Benchmarking packet reception with `AF_XDP`, DPDK and `io_uring`

Greg Depoire--Ferrer

Motivation

The performance of networking stacks

Why use a different networking stack?

- A need for processing packets on **commodity hardware** with **low overhead**.[1]
- **Portability** and integration with the operating system versus **performance**.

`io_uring`

Keeps evolving ⟶ need for new benchmarks.

---

[1] G. Wan, F. Gong, T. Barbette, and Z. Durumeric, "Retina: analyzing 100GbE traffic on commodity hardware," in *Proceedings of the ACM SIGCOMM 2022 Conference*, in SIGCOMM '22. Association for Computing Machinery, 2022, pp. 530–544. doi: 10.1145/3544216.3544227.

Comparisons between networking stacks

Networking stacks support different layers.

Ethernet

- `AF_XDP`
- DPDK

TCP

- `epoll`
- `io_uring`

Ethernet

DPDK

Description

- Kernel bypass ⟶ no overhead due to transition between kernel space and user space.
- Networking code is written to be as fast as possible ⟶ faster than Linux networking stack.
- Need to reserve an entire NIC for the application ⟶ no sharing of resources.
- Linux NIC drivers cannot be reused.

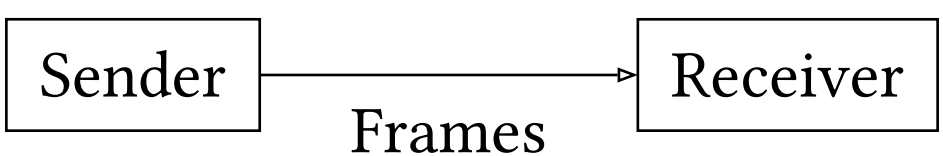
Express Data Path (XDP)

Description

- **XDP programs** are BPF programs that are called for every incoming packet just after reception but before allocating memory for a socket buffer.
- They can drop packets, modify them, and chose to pass them to the networking stack, redirect them to a port or to userspace for further processing.
- Packets redirected to userspace are received on a `AF_XDP` socket.
- Need to reserve a NIC queue for the application.
- Uses Linux NIC drivers.

Experiment

- *Sender* machine sends packets as fast as possible to *receiver* machine using `pktgen` (Linux kernel module).
- Measure the number of packets received per second.
- Vary the number of **cores** and **packet size**.



Results

Conclusions

TCP

BSD socket API

- Create a socket: `socket`, `accept`
- Receiving data: `recv`, `recvfrom`, `read`
- Sending data: `send`, `sendto`, `write`

Characteristics

- One system call per operation
- **Blocking**: no compute or IO can be done in the meanwhile

BSD socket API

Example

```
import random
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('0.0.0.0', 53))

while True:
    datagram, address = s.recvfrom(2048)
    print(f'Received {datagram} from {address}.')
```

`select` and friends

```
import socket, select

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 80))
s.listen(8)

sockets = [s]

while True:
    ready, _, _ = select.select(sockets, [], [])

for r in ready:
    if r == s:
        c, addr = s.accept()
        sockets.append(c)
    else:
        data = sock.recv(2048)
        if data:
            print(f'Received {data} from {r}')
        else:
            l.remove(r)
```

- **Readiness-based**
- One system call per operation
- **Non-blocking**
- Variants: `poll`, **`epoll`**, kqueue

Readiness-based interface

```
/* s is ready... */

for (;;) {
    char buf[2048];

    ssize_t n = recv(s, buf, sizeof(buf), 0);
    if (n == 0)
        break;

    process(buf, n);
}
```

`epoll`

- Also **readiness-based**
- Modern version of `select` with less copying
- FreeBSD equivalent: `kqueue`

