Advanced Network Programming (ANP)
XB_0048

Userspace Networking Stacks

Animesh Trivedi
Autumn 2020, Period 1

VRIJE UNIVERSITEIT AMSTERDAM

Layout of upcoming lectures - Part 1

Sep 1st, 2020 (today): ~~*Introduction and networking concepts*~~

Sep 3rd, 2020 (this Tuesday): ~~*Networking concepts (continued)*~~

Sep 8th, 2020 : ~~*Linux networking internals*~~

Sep 10th 2020: ~~*Multicore scalability*~~

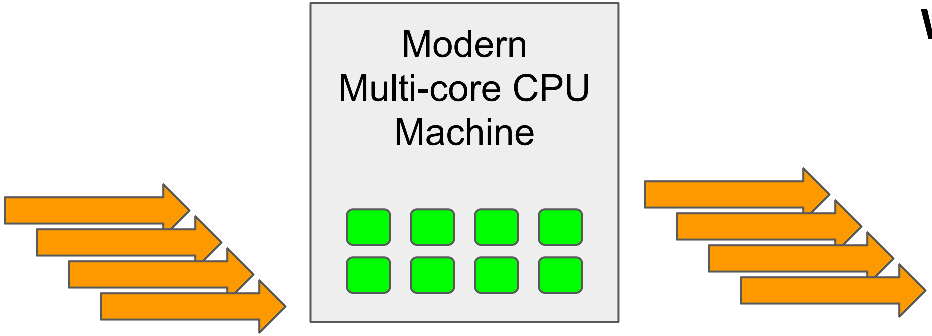
Sep 15th 2020: *Userspace networking stacks*

Sep 17th 2020: *Introduction to RDMA networking*

Multicore challenges
Interrupt load balancing
RSS and friends
SMP and NUMA
MegaPipe

Packet Processing Frameworks

What is packet processing? Lots of applications such as firewall, routers, forwarding, traffic generators, and middlewares that process and work on raw network packets - they are middleman



10 Gbps = 14.8 Mpps
100 Gbps = 148.8 Mpps

Why? use multi-core servers...

- High-end switches are expensive
- Have you seen their OSes?
  - Mostly a hard-to-use systems
  - Cumulus OS (exception)
- Not flexible (ASIC)
- You have a new protocol, or aggregator, or application-level hook? Forget about it

Alternative: use Linux/servers - simple !

Packet Processing Frameworks

What is packet processing? Lots of applications such as firewall, routers, forwarding, traffic generators, and middlewares that process and work on raw network packets - they are middleman

Basic packet processing, and small, short-lived connections (as we saw in Megapipe) have a lot in common :

- Less bandwidth-heavy, but more "***volume***" driven
- Small payloads
- Stress on per-packet processing cost

How many packets can one process per second, and with what resources?

- Alternatively: if you cannot process packets fast, you cannot do TCP/IP processing faster

Netmap: A Novel Framework for fast packet I/O

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.

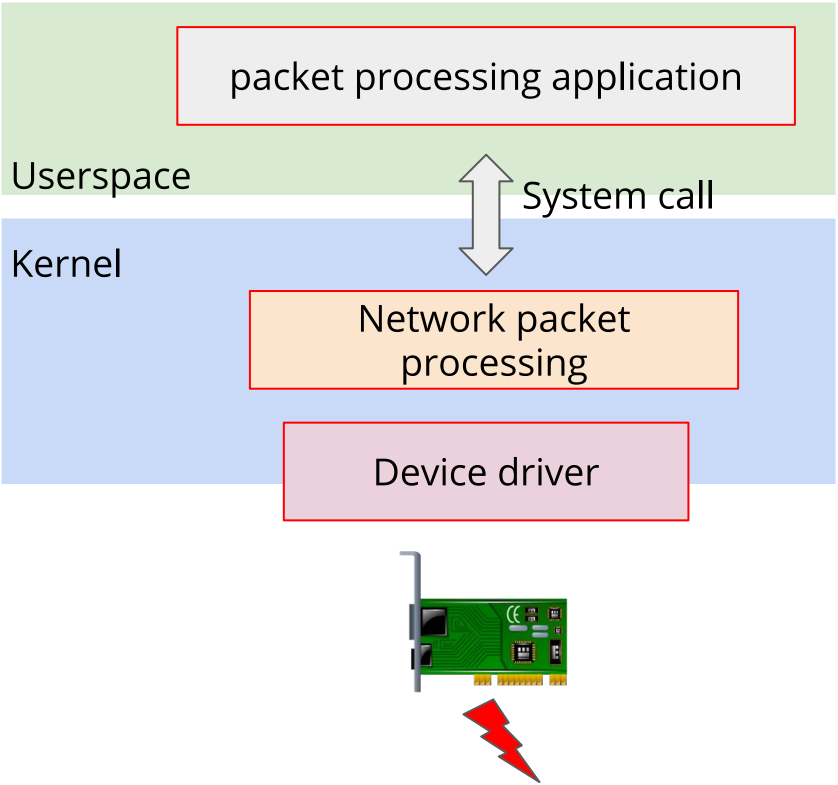
In building *netmap*, we identified and successfully reduced or removed three main packet processing costs: per-packet dynamic memory allocations, removed by preallocating resources; system call overheads, amortized over large batches; and memory copies, eliminated by sharing buffers and metadata between 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

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

https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf (won the best paper award!)

The key problem - Getting Fast Access to Packets



Typical options:

1. Raw sockets (AF_PACKETS)
    a. High overheads, packet copies, per packet system call
2. Packet filter hooks (BPF)
    a. Complex, in kernel, limited changes
3. Direct buffer access
    a. Run in kernel
    b. PF_RING : data copies and shared metadata overheads

No high-performance, safe, flexible way of getting access to raw packets

The root cause of high overheads - I

A single packet is defined by `struct sk_buf` inside Linux (or `mbuf` in BSD, MS Windows I don't know)

These structures are

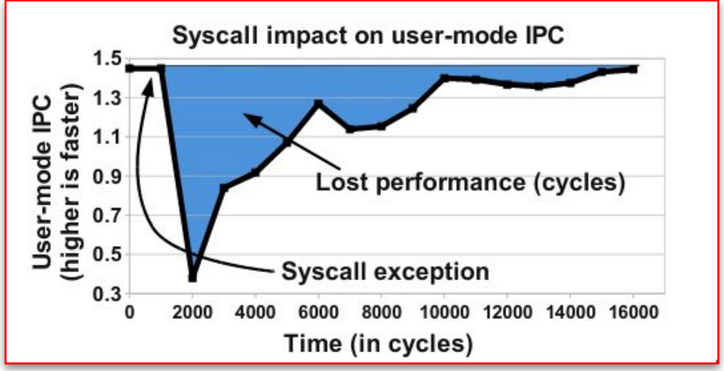
- extremely general packet representation - for any protocol not just TCP/IP
- contain *pointers* and *functions* for any thing possible on the packet
- very very large (struct sk_buf is more than 200 lines of code)
- has close to 100 variables to keep track of information
- on my 4.15 kernel (a bit outdated), it is : **232 bytes!**
  - **Calculate the overhead for a 64 byte packet (~80%)**
  - **Previous research has shown that 63% of the CPU usage during the processing of a 64 byte large packet is skbuff-related [1]**

https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2014-08-1/NET-2014-08-1_15.pdf
Skbuff : https://elixir.bootlin.com/linux/latest/source/include/linux/skbuff.h#L610

The root cause of high overheads - II

System calls are not cheap

- They trap into the kernel
- Disrupt ongoing processing
- Processor ring switch
- Security checks



FlexSC: Flexible System Call Scheduling with Exception-Less System Calls
https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Soares.pdf

(**PS**~ things look a bit differently now, see the *syscall* x86_64 instruction and the shared page table structure, https://www.kernel.org/doc/html/latest/process/adding-syscalls.html)

All this needs to happen 14.8 million times per second (for 10 Gbps)

How does it look performance wise?

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	

Almost 1 microsecond (950 nanoseconds) per packet !

