT(ring, i);
    }
}
```

Updates `netmap_ring` structure, based on the received data (`ring->cur` and `ring->avail` only, no data copied, no synchronisation)

Gets pointer to shared `pkf_buf`

Benefits and Limitations of netmap

- Memory shared between NIC and application
  - Misbehaving applications can read memory owned by NIC – see unpredictable contents since DMA active in this region
  - Kernel data structures are protected – cannot crash the kernel or see other kernel data
- Operates on granularity of network interface
  - Application has access to all traffic on netmap interface
  - Limited applicability – not a replacement for the TCP/IP stack, but well suited to network monitoring or software router implementation
- Performance excellent – saturates 10Gbps Ethernet sending minimum size packets on 900MHz CPU
  - Using minimum size packets requires highest packet rate → highest overhead
  - Similar performance receiving packets



Source: L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012.

StackMap: Accelerating TCP with netmap

- Key insight: the TCP/IP stack processing is not expensive – inefficiencies are primarily system call overheads, copying data, and Socket API limitations with large number of file descriptors
- The netmap framework avoids the copies and reduces number of system calls, and gains in performance – but without the TCP/IP stack
- StackMap integrates the Linux kernel TCP/IP stack with netmap
  - Uses kernel TCP/IP stack for the control place
  - Uses netmap for the data plane – new API
  - Like netmap, requires a dedicated network interface for each StackMap enabled application

StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata[†1], Michio Honda[2], Douglas Santry[2], and Lars Eggert[2]
[1]*Keio University*
[2]*NetApp*

Abstract

StackMap leverages the best aspects of kernel-bypass networking into a new low-latency Linux network service based on the full-featured TCP kernel implementation, by dedicating network interfaces to applications and offering an extended version of the netmap API as a zero-copy, low-overhead *data path* while retaining the socket API for the *control path*. For small-message, transactional workloads, StackMap outperforms baseline Linux by 4 to 80 % in latency and 4 to 391 % in throughput. It also achieves comparable performance with Seastar, a highly-optimized user-level TCP/IP stack for DPDK.

1 Introduction

The TCP/IP protocols are typically implemented as part of an operating system (OS) kernel and exposed to applications through an application programming interface (API) such as the socket API [61] standard. This protects and isolates applications from one another and allows the OS to arbitrate access to network resources. Applications can focus on implementing their specific higher-level functionality and need not deal with the details of network communication.

A shared kernel implementation of TCP/IP has other advantages. The commercialization of the Internet has required continuous improvements to end-to-end data transfers. A collaboration between commercial and open source developers, researchers and IETF participants over at least the last 25 years has been improving TCP/IP to scale to increasingly diverse network characteristics [11, 39, 58], growing traffic volumes [13, 32], and improved tolerance to throughput fluctuations and reduced transmission latencies [1, 10, 49].

A modern TCP/IP stack is consequently a complex, highly optimized and analyzed piece of software. Due to these complexities, only a small number of stacks (*e.g.*, Linux, Windows, Apple, BSD) have a competitive feature set and performance, and therefore push the vast majority of traffic. Because of this relatively small number of OS stacks (compared to the number of applications), TCP/IP improvements have a well-understood and relatively easy deployment path via kernel updates, without the need to change applications.

However, implementing TCP/IP in the kernel also has downsides, which are becoming more pronounced with larger network capacities and applications that are more sensitive to latency and jitter. Kernel data processing and queueing delays now dominate end-to-end latencies, particularly over uncongested network paths. For example, the fabric latency across a datacenter network is typically only a few µs. But a minimal HTTP transaction over the same fabric, consisting of a short "GET" request and an "OK" reply, takes tens to hundreds of µs (see Section 3).

Several recent proposals attempt to avoid these overheads in a radical fashion: they bypass the kernel stack and instead implement all TCP/IP processing inside the application in user space [24, 29, 37] or in a virtual machine context [4]. Although successful in avoiding overheads, these kernel-bypass proposals also do away with many of the benefits of a shared TCP/IP implementation: They usually implement a simplistic flavor of TCP/IP that does not include many of the performance optimizations of the OS stacks, it is unclear if and by whom future protocol improvements would be implemented and deployed, and the different TCP/IP versions used by different applications may negatively impact one another in the network.