io_uring

- **Completion-based**
- Start many operations with a single system call

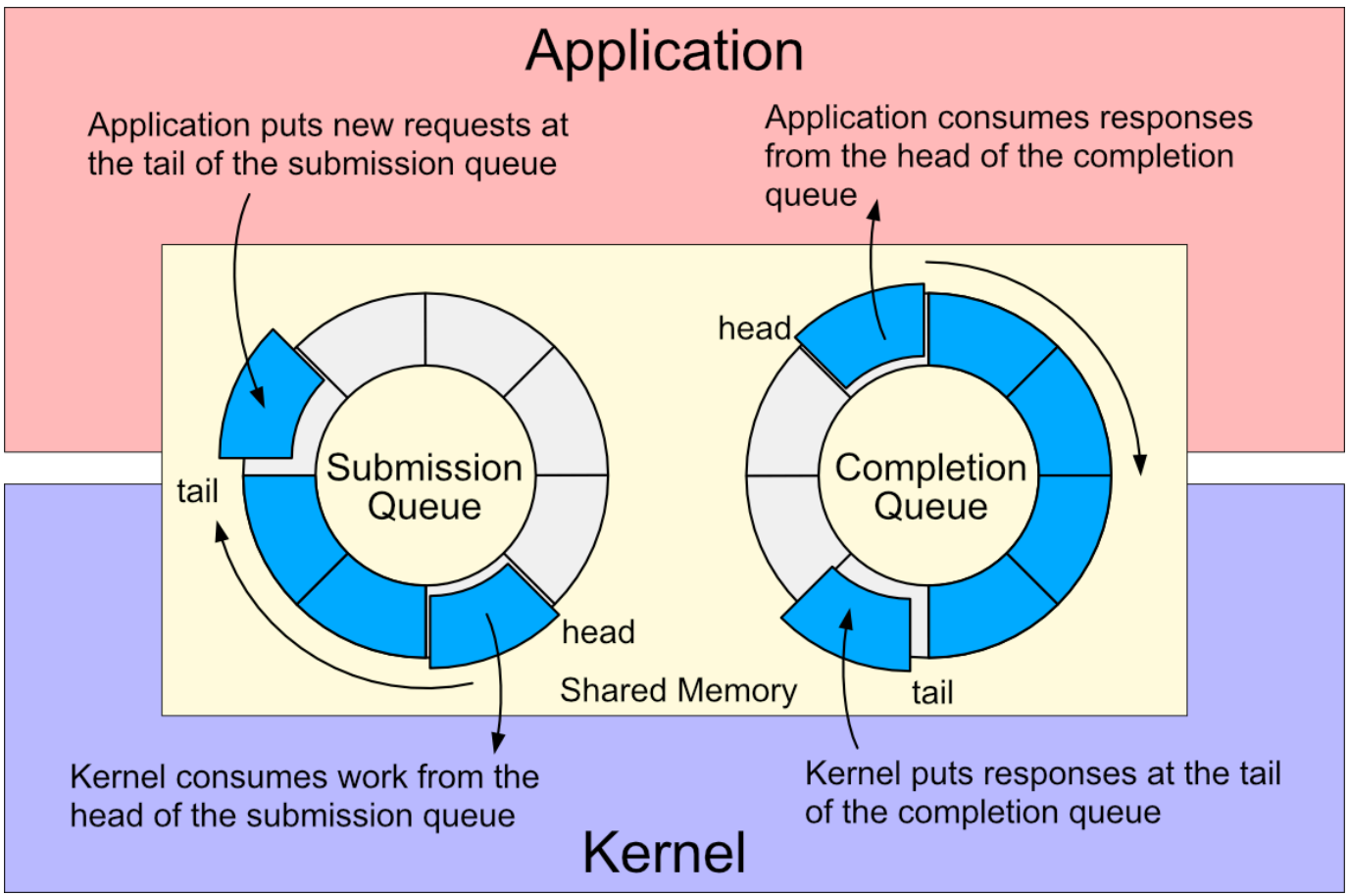
Completion-based interface

```
char *buf = malloc(2048);
start_read_operation(s, buf, sizeof(buf), 1234);

/* Later... */

int id = process_completed_operation();
printf("%d\n", id); // 1234
```

Presentation



`io_uring`

Performance improvements

- Fixed files and buffers (May 2019)
- Buffer ring (July 2022)
- Zero-copy reception (not merged yet!)

io_uring

Fixed files

Pre-register FDs with the ring to reduce the overhead of:

- reference counting, and
- descriptor table lookup.

```
int io_uring_register_files(
    struct io_uring *ring,
    const int *files,
    unsigned nr_files,
);
```

io_uring

Buffer ring

Instead of specifying a buffer to use in `read` operations, let the kernel pick a buffer from another ring buffer.

```
struct io_uring_buf_ring *io_uring_setup_buf_ring(
    struct io_uring *ring,
    unsigned int nentries,
    int bgid,
    unsigned int flags,
    int *ret,
);
```

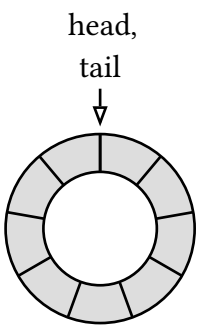
`io_uring`

Zero-copy reception

A new receive operation where the NIC directly writes to application memory.

io_uring

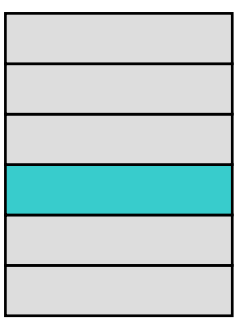
Area



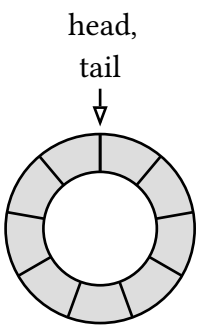
Refill buffer

1. App submits `RECV_ZC` operation.

io_uring



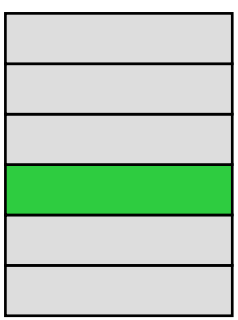
Area



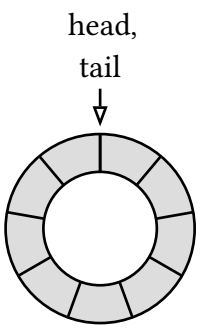
Refill buffer

1. App submits `RECV_ZC` operation.
2. Kernel picks free buffer in area.

io_uring



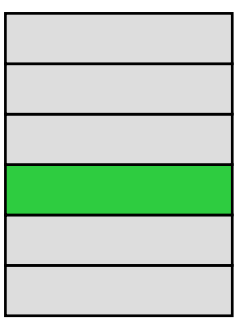
Area



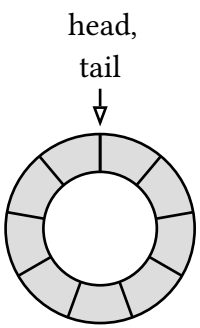
Refill buffer

1. App submits `RECV_ZC` operation.
2. Kernel picks free buffer in area.
3. NIC writes to buffer.

io_uring



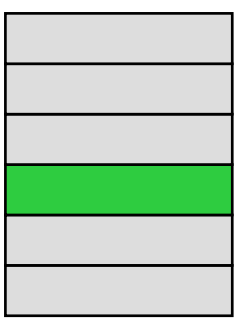
Area



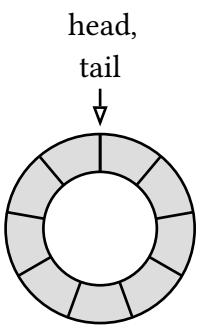
Refill buffer

1. App submits `RECV_ZC` operation.
2. Kernel picks free buffer in area.
3. NIC writes to buffer.
4. Kernel enqueues completion.

io_uring



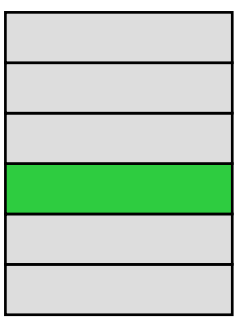
Area



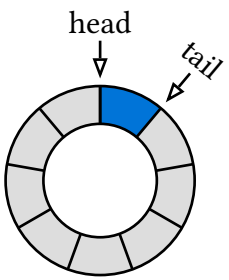
Refill buffer

1. App submits `RECV_ZC` operation.
2. Kernel picks free buffer in area.
3. NIC writes to buffer.
4. Kernel enqueues completion.
5. App processes data in buffer.

io_uring



Area

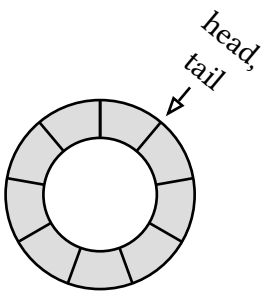


Refill buffer

1. App submits `RECV_ZC` operation.
2. Kernel picks free buffer in area.
3. NIC writes to buffer.
4. Kernel enqueues completion.
5. App processes data in buffer.
6. App enqueues buffer ready to be reused.

io_uring

Area



Refill buffer

1. App submits `RECV_ZC` operation.
2. Kernel picks free buffer in area.
3. NIC writes to buffer.
4. Kernel enqueues completion.
5. App processes data in buffer.
6. App enqueues buffer ready to be reused.
7. Kernel marks buffer as available.

io_uring

Support

This work is not merged yet into Linux and `liburing` (a library to use `io_uring`). The patch series have some example code, but not much other than that.

I created a Rust wrapper to setup the area and refill buffer needed to use the new `RECV_ZC` operation: https://github.com/beviu/io-uring-zcrx.

I modified the `io-uring` Rust crate to add support for this new operation: https://github.com/beviu/io-uring/compare/master...zcrx.

Experiment

- *Sender* machine sends data on TCP socket as fast as possible to *receiver* machine.
- Two TCP server implementations (see `server-io-uring` and `server-epoll` directories).
- Measure the bandwidth.

Results

Appendix

Virtual machine setup

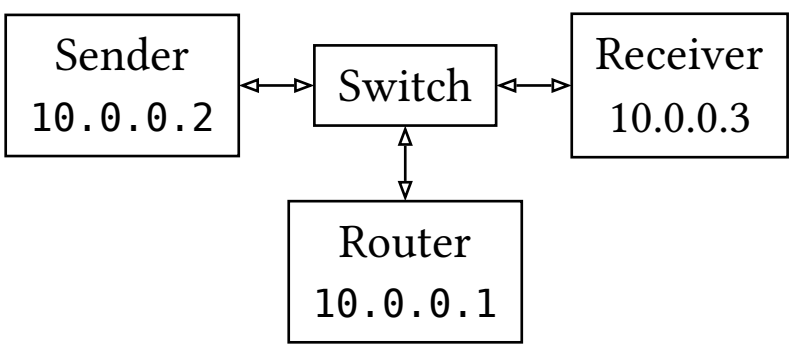
I prepared the benchmark setup beforehand, and tested it in virtual machines on my PC first to make sure I was ready before testing on actual Grid5000 hardware.

I installed Arch Linux in two VMs and configured a bridge with a tap device for each VM in the host. I used `virtio-net` NICs.

Virtual machine setup

Host network interfaces setup

```
ip tuntap add mode tap tap0
ip tuntap add mode tap tap1
ip link add name br0 type bridge
ip link set dev br0 up
ip link set tap0 up
ip link set tap1 up
ip link set tap0 master br0
ip link set tap1 master br0
ip addr add 10.0.0.1/24 dev br0
```



Configure a NAT to give the tap devices access to the Internet:

```
sysctl net.ipv4.ip_forward=1
nft add table nat
nft 'add chain nat postrouting { type nat hook postrouting
priority 100 ; }'
nft add rule nat postrouting ip saddr 10.0.0.0/24 masquerade
```

Virtual machine setup

QEMU command line

Make sure to replace the tap interface name and MAC address for the receiver VM. The arguments for specifying the drives were omitted.

```
qemu-system-x86_64 \
  -machine type=q35,accel=kvm,kernel-irqchip=split \
  -cpu host -smp 4 -m 2G \
  -device intel-iommu,intremap=on,caching-mode=on \
  -vga none \
  -serial stdio \
  -monitor none \
  -nographic \
  -netdev tap,id=tap,ifname=tap0,script=no,downscript=no \
  -device virtio-net-pci,netdev=tap,mac=52:54:00:f8:e2:e3
```

Guest network interfaces setup

On the sender VM:

```
ip addr add 10.0.0.2/24 dev enp0s2
```

On the receiver VM:

```
ip addr add 10.0.0.3/24 dev enp0s2
```

Compiling DPDK

```
curl https://fast.dpdk.org/rel/dpdk-24.11.1.tar.xz -O
tar -xf dpdk-24.11.1.tar.xz
cd dpdk-stable-24.11.1/
meson setup build -Dplatform=native
cd build
ninja
sudo ninja install
sudo sh -c 'echo /usr/local/lib > /etc/ld.so.conf.d/
local.conf' # Needed on Arch Linux.
sudo ldconfig
```

Compiling Pktgen-DPDK

```
git clone https://github.com/pktgen/Pktgen-DPDK.git --branch
pktgen-24.10.3
cd Pktgen-DPDK
PKG_CONFIG_PATH=/usr/local/lib/pkgconfig meson setup build
cd build
ninja
sudo ninja install
```

Running Pktgen-DPDK

Make sure to replace the PCI slot with your NIC's.

```
sudo modprobe vfio-pci
sudo dpdk-devbind.py --bind vfio-pci 0000:00:02.0 --force
sudo sh -c 'echo 512 > /sys/kernel/mm/hugepages/
hugepages-2048kB/nr_hugepages'
sudo pktgen -l 0,1 -n 1 -a 0000:00:02.0 -- -P -T -m 1.0
```

Running the `AF_XDP` server

I used an existing server implementation. It receives packets and drops them right away.

```
git clone https://github.com/xdp-project/bpf-examples.git --
recurse-submodules
cd bpf-examples
cd AF_XDP-example
make
```

Run the *rxdrop* example:

```
sudo taskset -c 0 ./xdpsock -i enp0s2 -q 0 -r
```