Let's do a basic calculation, what is the time budget per packet

Ethernet payload is 64 bytes, with that total Ethernet frame (20B ETH headers) is 84 bytes

10 Gbps = 10,000,000,000 bits /sec
--
84 * 8
= 14,880,952 packets per second

=> 67.20 nanoseconds / per packet (on 10 Gbps)

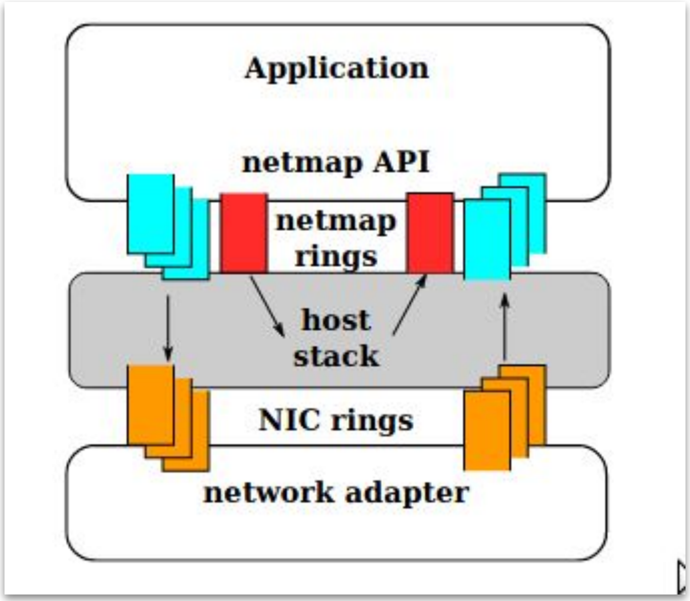
We are clearly way off, and need to optimize it everywhere

What Optimizations does netmap proposes

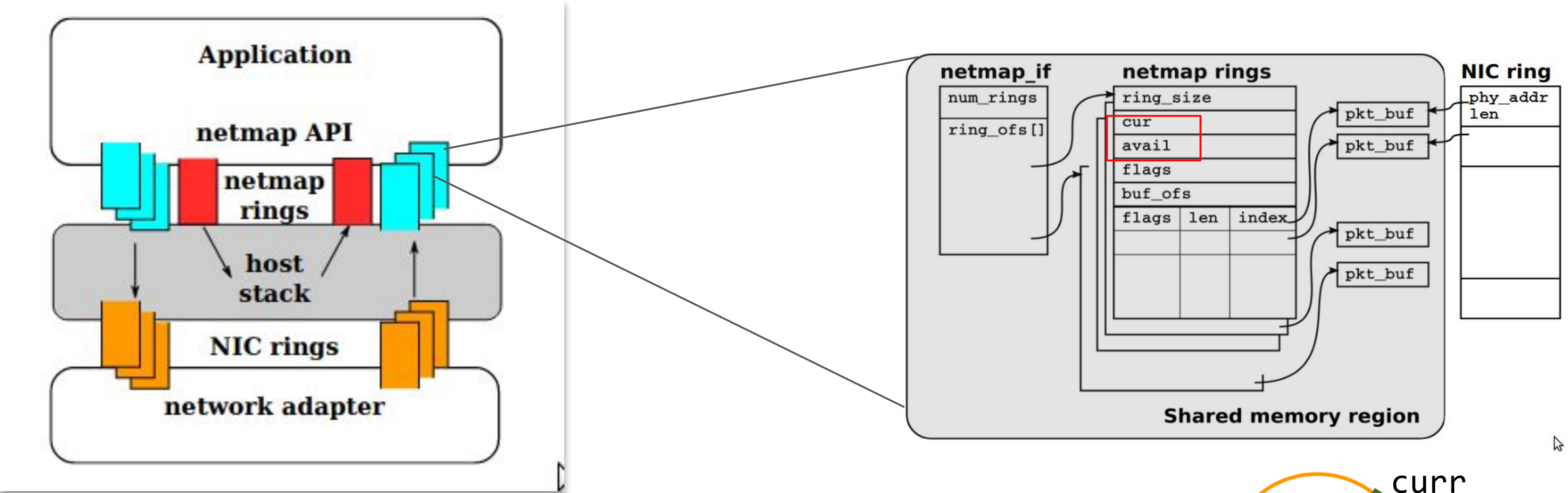
1. Better packet buffer management
   a. All uniform packets, a pool of them are initialized at the boot time (preallocation)
   b. Linear, no fragmentation/reassembly

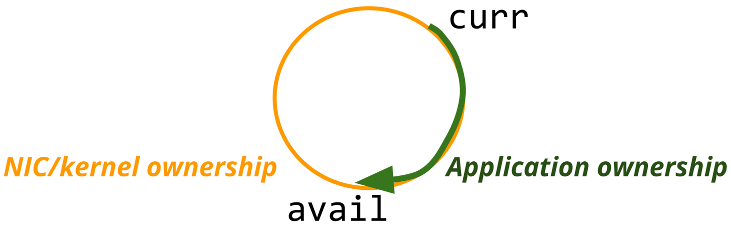
2. Give direct and safe access to NICs RX and TX queues
   a. Zero copy data movement
   b. Very small shared state (a few pointers)

3. Batched system call processing
   a. Send/recv multiple packets in a single call



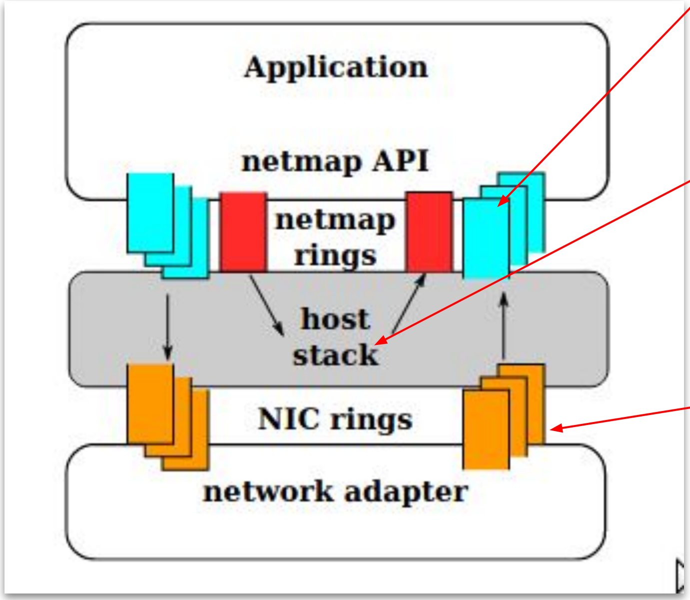
The packet presentation





- **Fixed packet size** : 2 KB (no fragmentation)
- Memory allocated by the kernel (protected) and shared between the NIC and application
- Multiples queues - per core mapping
  - Each queue has its own file descriptor and memory

The Zero-copy stack



- Raw packets are build and transmitted directly from the user space
- System call only to notify the NIC there is work to (so multiple packets can be queued)
- Processing of the `ioctl/select/poll` calls - like any other file descriptor

Very easy to integrate with ***the familiar Linux/IO API***

To achieve this, need support from the NIC driver and the NIC itself with certain capabilities

- Multiqueue interface (virtualization)
- High DMA (to DMA any memory, 64 bits)

A quick glimpse at the code

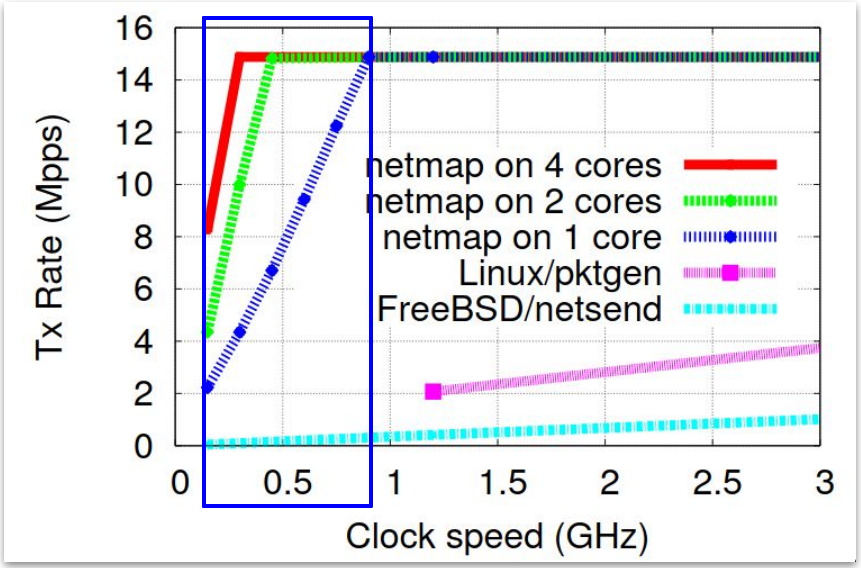
Why does this code looks so strange? Where is a socket?