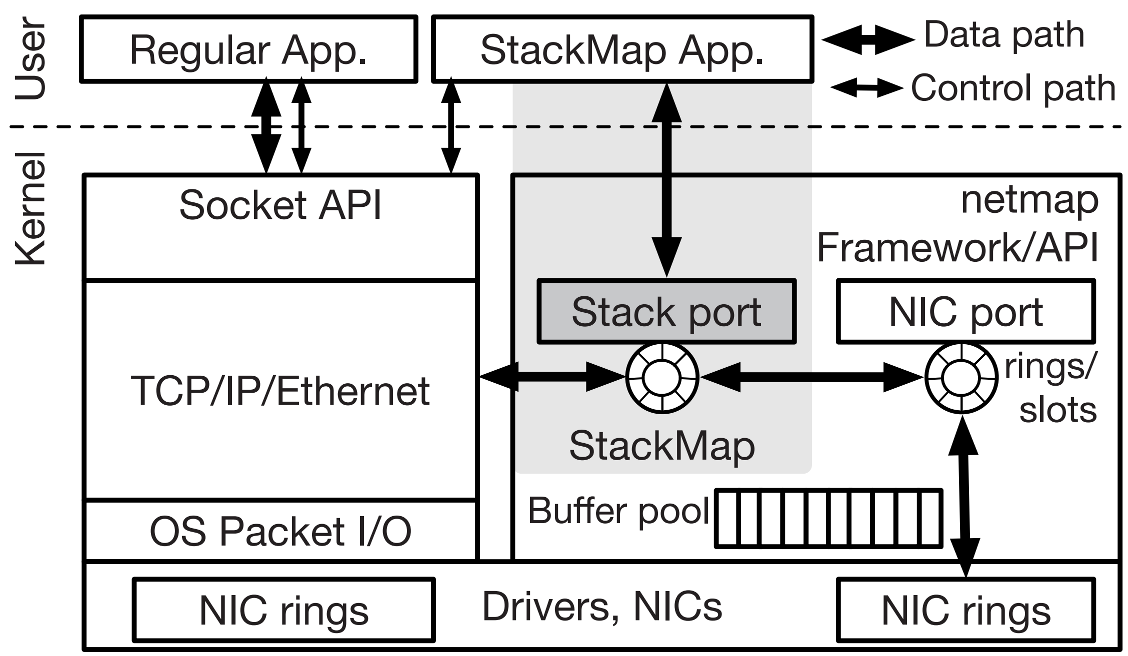
It is questionable whether kernel-bypass approaches are suitable even for highly specialized network environments such as datacenters. Due to economic reasons [17], they are assembled from commodity switches and do not feature a centralized flow scheduler [2, 45]. Therefore, path characteristics in such datacenters vary, and more advanced TCP protocol features may be useful in order to guarantee sub-millisecond flow completion times.

[†]Most of the research was done during an internship at NetApp.

USENIX Association 2016 USENIX Annual Technical Conference 43

K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

StackMap Architecture



Source: K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016.

- Socket API for control: `socket()`, `bind()`, `listen()`, `accept()`, etc.
