The eXpress Data Path

Fast Programmable Packet Processing in the Operating System Kernel

Toke Høiland-Jørgensen (Karlstad University)
Jesper Dangaard Brouer (Red Hat)
Daniel Borkmann (Cilium.io)
John Fastabend (Cilium.io)
Tom Herbert (Quantonium Inc)
David Ahern (Cumulus Networks)
David Miller (Red Hat)

CoNEXT '18
Heraklion, Greece, Dec 2018

Outline

- Challenges with high-speed packet processing
- XDP design
- Performance evaluation
- Example applications
- Conclusion

High-Speed Packet Processing is Hard

- Millions of packets per second
  - 10 Gbps: 14.8Mpps / 67.5 ns per packet
  - 100 Gbps: 148Mpps / 6.75 ns per packet

Operating system stacks are too slow to keep up

Previous solutions

- Kernel bypass - move hardware to userspace
  - High performance, but hard to integrate with the system
- Fast-path frame-to-userspace solutions (Netmap etc)
  - Kernel in control, but lower performance
- Custom in-kernel modules (e.g., Open vSwitch)
  - Avoids context switching, but is a maintenance burden

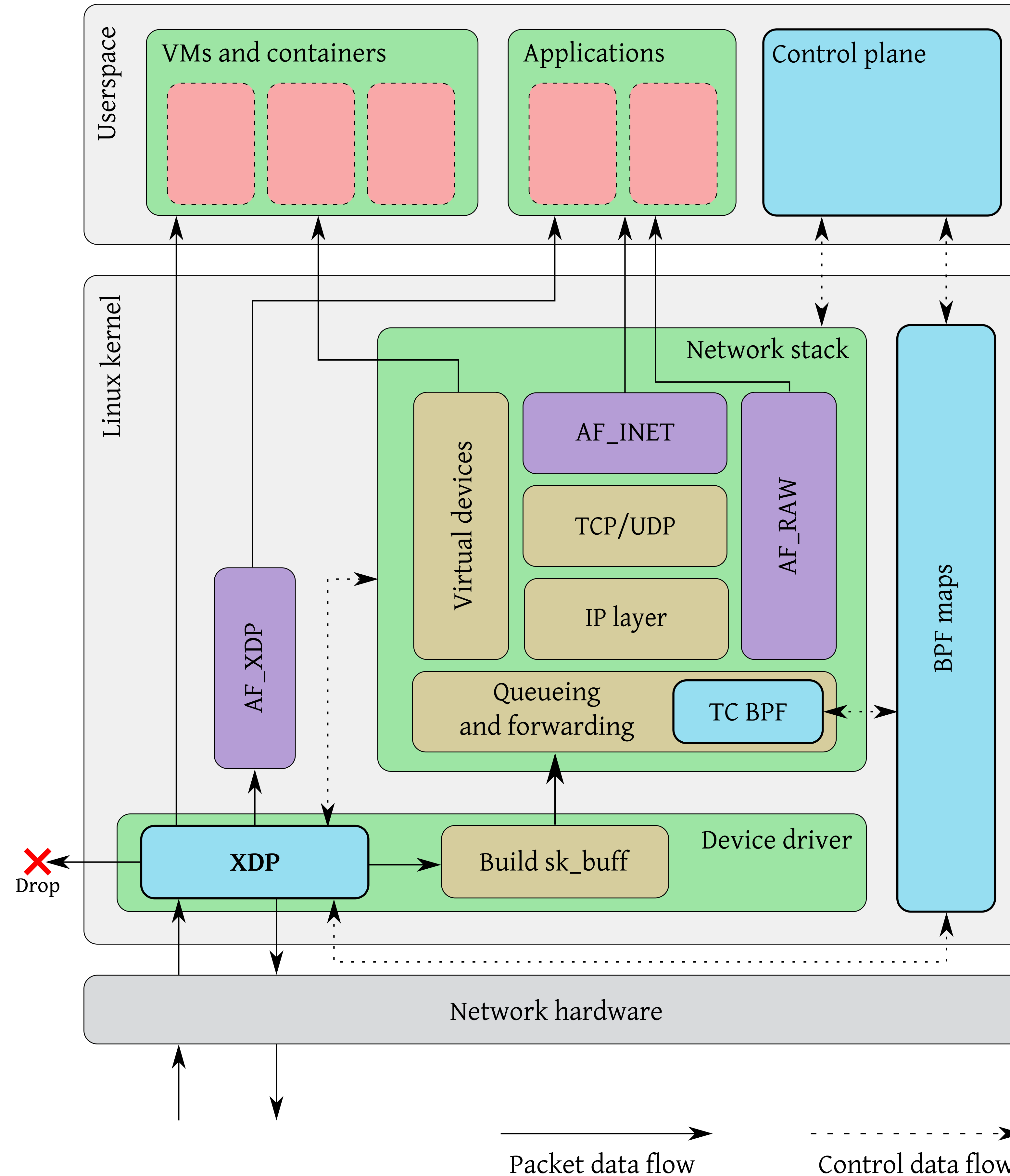
XDP: Move processing into the kernel instead