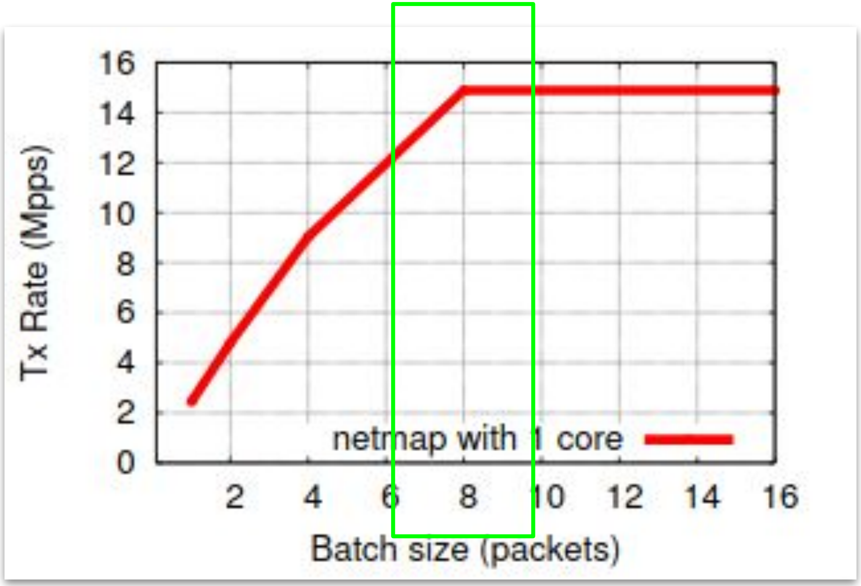
```
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(nmr.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &nmr);
p = mmap(0, nmr.memsize, fds.fd);
nifp = NETMAP_IF(p, nmr.offset);
fds.events = POLLOUT;
for (;;) {
  poll(fds, 1, -1);
  for (r = 0; r < nmr.num_queues; r++) {
    ring = NETMAP_TXRING(nifp, r);
    while (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);
    }
  }
}
```

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

Example of a zero-copy data forwarding

What does all this buys you





1. At 1GHz speed, netmap can saturate a 14.8 Mpps link (default Linux and BSD cannot, with a single core). E.g., Linux has 4 Mpps/core → ~4 cores
2. Batching helps to achieve *"line-rate"*
   a. One cannot achieve line rates without batching
   b. But batching (typically) increases latency

What has netmap done?

Highly influential work

1. Brings attention to per-packet processing
2. Shows the benefit of
    a. Pre-allocating number of buffers (we will come back to this idea later)
    b. Doing system call batching
    c. Flexible packet processing implementation in user space

The last point is very important: *if we can do fast packet processing in userspace, then can we build a fast networking stack in user space?*

What has netmap done?

Highly influential work

1. Brings attention to per-packet processing
2. Shows the benefit of
   a. Pre-allocating number of buffers (we will come back to this idea later)
   b. Doing system call batching
   c. Flexible packet processing implementation in user space

The last point is very important: *if we can do fast packet processing in userspace, then can we build a fast networking stack in user space?*

*PS~ you guys are building one, if not the "**fast**" one ;)*

May be, i am wrong!

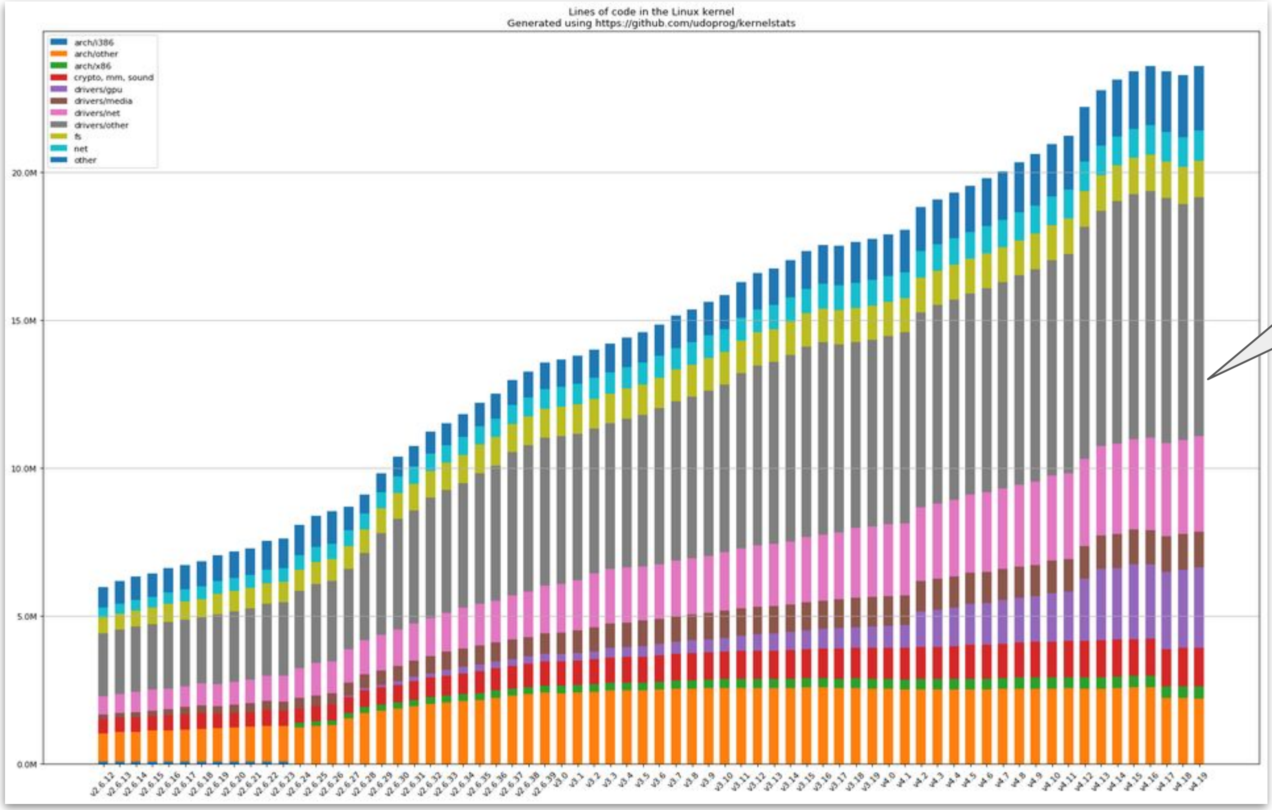
What is unique about packet processing in user space

An operating system kernel is a sacred place

- A modern miracle ...
- Very strictly regulated (arch. + philosophically)
    - Remember: Tanenbaum–Torvalds debate
    - https://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate

https://hackaday.com/2019/10/31/linuxs-marketing-problem/

Linux kernel size - ~30 Million LOC and counting



Do you want to test your code against 30M LOC?

https://www.reddit.com/r/linux/comments/9uxwli/lines_of_code_in_the_linux_kernel/

What is unique about packet processing in user space

An operating system kernel is a sacred place

- A modern miracle ...
- Very strictly regulated (arch. + philosophically)
  - Remember: Tanenbaum–Torvalds debate
  - https://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate
- Needs to run reliably from micro-controllers, cameras, sensors, phones, desktop, servers, supercomputers.
- No security leaks, multiple users
- Any processor and memory architecture for the next **10, 20, 30 years!**

Any special code/customization for one use case : A big no, no !

https://hackaday.com/2019/10/31/linuxs-marketing-problem/

But user space programs are not special

It is your application - do whatever you want

Linux (or any other framework like netmap) ensures proper packet delivery and nothing more

Do value addition:

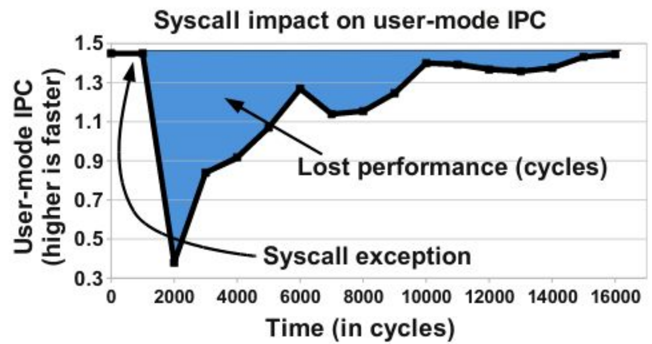
- Tunneling, VPN, tethering, encryption, TORing
- Cloud computing with flexible networking
- Content distribution networks (Geo-locality)
- And much much more ...