- Netmap API used for data
  - `STACKMAP_BUF()` updates netmap's circular buffer, and passes data through the TCP/IP stack for processing
- StackMap manages the buffer pool
  - Circular buffer used by netmap
  - If packet must be stored for retransmission, its buffer is swapped out of the netmap ring and replaced by another from the pool, until needed
    - Zero copy – just swaps pointers; buffer is already shared between netmap, kernel, and application
  - Also manages *scratchpad* data structure, to simplify iteration over data from on multiple connections

StackMap API

```
struct sockaddr_in sin = { AF_INET, "10.0.0.1", INADDR_ANY };
int sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(sd, &sin);
// prefix "stack" opens stack port for given interface
struct nm_desc *nmd = nm_open("stack:ix0");
connect(sd, dst_addr); /* non-blocking */
// transmit using ring 0 only, for this example
struct netmap_ring *ring = NETMAP_TXRING(nmd->nifp, 0);
uint32_t cur = ring->cur;
while (app_has_data && cur != ring->tail) {
    struct netmap_slot *slot = &ring->slot[cur];
    char *buf = STACKMAP_BUF(ring, slot->buf_index);
    // place payload in buf, then
    slot->fd = sd;
    cur = nm_ring_next(ring, cur);
}
ring->head = ring->cur = cur;
ioctl(nmd->fd, NIOCTXSYNC);
```

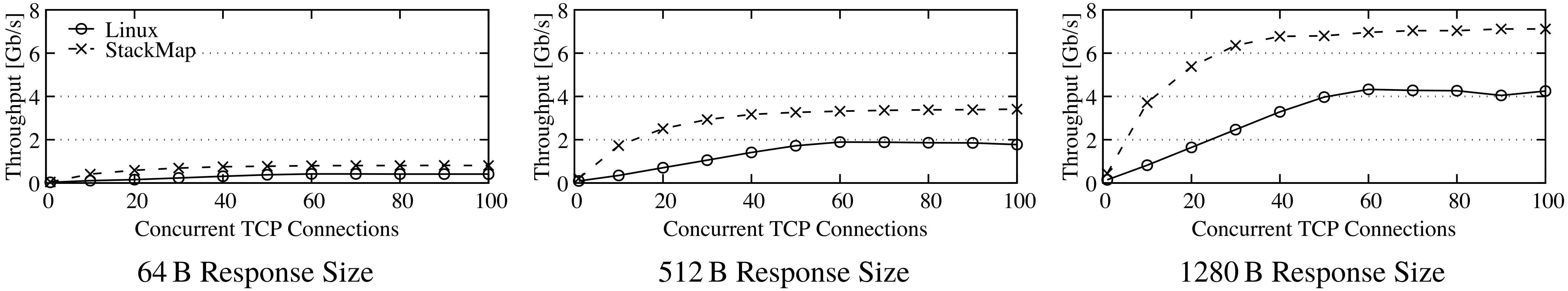
Sockets API used to initiate connection

The netmap API is used to send and receive data

The `STACKMAP_BUF()` call is an extension that passes the data to the kernel TCP/IP stack, handles ACKs, retransmission, congestion control, etc.

Source: K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016.

StackMap Performance



- Linux with StackMap extensions outperforms standard Linux
- Primary benefits are avoiding copies and better scaling with concurrent flows: performance benefit increases with response size and number of connections
- Note: uses the full kernel TCP/IP stack – not a cut-down version – benefits due to new API, not reduced functionality

Accelerating TCP: Other Approaches

- Numerous other attempts to accelerate TCP/IP stack processing exist:
  - Example: sandstorm builds on netmap; combines application, TCP/IP, and ethernet processing into a user-space library – builds highly optimised, special purpose, protocol stack for each application (see “Network Stack Specialization for Performance”)
  - Example: Google’s QUIC protocol aims to provide an alternative to TCP, with better performance, in a user space protocol running over UDP
  - …

- No clear consensus on the right approach
  - API changes to batch packet processing and reduce copies are clearly beneficial – how general purpose the resulting API needs to be is unclear
  - Pervasive encryption may reduce the benefits – you need to make a copy while decrypting, and many of these approaches benefit from zero-copy stacks

I. Marinos, R. N. M. Watson, and M. Handley. Network stack specialization for performance. Proceedings of the SIGCOMM Conference, Chicago, IL, USA, August 2014. ACM. DOI: 10.1145/2619239.2626311 (Open access via http://dl.acm.org/authorize?N71202)

Conclusions

- Improvements to network performance relative to CPU performance highlight limitations of Sockets API for high performance – ongoing work to find good new APIs
- Beware: it's very hard to beat TCP – incredibly well tuned, secure, and well maintained
  - TCP highly optimised over many years – handles edge cases in security and congestion control that are non-obvious
    - IETF RFC 7414 ("A Roadmap for TCP Specification Documents") is 55 pages, and references 150 other documents – textbook explanations of TCP omit much important detail
  - Likely better to optimise kernel TCP stack and API, than end up stuck on a poorly maintained user-space stack

Further Reading

netmap: a novel framework for fast packet I/O

Luigi Rizzo, *Università di Pisa, Italy**
Proceedings of the 2012 USENIX Annual Technical Conference, June 2012.