XDP: Benefits

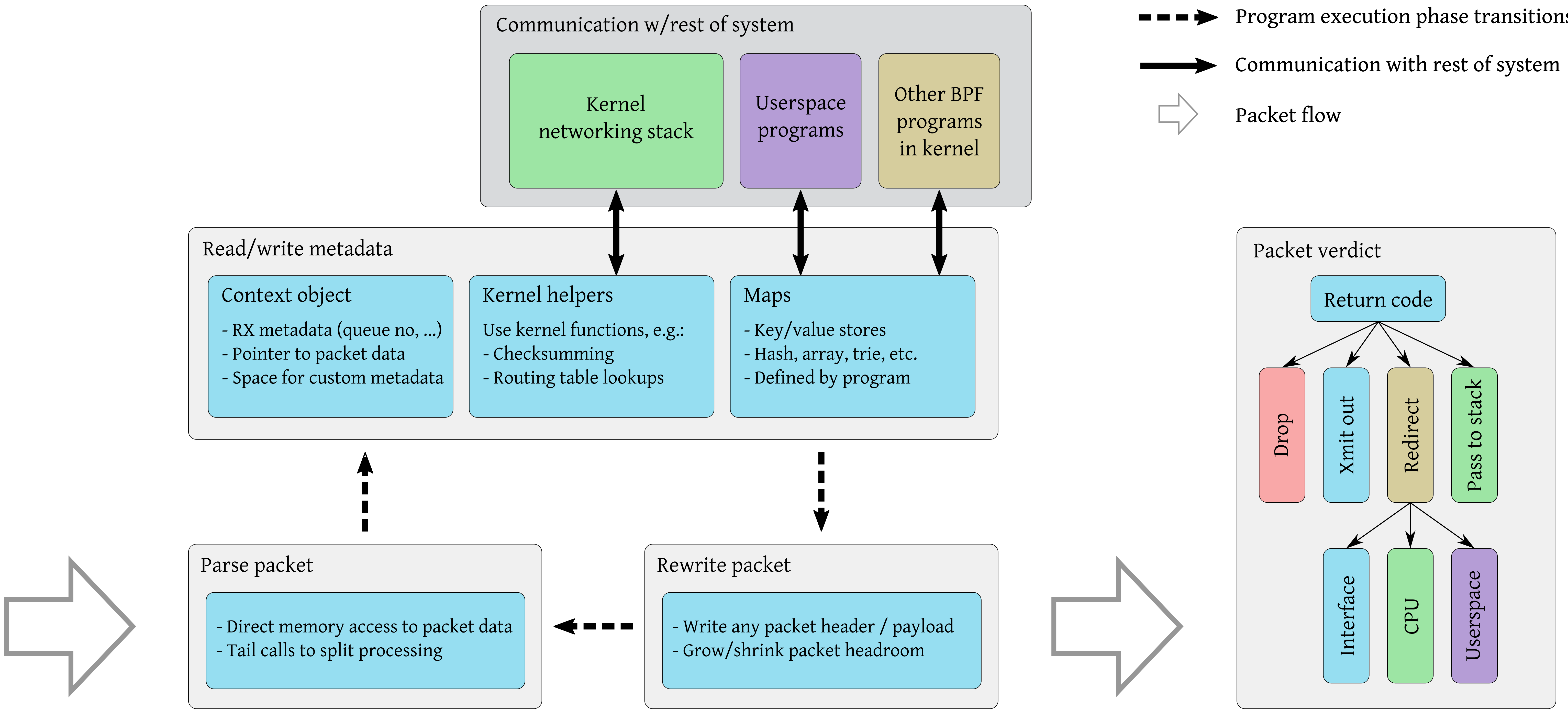
- Integrated with the kernel; driver retains control of hardware
- Can selectively use kernel stack features
- Stable API
- No packet re-injection needed
- Transparent to the host
- Dynamically re-programmable
- Doesn’t need a full CPU core

XDP: Overall design

- The XDP driver hook
- The eBPF virtual machine
- BPF maps
- The eBPF verifier



XDP program flow



Example XDP program

```c
/* map used to count packets; key is IP protocol,
   value is pkt count */
struct bpf_map_def SEC("maps") rxcnt = {
        .type = BPF_MAP_TYPE_PERCPU_ARRAY,
        .key_size = sizeof(u32),
        .value_size = sizeof(long),
        .max_entries = 256,
};

/* swaps MAC addresses using direct packet data access */
static void swap_src_dst_mac(void *data)
{
        unsigned short *p = data;
        unsigned short dst[3];
        dst[0] = p[0];
        dst[1] = p[1];
        dst[2] = p[2];
        p[0]   = p[3];
        p[1]   = p[4];
        p[2]   = p[5];
        p[3]   = dst[0];
        p[4]   = dst[1];
        p[5]   = dst[2];
}

static int parse_ipv4(void *data, u64 nh_off, void *data_end)
{
        struct iphdr *iph = data + nh_off;
        if (iph + 1 > data_end)
                return 0;
        return iph->protocol;
}
```

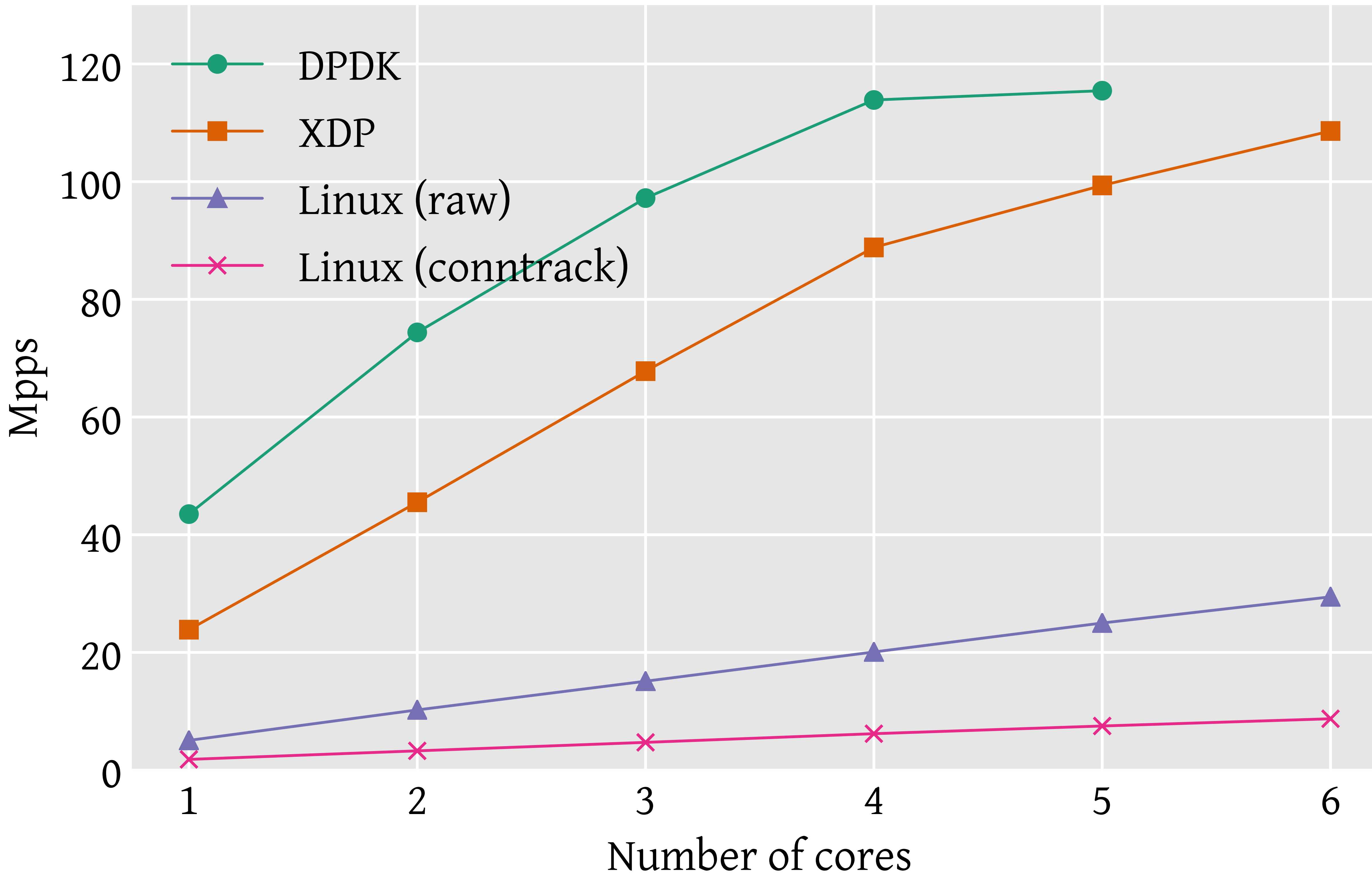
```c
SEC("xdp1") /* marks main eBPF program entry point */
int xdp_prog1(struct xdp_md *ctx)
{
        void *data_end = (void *)(long)ctx->data_end;
        void *data = (void *)(long)ctx->data;
        struct ethhdr *eth = data; int rc = XDP_DROP;
        long *value; u16 h_proto; u64 nh_off; u32 ipproto;

        nh_off = sizeof(*eth);
        if (data + nh_off > data_end)
                return rc;

        h_proto = eth->h_proto;

        /* check VLAN tag; could be repeated to support double-tagged VLAN */
        if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
                struct vlan_hdr *vhdr;

                vhdr = data + nh_off;
                nh_off += sizeof(struct vlan_hdr);
                if (data + nh_off > data_end)
                        return rc;
                h_proto = vhdr->h_vlan_encapsulated_proto;
        }

        if (h_proto == htons(ETH_P_IP))
                ipproto = parse_ipv4(data, nh_off, data_end);
        else if (h_proto == htons(ETH_P_IPV6))
                ipproto = parse_ipv6(data, nh_off, data_end);
        else
                ipproto = 0;

        /* lookup map element for ip protocol, used for packet counter */
        value = bpf_map_lookup_elem(&rxcnt, &ipproto);
        if (value)
                *value += 1;

        /* swap MAC addrs for UDP packets, transmit out this interface */
        if (ipproto == IPPROTO_UDP) {
                swap_src_dst_mac(data);
                rc = XDP_TX;
        }
        return rc;
}
```

Performance benchmarks

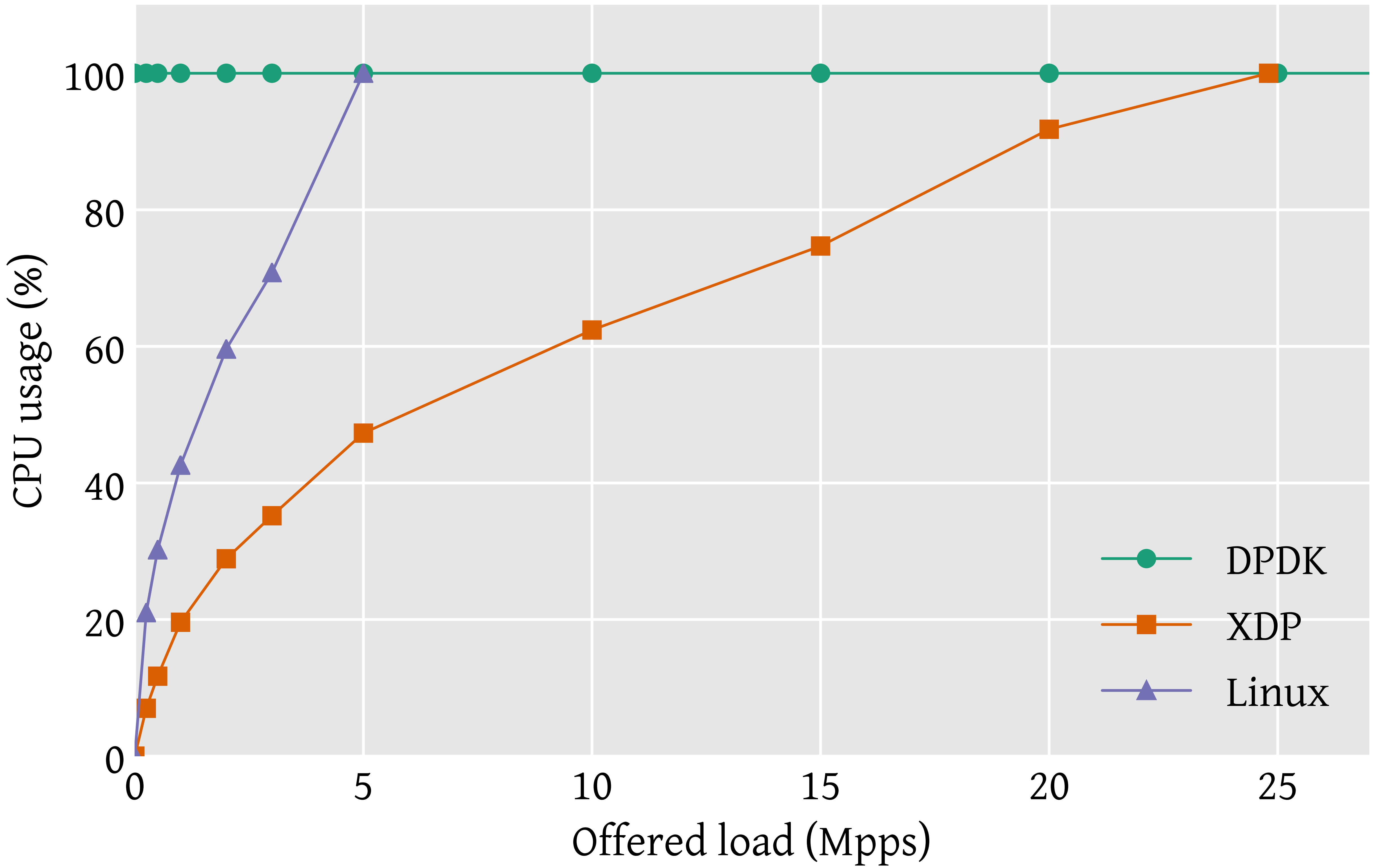
- Benchmark against DPDK
  - Establishes baseline performance
  - Simple tests
    - Packet drop performance
    - CPU usage
    - Packet forwarding performance

All tests are with 64 byte packets - measuring Packets Per Second (PPS).

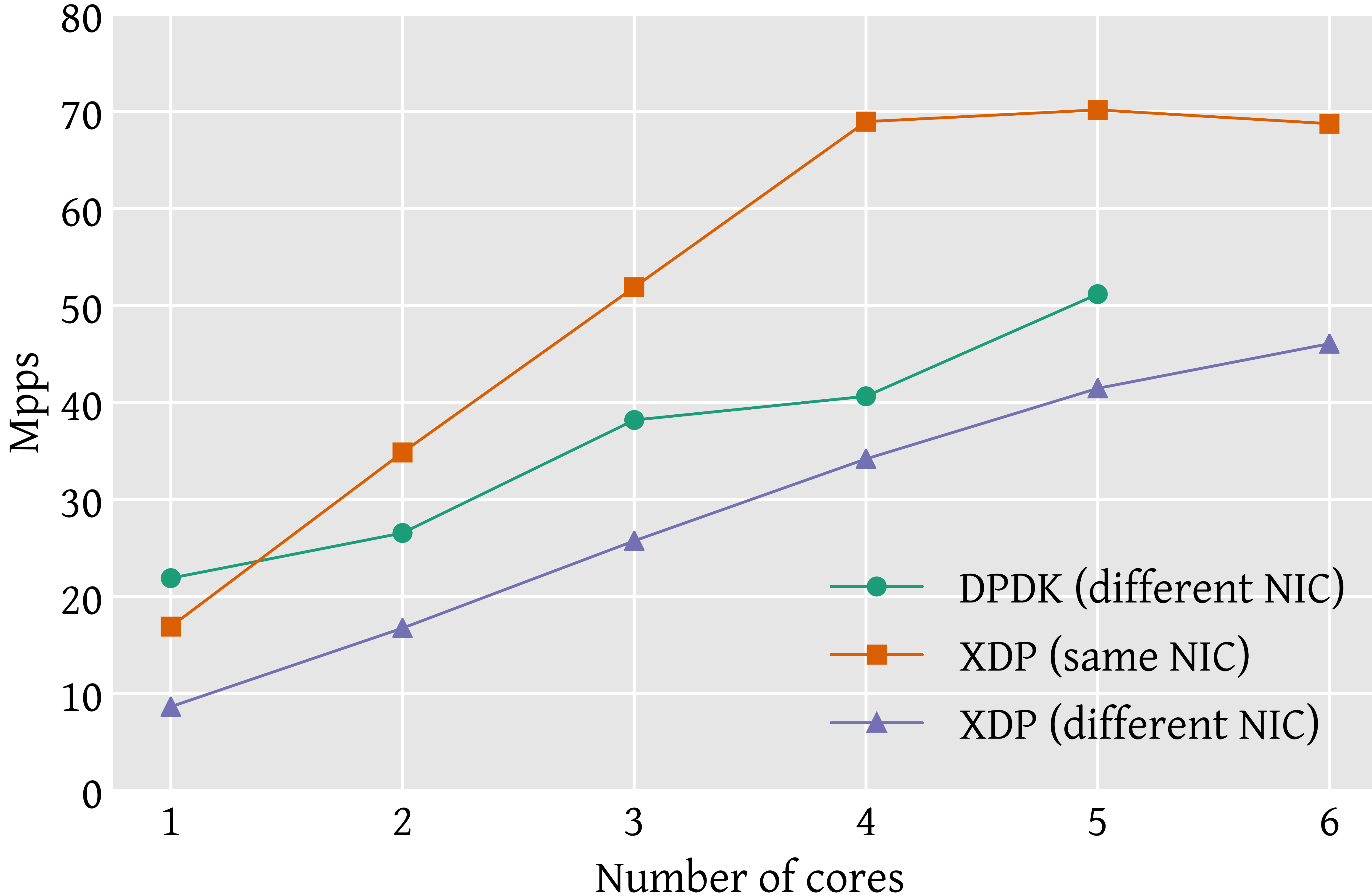
Packet drop performance



CPU usage in drop test



Packet forwarding throughput



Packet forwarding latency

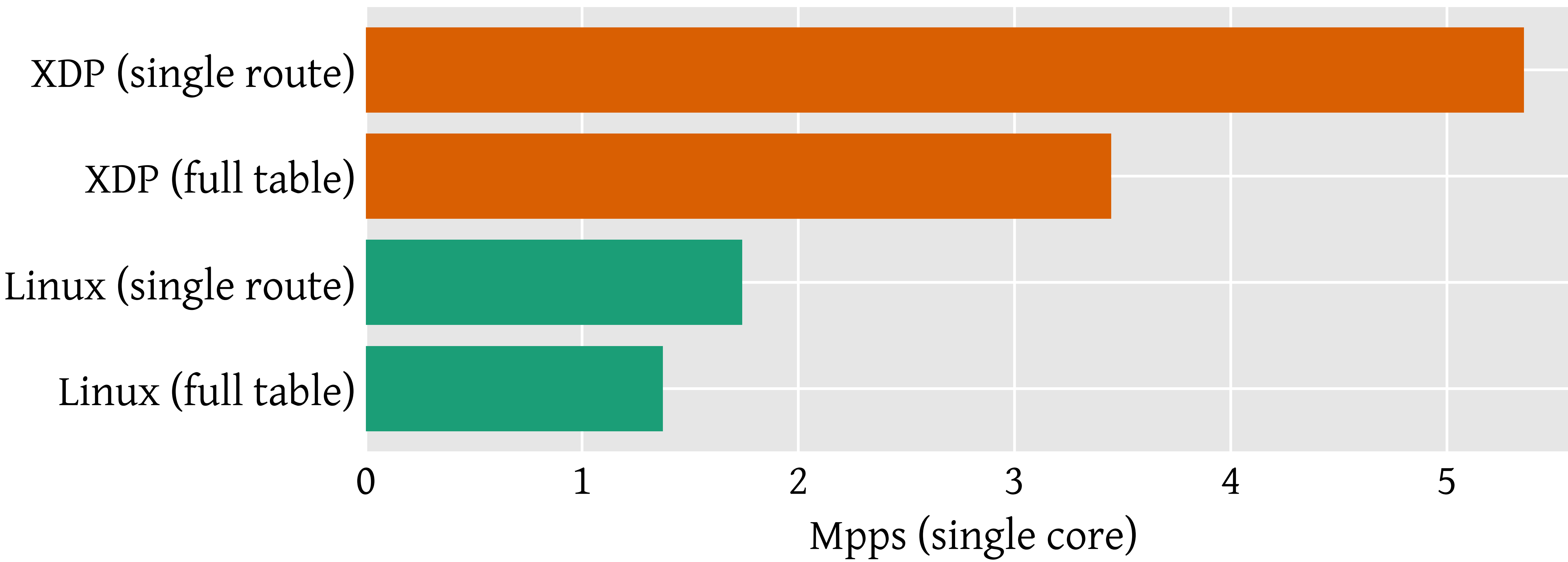
	Average		Maximum		$< 10\mu s$	
	100 pps	1 Mpps	100 pps	1 Mpps	100 pps	1 Mpps
XDP	$82\mu s$	$7\mu s$	$272\mu s$	$202\mu s$	0%	98.1%
DPDK	$2\mu s$	$3\mu s$	$161\mu s$	$189\mu s$	99.5%	99.0%

Application proof-of-concept

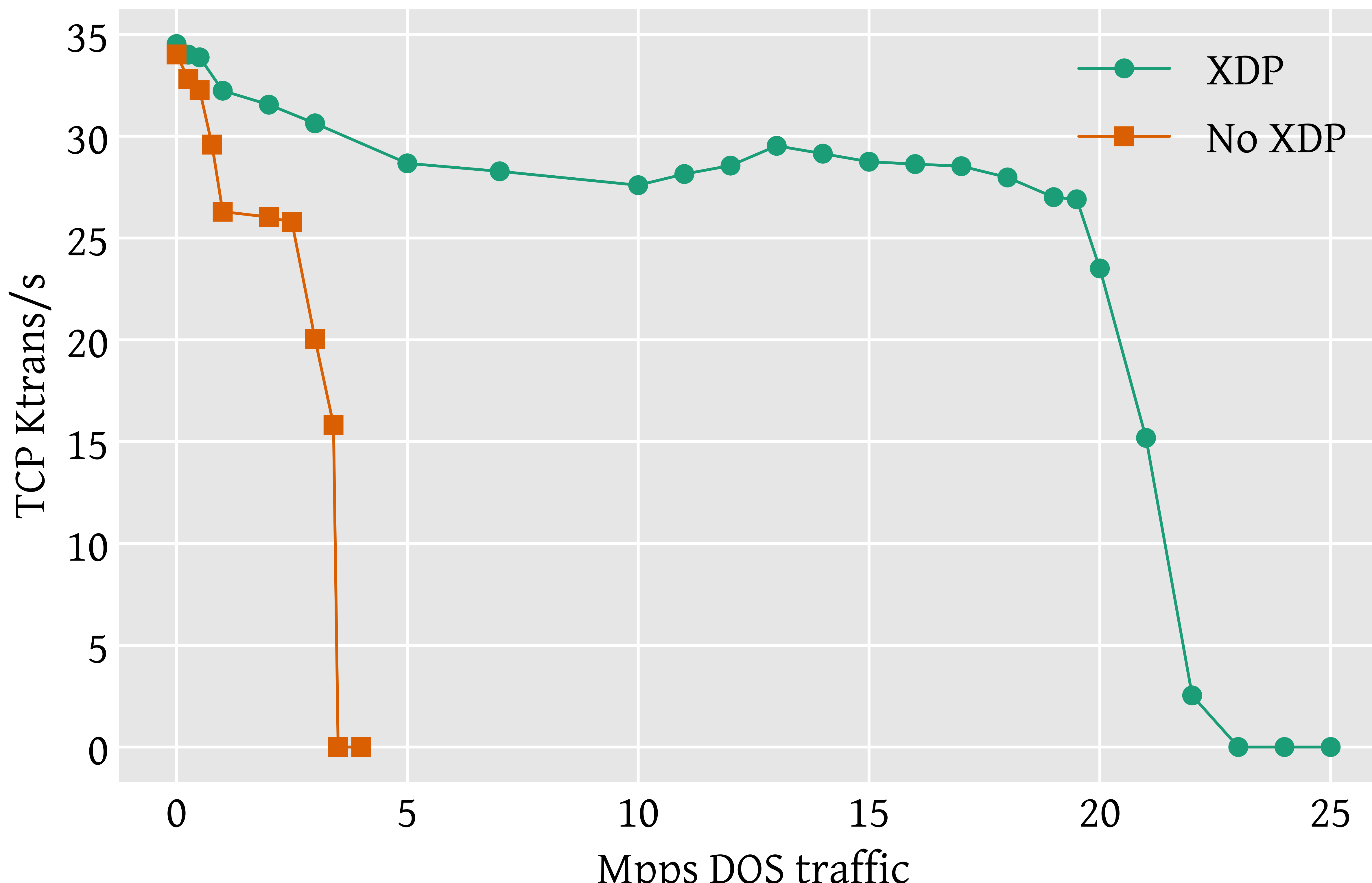
- Shows feasibility of three applications:
  - Software router
  - DDoS protection system
  - Layer-4 load balancer

Not a benchmark against state-of-the-art implementations

Software routing performance



DDoS protection



Load balancer performance

CPU Cores	1	2	3	4	5	6
XDP (Katran)	5.2	10.1	14.6	19.5	23.4	29.3
Linux (IPVS)	1.2	2.4	3.7	4.8	6.0	7.3

Based on the Katran load balancer (open sourced by Facebook).

Summary

XDP:

- Integrates programmable packet processing into the kernel
- Combines speed with flexibility
- Is supported by the Linux kernel community
- Is already used in high-profile production use cases

See https://github.com/tohojo/xdp-paper for details

Acknowledgements

XDP has been developed over a number of years by the Linux networking community. Thanks to everyone involved; in particular, to:

- Alexei Starovoitov for his work on the eBPF VM and verifier
- Björn Töpel and Magnus Karlsson for their work on AF_XDP

Also thanks to our anonymous reviewers, and to our shepherd Srinivas Narayana for their helpful comments on the paper