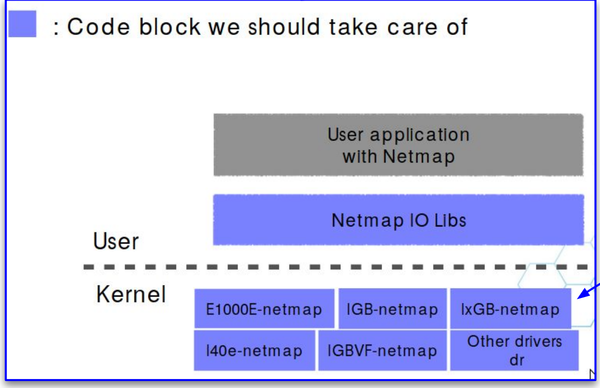
You and I can hack for anything without needing additional kernel complexity

Netmap challenges

Netmap still is integrated in the Linux kernel I/O subsystem

- ioctl calls, select/poll infrastructure
  system calls have its own associated overheads
- need support from every NIC “driver” (too many pieces)





FlexSC: Flexible System Call Scheduling with Exception-Less System Calls
https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Soares.pdf

(**PS**~ things look a bit differently now, see the *syscall* x86_64 instruction and the shared page table structure, https://www.kernel.org/doc/html/latest/process/adding-syscalls.html)

https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/DPDK-China2017-Lin-Telco-Data-Plane-Status.pdf

DPDK Framework

Intel started it in 2010 - **Data Plane Development Kit (DPDK)**

- they wanted to sell CPUs
- they want to show how fast their CPU was for packet Processing

Build the **fastest possible** packet processing framework - extreme!

Highly influential and successful framework (in academia + industry as well) - since then it is multi-vendor, open-source initiative see www.dpdk.org

Used in production for software switches, routers, and cloud networking infrastructure

https://www.zeroto60times.com/formula-one-f1-0-60-times/

DPDK Framework

Intel started it in 2010 - **Data Plane Development Kit (DPDK)**

- they wanted to sell CPUs
- they want to show how fast their CPU was for packet processing

- **Data path** - code path where the actual work is done
  - Try to make it straight forward, no blocking calls, everything is ready to go
- **Control path** - code where resources are managed
  - Slow(er), resourced need to be allocated and managed, can block/sleep

- **Fast path** - common case execution (typically few branches, very simple code)
  - E.g., the next TCP packet is a data packet in EST. state in order, no crazy flags
- **Slow path** - more sanity checks (more branches, hence poor(er) performance)
  - A TCP packet with URG and PSH flag set in the flag

https://www.zeroto60times.com/formula-one-f1-0-60-times/

DPDK Architecture

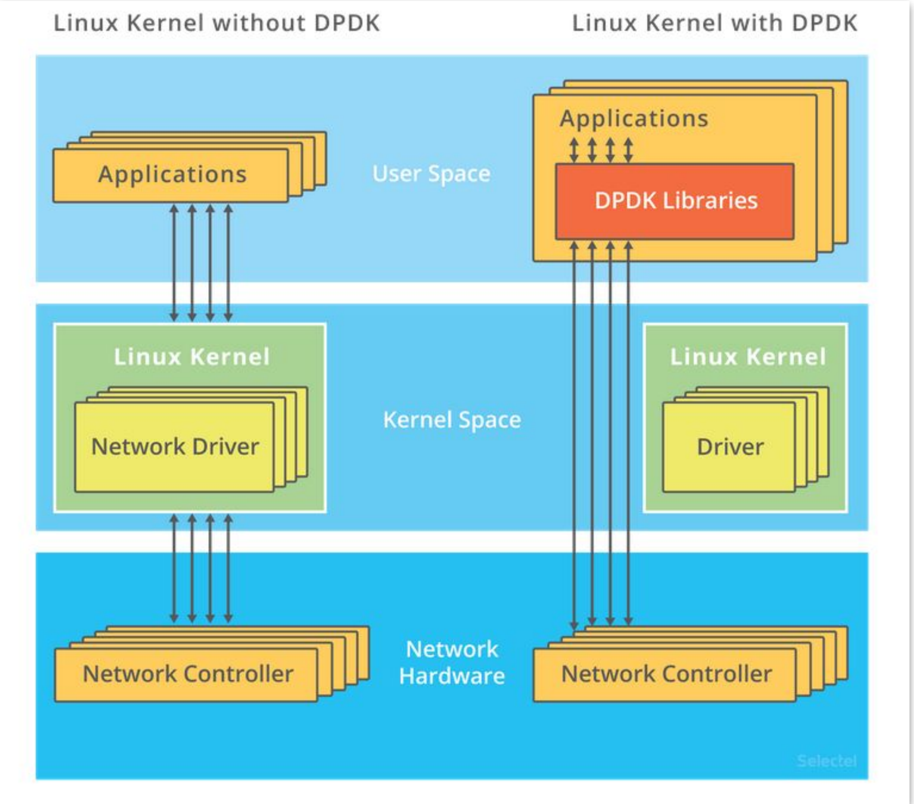
Direct user space packet processing

A list of standard set of infrastructure libraries

No device driver modifications needed, uses (out of the others) Linux's UIO framework

- does userspace memory mapped I/O

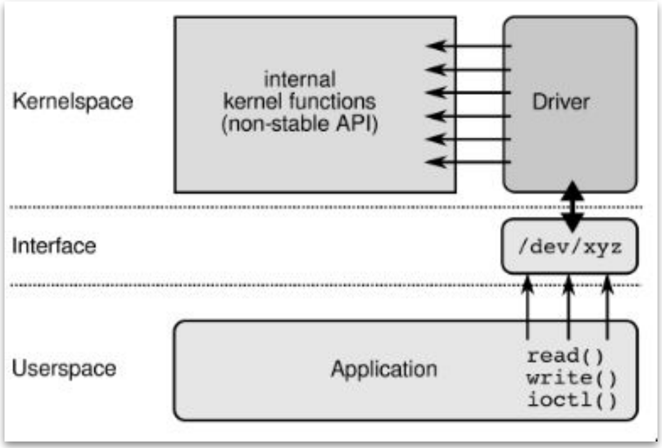
No system call - ONLY polling based drivers on memory-mapped registers

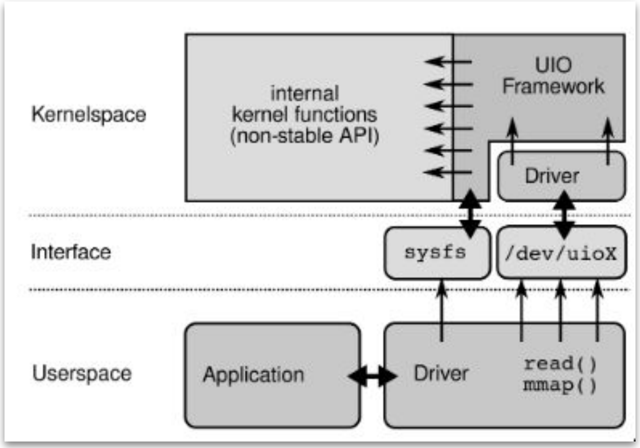


https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2014-08-1/NET-2014-08-1_15.pdf

Linux UIO

Read upside down



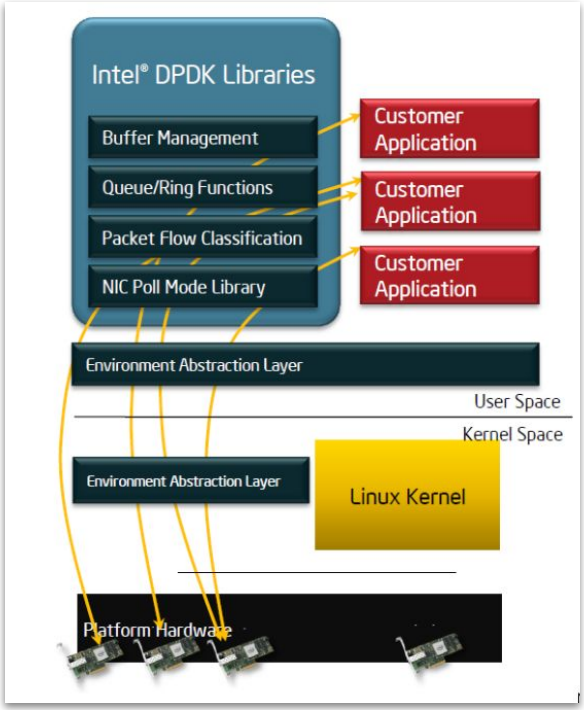


Not every device needs a sophisticated device driver, if all a device does is take commands on some registers and generate interrupts, then use UIO

- Need that device can be managed completely by memory mapped I/O
- Interrupts are delivered as events on the file descriptor
- No need to recompile kernel
- `E.g., uio_pcie_generic` (<-- example driver, there are others)

https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html
https://www.osadl.org/fileadmin/dam/rtlws/12/Koch.pdf

Key ideas in DPDK



1. No system calls or interrupts - all polling
2. No kernel overheads in the **data path**

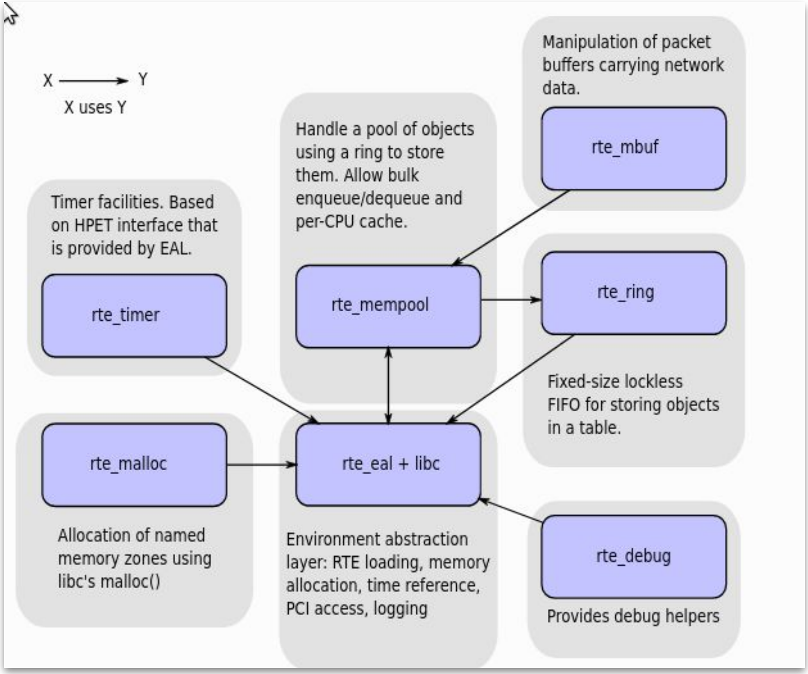
   kernel involvement = ZERO
3. Multiple libraries supporting
   a. Multicore affinity - core/thread pinning
   b. Buffer management - packet buffers
   c. Lockless queue management - using CAS
   d. Huge pages - reduces TLB pressure
   e. Bulk / burst throughput I/O calls - amortize function call invocations (no syscalls here)

https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2014-08-1/NET-2014-08-1_15.pdf

DPDK: High-level components

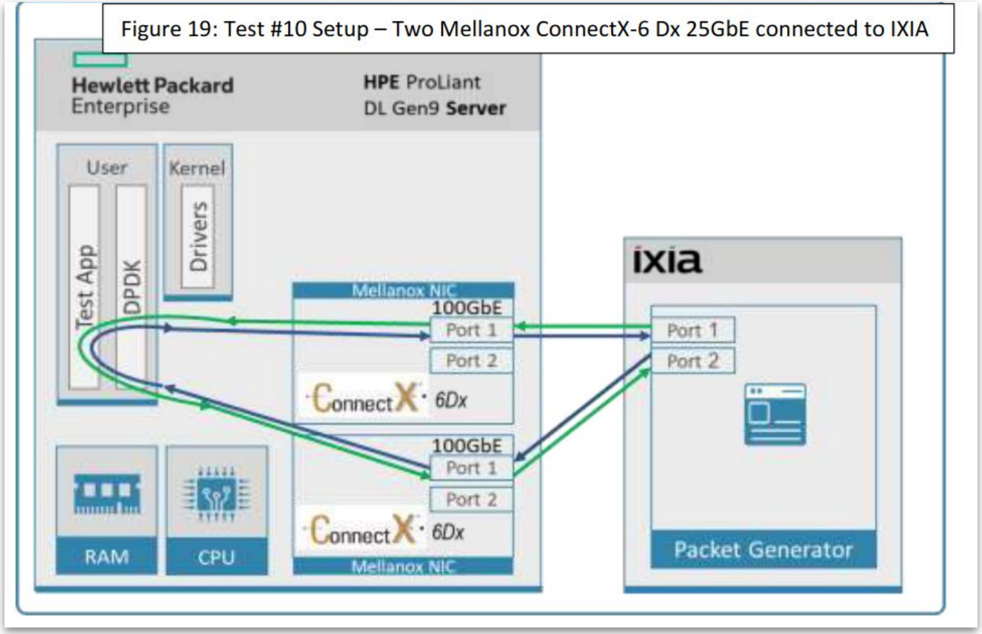
At this point, DPDK is a large framework which is almost rebuilding the whole Linux networking infrastructure in userspace for FAST packet processing

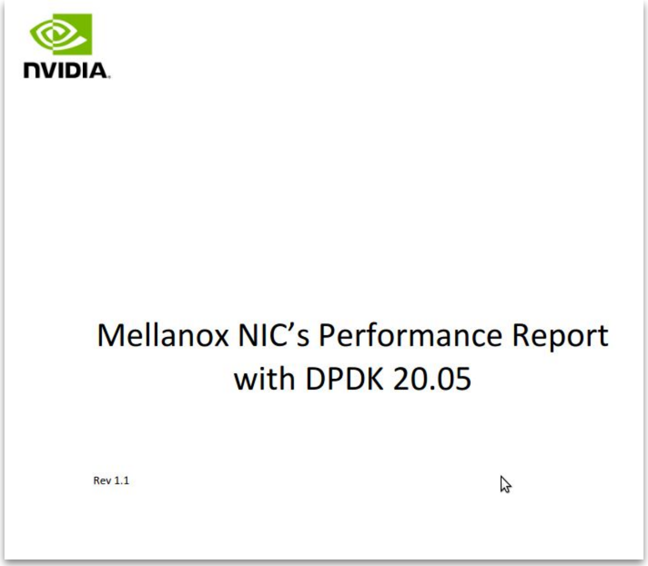
1. Timer facility
2. NUMA-aware, flow-aware memory, core-local allocators (memory management)
3. Per-CPU ring management (notification between CPUs)
4. Debuggers



https://doc.dpdk.org/guides/prog_guide/overview.html

DPDK: Performance - setup

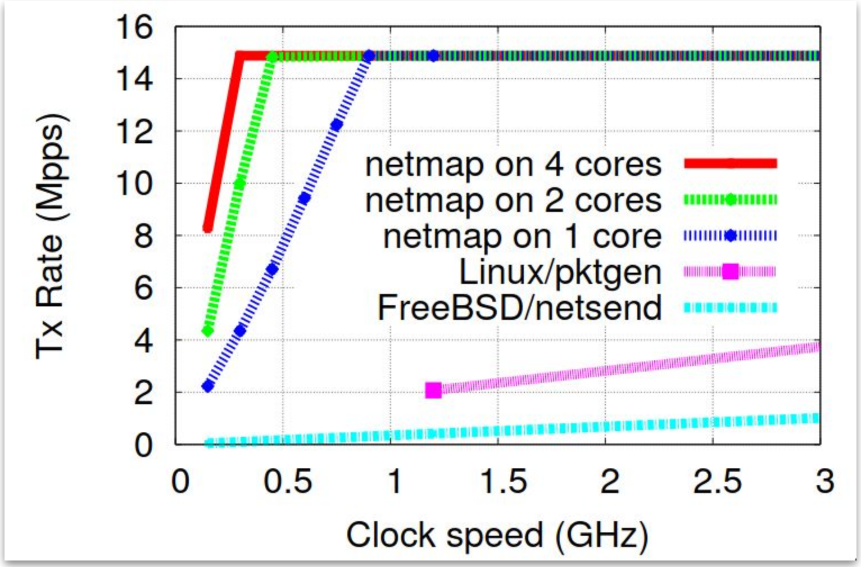




Pretty cool document - gives you whole bunch of insights what is needed to get performance, guesses?

http://fast.dpdk.org/doc/perf/DPDK_20_05_Mellanox_NIC_performance_report.pdf

Recap: netmap performance



1 core (the blue line), hits 14 Mpps at ~1GHz

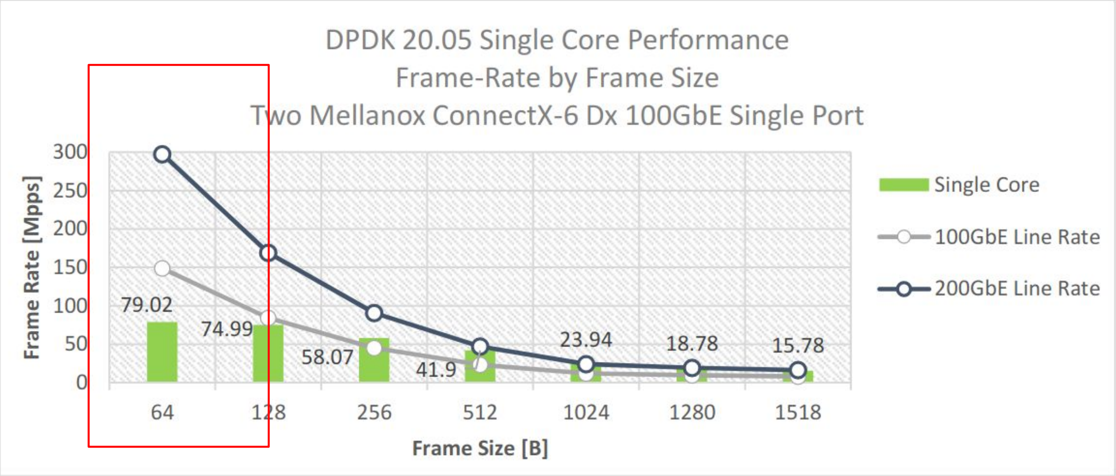
So, let's extrapolate, top of the line CPU frequency ~2-3 GHz ⇒ 14 x {2, 3} ⇒ {28, 42} Mpps

Right?

DPDK: Performance

Insane performance

- **80 Mpps / core**
- **33 cycles / packet**



Frame Size (Bytes)	Frame Rate (Mpps)	Line Rate [200G] (Mpps)	Line Rate [100G] (Mpps)	Throughput (Gbps)	CPU Cycles per packet NOTE: Lower is Better
64	79.02	297.62	148.81	40.459	33
128	74.99	168.92	84.46	76.789	33
256	58.07	90.58	45.29	118.923	30
512	41.9	46.99	23.50	171.623	31
1024	23.94	23.95	11.97	196.131	33
1280	18.78	19.23	9.62	192.342	32
1518	15.78	16.25	8.13	191.638	34

Upcoming DAS-6 VU Supercomputer

- DAS-6 has 100 Gbps Ethernet
- We are just finalizing the configuration now
- It should be up and operational in a few months time

Want to experiment, and generate 148 Mpps? ;) Come talk to us!

mTCP: Scalable User Space TCP Stack

mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems

EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong
Sunghwan Ihm*, Dongsu Han, and KyoungSoo Park
KAIST *Princeton University

Abstract

Scaling the performance of short TCP connections on multicore systems is fundamentally challenging. Although many proposals have attempted to address various shortcomings, inefficiency of the kernel implementation still persists. For example, even state-of-the-art designs spend 70% to 80% of CPU cycles in handling TCP connections in the kernel, leaving only small room for innovation in the user-level program.

This work presents mTCP, a high-performance user-level TCP stack for multicore systems. mTCP addresses the inefficiencies from the ground up—from packet I/O and TCP connection management to the application interface. In addition to adopting well-known techniques, our design (1) translates multiple expensive system calls into a single shared memory reference, (2) allows efficient flow-

also critical for backend systems (*e.g.*, memcached clusters [36]) and middleboxes (*e.g.*, SSL proxies [32] and redundancy elimination [31]) that must process TCP connections at high speed. Despite recent advances in software packet processing [4, 7, 21, 27, 39], supporting high TCP transaction rates remains very challenging. For example, Linux TCP transaction rates peak at about 0.3 million transactions per second (shown in Section 5), whereas packet I/O can scale up to tens of millions packets per second [4, 27, 39].

Prior studies attribute the inefficiency to either the high system call overhead of the operating system [28, 40, 43] or inefficient implementations that cause resource contention on multicore systems [37]. The former approach drastically changes the I/O abstraction (*e.g.*, socket API) to amortize the cost of system calls. The practical lim-

https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-jeong.pdf

mTCP: Scalable User Space TCP Stack

A bit of specialization

mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems

EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong
Sunghwan Ihm*, Dongsu Han, and KyoungSoo Park
KAIST *Princeton University

Abstract

Scaling the performance of short TCP connections on multicore systems is fundamentally challenging. Although many proposals have attempted to address various shortcomings, inefficiency of the kernel implementation still persists. For example, even state-of-the-art designs spend 70% to 80% of CPU cycles in handling TCP connections in the kernel, leaving only small room for innovation in the user-level program.

This work presents mTCP, a high-performance user-level TCP stack for multicore systems. mTCP addresses the inefficiencies from the ground up—from packet I/O and TCP connection management to the application interface. In addition to adopting well-known techniques, our design (1) translates multiple expensive system calls into a single shared memory reference, (2) allows efficient flow-

also critical for backend systems (*e.g.*, memcached clusters [36]) and middleboxes (*e.g.*, SSL proxies [32] and redundancy elimination [31]) that must process TCP connections at high speed. Despite recent advances in software packet processing [4, 7, 21, 27, 39], supporting high TCP transaction rates remains very challenging. For example, Linux TCP transaction rates peak at about 0.3 million transactions per second (shown in Section 5), whereas packet I/O can scale up to tens of millions packets per second [4, 27, 39].

Prior studies attribute the inefficiency to either the high system call overhead of the operating system [28, 40, 43] or inefficient implementations that cause resource contention on multicore systems [37]. The former approach drastically changes the I/O abstraction (*e.g.*, socket API) to amortize the cost of system calls. The practical lim-

https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-jeong.pdf

What is the problem that mTCP is solving?

Building on from MegaPipe, focus is on small, short-lived connections

1. Do multi core scalability *(MegaPipe does it)*
2. No new radical API *(Limitations of MegaPipe)*
3. No kernel modification*(Limitations of MegaPipe)*

Why in user space? We have seen some arguments for *packet processing*

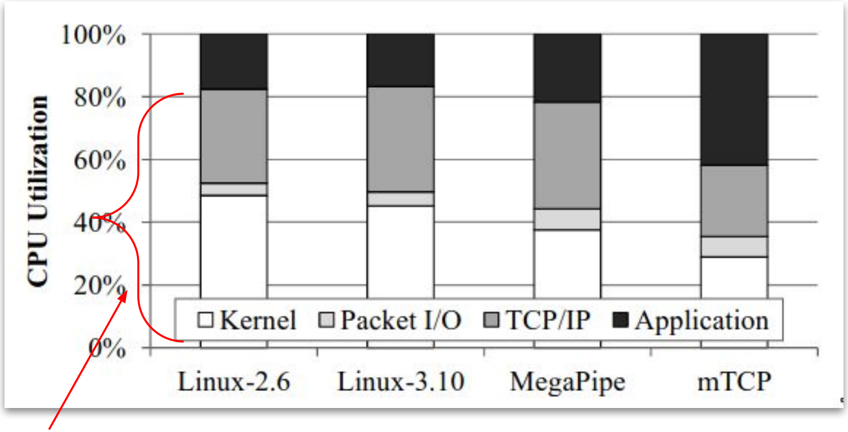
- Expensive syscall
- Metadata/data copies
- Kernel environment
- Generality vs specialization argument

What is the problem that mTCP is solving?

(recap) Challenges with the kernel stack

1. Locality: SO_REUSE and split among cores
2. Shared fd space : decouple fd
3. Inefficient packet processing (netmap)
4. Syscall overheads (batching)

Previous works improve, but still not quite.



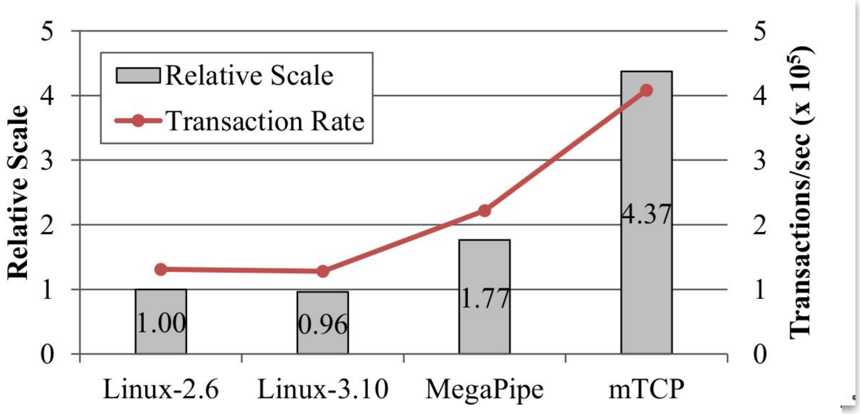
Between all the packet processing kernel, and TCP/IP - there is very limited number of CPU cycles are left for the application
Kernel consumes 80% of CPU cycles

What is the problem that mTCP is solving?

(recap) Challenges with the kernel stack

1. Locality: SO_REUSE and split
2. Shared fd space
3. Inefficient packet processing (netmap)
4. Syscall overheads

Previous works improve, but still not quite.

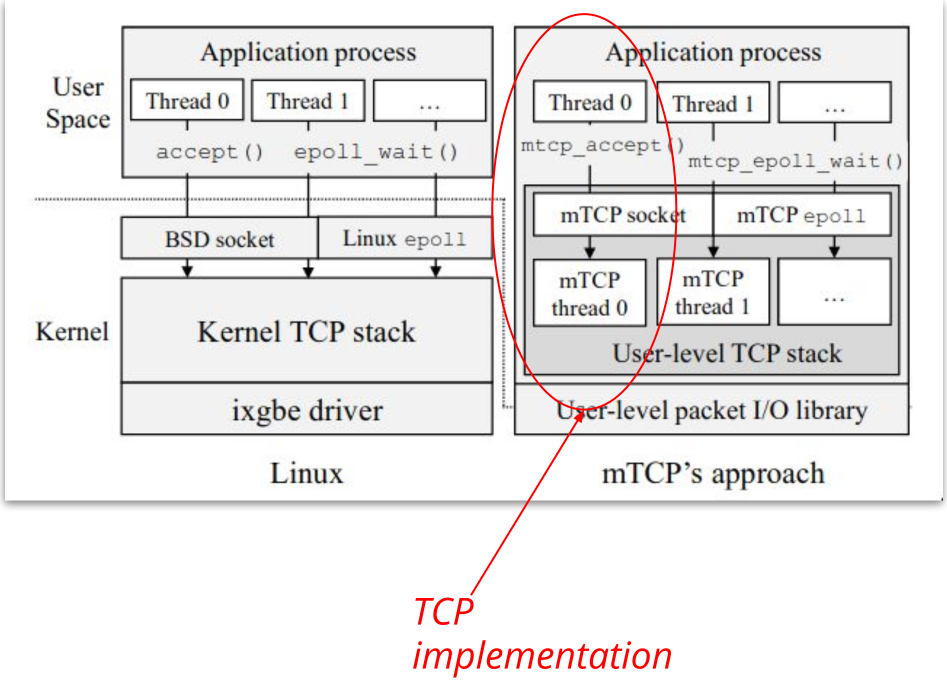


Research question: we have seen userspace packet I/O doing 10s million packets/sec in userspace? Can we do the same with the TCP stack (with sockets) implementation?

Not all cycles are spent equally

mTCP basic ideas

- TCP stack implementation in userspace
  - *Flexibility and easy of development and use*
  - *Specialization of TCP - common options, fast path*
- Leverage packet processing frameworks to deliver performance
  - Uses packet shader (similar idea as netmap)
  - 10s million packets/sec in user space
- Multi-core scalability
  - Per-application thread design
  - Transparent batching of events and packet I/O



mTCP: Packet Processing Improvements

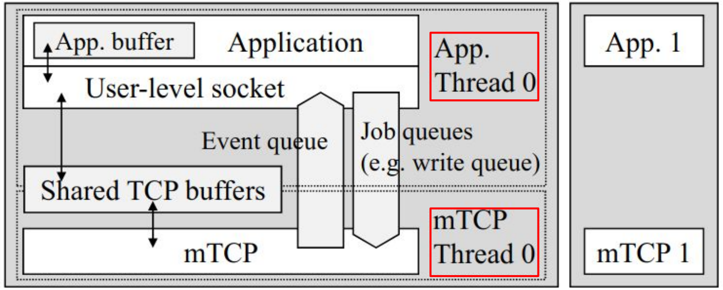
Key challenges (beyond what we discussed previously):

1. DPDK does polling - waste of CPU cycles
2. Netmaps allows `select/epoll` on file descriptors, but integrated with kernel

mTCP does its own implementation of select on TX/RX queues (not files)

- `ps_select(queues, timeout);`
  - Returns immediately with packets, if there are
  - Otherwise, wait for events from the kernel
- Not integrated with the Linux file/event management system to avoid overheads
- mTCP's underlying PS engine also support packet batching
  - Amortize for DMA, IOMMU costs (and other associated architectural costs)

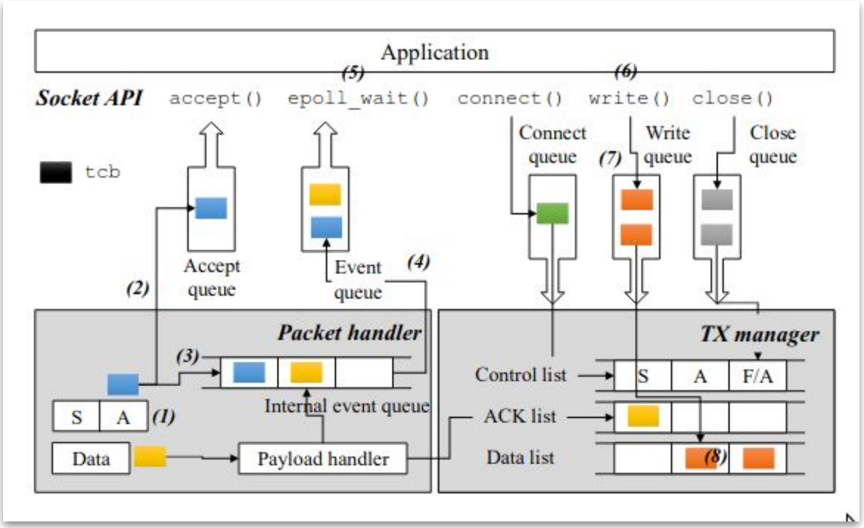
mTCP: Userlevel TCP Stack



- Can it have **zero-thread** TCP model?
  - Means - can there be no active threads in the mTCP library?
  - Answer: No, why?
- The thread model: `1:1 mTCP:app_thread`
  - Shared TCP buffer , access only via using the job and events queues
  - Internal I/O and event queues, supporting queueing and batching
- Application and mTCP threads
  - All data structures (file descriptors, TCP state information) partitioned between cores
  - Threads pinned together on one core, and RSS configured to deliver packet there
  - Lock-free data structures using single producer/single consumer queues
  - Each core has its own memory allocator and caching

TCP walk through

1. Look up TCP control block (**TCB**) (see RFC 793)
2. If it is an ACK for SYN - then put it in the accept queue
3. Process a bunch of TCP packets
4. Queue event and wake up process
5. App get multiple events in a single epoll notification
6. Write multiple responses (no context switch)
7. Enqueue TCB in the write queue for processing
8. Packet transmission

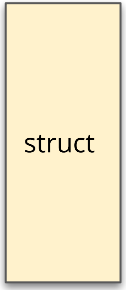


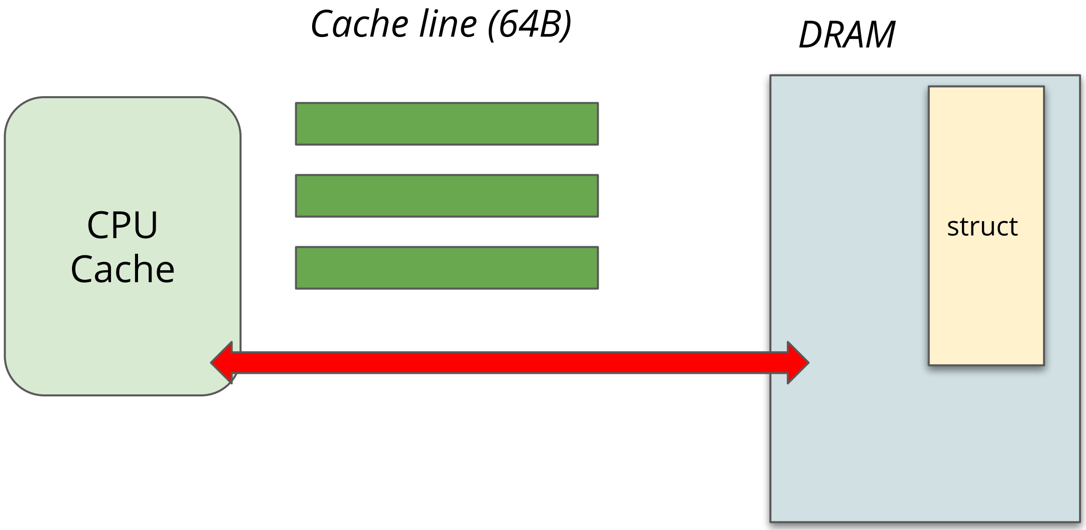
No global queues, all core local, no locking. mTCP offers the same BSD/TCP socket semantics

- `socket → mtcp_socket`
- `send → mtcp_send`
- `poll → mtcp_poll`

Cache Alignments

```
struct tcp_state
{
  ● seq_num
  ● ack_num
  ● src_port
  ● dst_port
  ● tcp_state
...
};
```





Recall:

- CPU caches have cache lines of certain size : 64 bytes (typically)
- That is the unit of data transfer between the CPU cache and DRAM

So you want to align your tcp_struct on the cache line size

- Group together frequently accessed items

For example ...

```
struct tcp_state {
...
// 56 bytes of data
uint32_t process_id; // +4B => 60B
uint32_t sequence_num; // +4B => 64B
//--- next cache line
uint32_t ack_num;// +4 bytes => 68B
// other local resources
...
}
```

0x0000

0x0040

0x0080

Here in this case, due to the unfortunate ordering in which the struct fields are defined, seq and ack number happen to lie on different cache lines

However, often they are processes together. Hence, it makes sense to pack them on the same cache line by reordering their definition order

In the Linux kernel you see many such examples ...

For example ...

```
struct tcp_state {
...
// 56 bytes of data
uint32_t process_id; // +4B => 60B
uint32_t sequence_num; // +4B => 64B
//--- next cache line
uint32_t ack_num;// +4 bytes => 68B
// other local resources
...
}
```

0x0000

0x0040

0x0080

```
struct tcp_state {
...
// 56 bytes of data
uint32_t ack_num;// +4 bytes => 60B
uint32_t sequence_num; // +4B => 64B
//--- next cache line
uint32_t process_id; // +4B => 68B
// other local resources
...
}
```

Here in this case, due to the unfortunate ordering in which the struct fields are defined, seq and ack number happen to lie on different cache lines

However, often they are processes together. Hence, it makes sense to pack them on the same cache line by reordering their definition order

In the Linux kernel you see many such examples ...

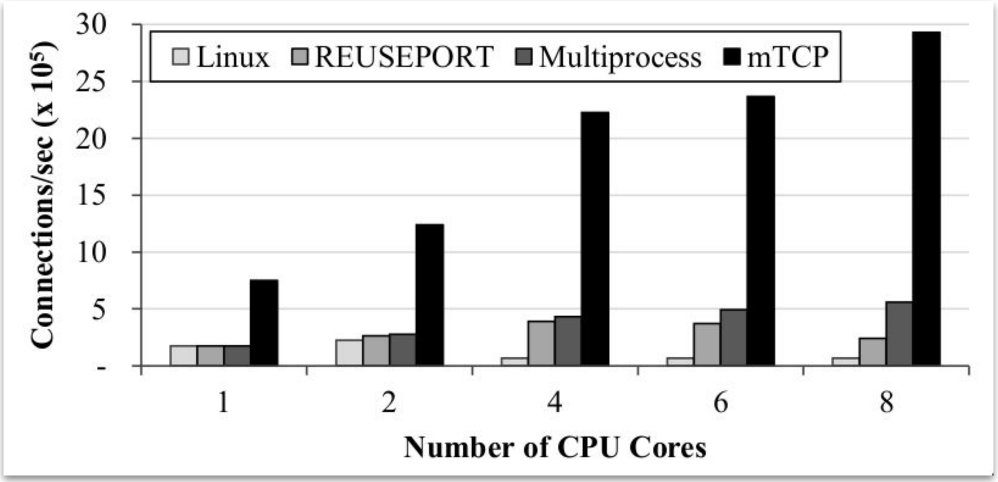
Two TCP specific optimizations

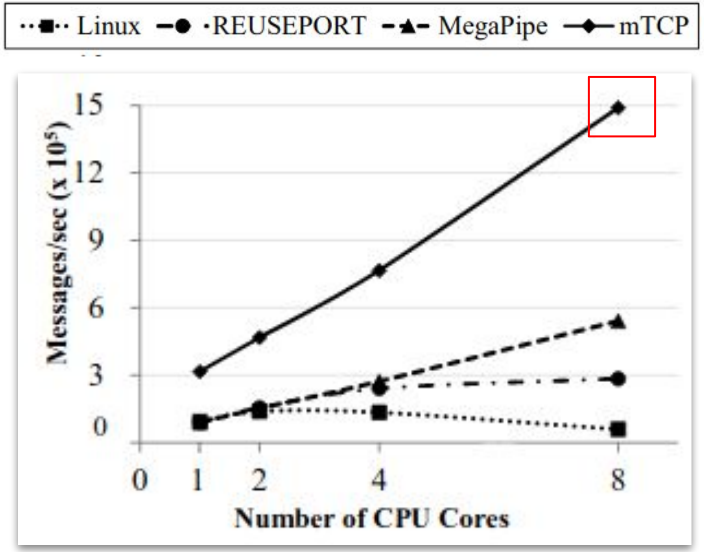
1. 3 types of the tx queues: **control, ACK, and data**
   a. Small, short-lived TCP connection are control message heavy (SYN/ACK)
   b. Priority to the “control” packets, when transmissing : (priority order)

      Control (SYN/SYNACK)→ ACK → data packets

2. TCP cache
   a. Lot of new connection mean: lots of new socket descriptors, buffers, TCBs, queues, pointers, structure allocation
   b. Proposes a pre-allocated pool of structures per thread and reuse it constantly

What all of this buys you?





- Significant performance improvements over previous efforts
- **Milestone work**: proof of concept of an efficient TCP stack in userspace
  - Implements all known optimizations
  - End-to-end batching : packet, events, I/O calls

Limitations / Considerations for mTCP

Limited memory protection between the shared mTCP library and application

- The idea of “**fate-sharing**”

Change in application semantics if they attach to specific “file descriptor” semantics (not all Linux I/O are supported on the fd)

By passing all kernel services - packet scheduling, firewalling, routing

Limited number of TX/RX queues, and no unlimited multi-application support

Conceptually

Your ANP netstack is very close to what mTCP has build

- Except you are not using a user space packet processing library but the Linux TUN/TAP infrastructure

- Think about ...
  - How do you allocate a file descriptor for the socket call ?
  - Are you doing something to deliver better multi core scalability?
  - Are you doing something for better cache alignments?
  - What is your threading model?

Recap

	Accept queue	Locality	API	Event handling	Packet I/O	App. changes	Kernel modification
Netmap	*x*	*x*	*x*	*Syscall*	*Batched, events*	*x*	*Only the NIC driver*
DPDK	*x*	*Yes*	*x*	*x*	*Polling*	*x*	*Support from the NIC driver*
Linux 2.16	*Shared*	*None*	*BSD sockets*	*Syscalls*	*Per-packet*	*No*	*No*
Linux 3.19	*Per-core*	*None*	*BSD sockets*	*Syscalls*	*Per-packet*	***SO_REUSEPORT***	*No*
MegaPipe	*Per-core*	*Yes*	*lwsocket*	*Batched Syscalls*	*Per-packet*	*Yes, new API*	*Yes*
mTCP	*Per-core*	*Yes*	*mTCP sockets*	*Batched Funcalls*	*Batched*	*Minimum, mTCP sockets*	*No (multiqueue)*

What you should know from this lecture

1. What are packet processing frameworks
2. What are the key innovations in DPDK and Netmap
3. What is good or bad about the kernel space Linux networking stack
4. What is good or bad about user space networking stacks
5. What is difference between mTCP and Megapipe approaches
6. General concerns, tricks, and design choices for userspace networking stacks - batching, locality and affinity, APIs and internals

Further reading

1. Userspace Networking with DPDK, https://www.linuxjournal.com/content/userspace-networking-dpdk
2. Understanding DPDK, https://www.slideshare.net/garyachy/dpdk-44585840
3. Introduction to DPDK: Architecture and Principles, https://blog.selectel.com/introduction-dpdk-architecture-principles/
4. DPDK: Multi Architecture High Performance Packet Processing, https://www.slideshare.net/MichelleHolley1/dpdk-multi-architecture-high-performance-packet-processing-72911726