`https://www.usenix.org/conference/atc12/` †

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

*This work was funded by the EU FP7 project CHANGE (257422).
†USENIX plans to publish this paper on the *Proceedings of the 2012 USENIX Annual Technical Conference*, which will be available at this URL after June 13, 2012. You may also find this paper, with related material, on the author's site, `http://info.iet.unipi.it/ luigi/netmap/`

[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.)

1

StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata†1, Michio Honda[2], Douglas Santry[2], and Lars Eggert[2]
[1]*Keio University*
[2]*NetApp*

Abstract

StackMap leverages the best aspects of kernel-bypass networking into a new low-latency Linux network service based on the full-featured TCP kernel implementation, by dedicating network interfaces to applications and offering an extended version of the netmap API as a zero-copy, low-overhead *data path* while retaining the socket API for the *control path*. For small-message, transactional workloads, StackMap outperforms baseline Linux by 4 to 80 % in latency and 4 to 391 % in throughput. It also achieves comparable performance with Seastar, a highly-optimized user-level TCP/IP stack for DPDK.

1 Introduction

The TCP/IP protocols are typically implemented as part of an operating system (OS) kernel and exposed to applications through an application programming interface (API) such as the socket API [61] standard. This protects and isolates applications from one another and allows the OS to arbitrate access to network resources. Applications can focus on implementing their specific higher-level functionality and need not deal with the details of network communication.

A shared kernel implementation of TCP/IP has other advantages. The commercialization of the Internet has required continuous improvements to end-to-end data transfers. A collaboration between commercial and open source developers, researchers and IETF participants over at least the last 25 years has been improving TCP/IP to scale to increasingly diverse network characteristics [11, 39, 58], growing traffic volumes [13, 32], and improved tolerance to throughput fluctuations and reduced transmission latencies [1, 10, 49].

A modern TCP/IP stack is consequently a complex, highly optimized and analyzed piece of software. Due to these complexities, only a small number of stacks (*e.g.*, Linux, Windows, Apple, BSD) have a competitive feature set and performance, and therefore push the vast majority of traffic. Because of this relatively small number of OS stacks (compared to the number of applications), TCP/IP improvements have a well-understood and relatively easy deployment path via kernel updates, without the need to change applications.

However, implementing TCP/IP in the kernel also has downsides, which are becoming more pronounced with larger network capacities and applications that are more sensitive to latency and jitter. Kernel data processing and queueing delays now dominate end-to-end latencies, particularly over uncongested network paths. For example, the fabric latency across a datacenter network is typically only a few µs. But a minimal HTTP transaction over the same fabric, consisting of a short "GET" request and an "OK" reply, takes tens to hundreds of µs (see Section 3).

Several recent proposals attempt to avoid these overheads in a radical fashion: they bypass the kernel stack and instead implement all TCP/IP processing inside the application in user space [24, 29, 37] or in a virtual machine context [4]. Although successful in avoiding overheads, these kernel-bypass proposals also do away with many of the benefits of a shared TCP/IP implementation: They usually implement a simplistic flavor of TCP/IP that does not include many of the performance optimizations of the OS stacks, it is unclear if and by whom future protocol improvements would be implemented and deployed, and the different TCP/IP versions used by different applications may negatively impact one another in the network.

It is questionable whether kernel-bypass approaches are suitable even for highly specialized network environments such as datacenters. Due to economic reasons [17], they are assembled from commodity switches and do not feature a centralized flow scheduler [2, 45]. Therefore, path characteristics in such datacenters vary, and more advanced TCP protocol features may be useful in order to guarantee sub-millisecond flow completion times.

†Most of the research was done during an internship at NetApp.

USENIX Association 2016 USENIX Annual Technical Conference 43

L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo (paper and video of presentation)

K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata