BP4

***B**eyond **P**acket **P**rocessing towards **P**rotocol **P**rocessing*

Optimizing host networking processing

(applying techniques of P4 to the host networking stack)

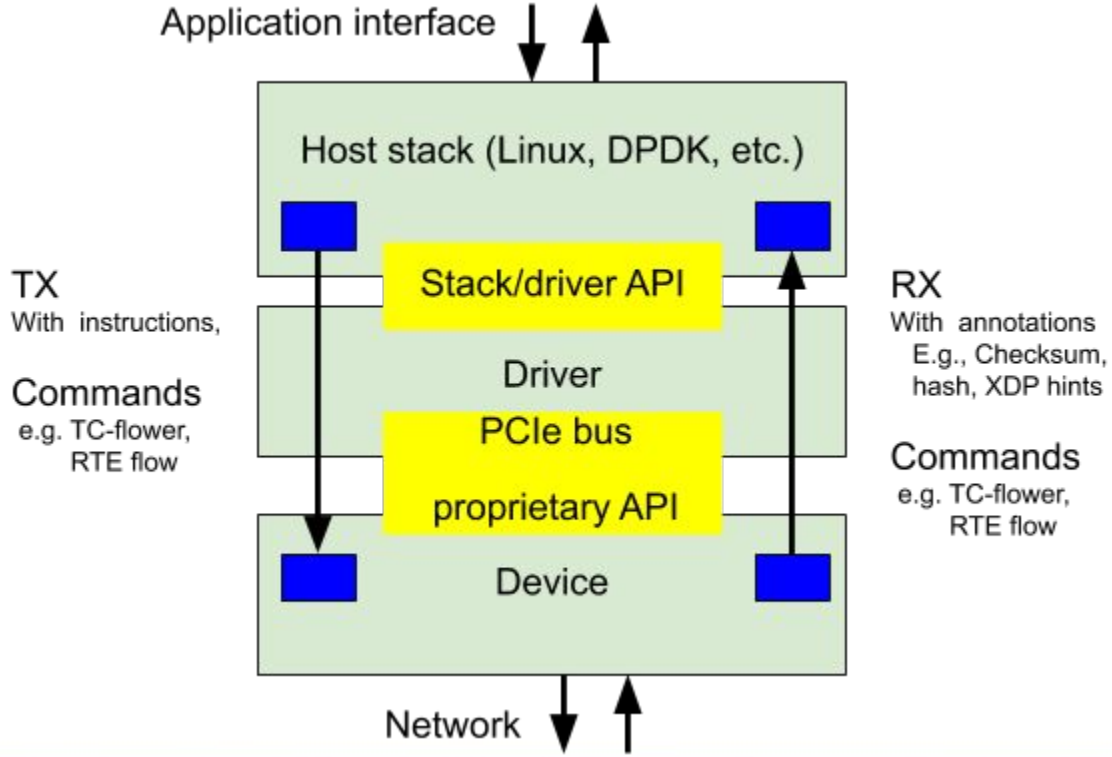
Problems in host networking

- Hosts are not routers
- Hosts concerned with all seven layers of OSI model
- Stateful versus stateless (flow state)
- PDUs not just packets
- Success of HW offload is underwhelming
- Myth of “fast path” and “common case”
- Traditional stack/driver/device model is a bottleneck
- Host stack are in C/C++ (resistance to change)
- Performance+flexibility against conventional wisdom

Host side terminology

- eBPF: generic bytecode ran in kernel
- XDP: prommable fast path in Linux device drivers
- DPDK: userspace raw access to networking queues
- VPP: vector processing in DPDK
- Offload: Device performs host functions (decoupled)
- Acceleration: In-line functional HW interface
- SmartNIC: Host on a NIC

Traditional stack/driver/device model



Host functions to optimize/accelerate

- Checksum, crypto, hashes
- State lookup
- State maintenance
- Segmentation and reassembly (PDU queues)
- Protocol parsing
- Data movement/header-data split
- Header field access.byte swap
- Parallelism

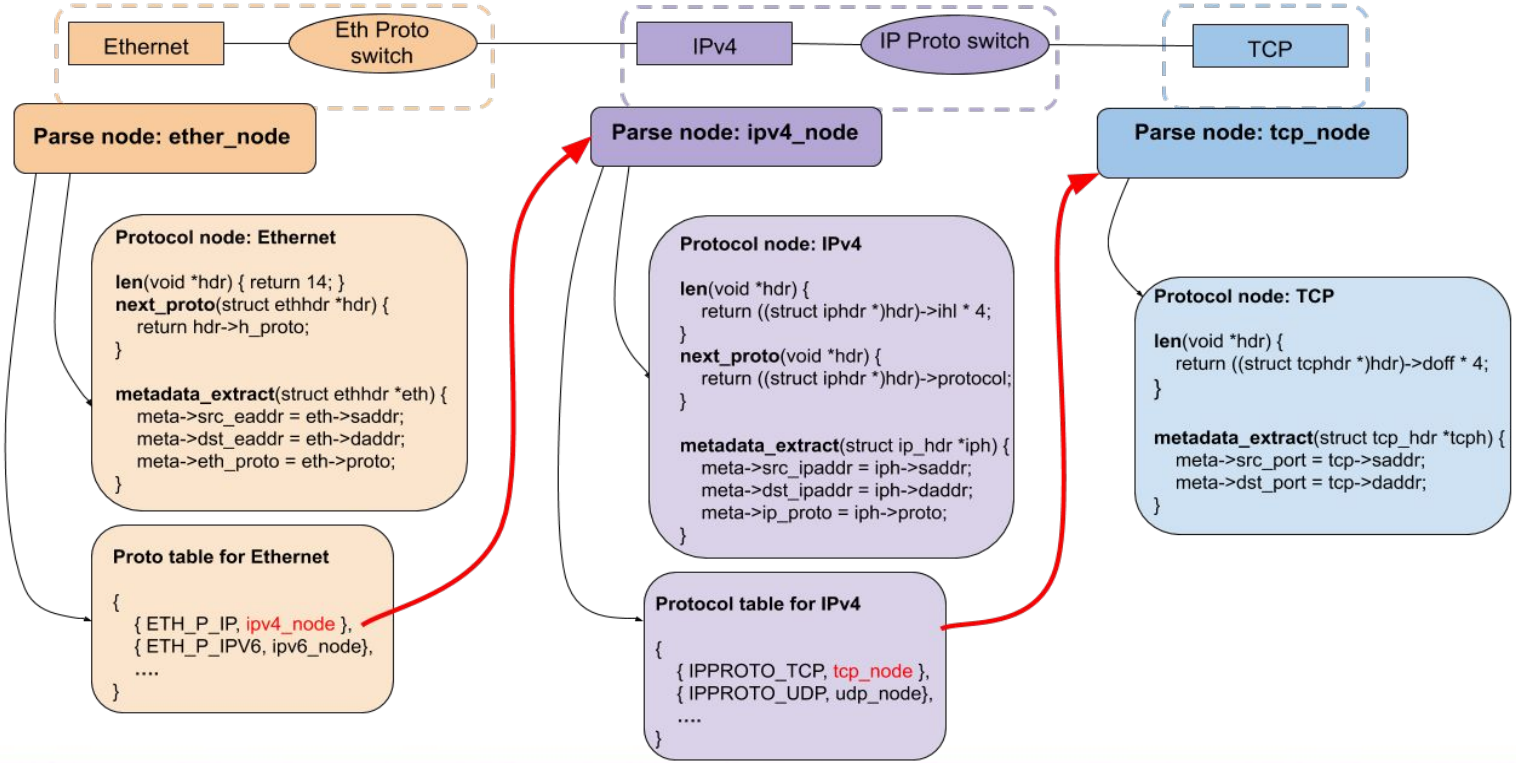
Host functions to optimize/accelerate

- Checksum, crypto, hashes
- State lookup
- State maintenance
- Segmentation and reassembly (PDU queues)
- Protocol parsing
- Data movement/header-data split
- Header field access.byte swap
- Parallelism

BP4 protocol parser (in a C library)

- Declarative representation better than imperative
- parse_pdu function parses PDU. Input is
  - PDU (pointer and length)
  - Parse graph (linked data structures)
  - Tables for next protocol
- Parse graph composed of nodes
  - Functions for proto walk (next protocol, lengths)
  - Functions to extract metadata
  - Functions per process layer processing

Parsing example



OVS parsing example

Virtual switch packets between VMs or the network*

1) Receive packet from network or VM
2) Parse packet and extract metadata
3) Perform some lookup(s) on metadata to get action(s)
4) Execute action(s) like forward to VM host, drop

*Details vary in the protocols an implementation can parse, the lookups performed (like macro flow/macro flow) and the actions performed. Implementations will add their own bells and whistles like analytics, conntrack, mirroring actions, etc.

ovs_parser

```
if (eth_proto == bpf_htons(ETH_P_IP)) {
        struct iphdr nh;

        printt("parse ipv4\n");
        if (skb_load_bytes(skb, offset, &nh, sizeof(nh)) < 0) {
            err = ovs_header_too_short;
            printt("ERR: load byte %d\n", __LINE__);
            goto end;
        }
        offset += nh.ihl * 4;
        hdrs.valid |= IPV4_VALID;

        hdrs.ipv4.ttl = nh.ttl;                /* u8 */
        hdrs.ipv4.tos = nh.tos;                /* u8 */
        hdrs.ipv4.protocol = nh.protocol;     /* u8*/
        hdrs.ipv4.srcAddr = nh.saddr;         /* be32 */
        hdrs.ipv4.dstAddr = nh.daddr;         /* be32 */

        nw_proto = hdrs.ipv4.protocol;
        printt("next proto 0x%x\n", nw_proto);
} else if (eth_proto == bpf_htons(ETH_P_ARP) ||
               eth_proto == bpf_htons(ETH_P_RARP)) {
        struct arp_rarp_t *arp;

        printt("parse arp/rarp\n");

        /* the struct arp_rarp_t is wired format */
        arp = &hdrs.arp;
        if (skb_load_bytes(skb, offset, arp, sizeof(hdrs.arp)) < 0)
{
            err = ovs_header_too_short;
            printt("ERR: load byte %d\n", __LINE__);
            goto end;
        }
        offset += sizeof(hdrs.arp);
        hdrs.valid |= ARP_VALID;

        if (arp->ar_hrd == bpf_htons(ARPHRD_ETHER) &&
            arp->ar_pro == bpf_htons(ETH_P_IP) &&
            arp->ar_hln == ETH_ALEN &&
            arp->ar_pln == 4) {
            printt("valid arp\n");
        } else {
            printt("ERR: invalid arp\n");
        }
        goto parse_metadata;

    }
} else if (eth_proto == bpf_htons(ETH_P_IPV6)) {

        struct ipv6hdr ip6hdr;    /* wired format */

        if (skb_load_bytes(skb, offset, &ip6hdr, sizeof(ip6hdr)) <
0) {
            err = ovs_header_too_short;
            printt("ERR: load byte %d\n", __LINE__);
            goto end;
        }
        offset += sizeof(struct ipv6hdr); /* wired format */
        hdrs.valid |= IPV6_VALID;

        printt("parse ipv6\n");

        memcpy(&hdrs.ipv6.flowLabel, &ip6hdr.flow_lbl, 4); //FIXME
        memcpy(&hdrs.ipv6.srcAddr, &ip6hdr.saddr, 16);
        memcpy(&hdrs.ipv6.dstAddr, &ip6hdr.daddr, 16);

        nw_proto = ip6hdr.nexthdr;

        if (ipv6_has_ext(nw_proto)) {
            printt("WARN: ipv6 nexthdr %x does not supported\n",
nw_proto);
            // need to update offset
        }

        printt("next proto = %x\n", nw_proto);
}
```

ovs_parser

```
if (eth_proto == bpf_htons(ETH_P_IP)) {
        struct iphdr nh;

        printt("parse ipv4\n");
        if (skb_load_bytes(skb, offset, &nh, sizeof(nh)) <
            err = ovs_header_too_short;
            printt("ERR: load byte %d\n", __LINE__);
            goto end;
        }
        offset += nh.ihl * 4;
        hdrs.valid |= IPV4_VALID;

        hdrs.ipv4.ttl = nh.ttl;                /* u8 */
        hdrs.ipv4.tos = nh.tos;                /* u8 */
        hdrs.ipv4.protocol = nh.protocol;     /* u8*/
        hdrs.ipv4.srcAddr = nh.saddr;        /* be32 */
        hdrs.ipv4.dstAddr = nh.daddr;        /* be32 */

        nw_proto = hdrs.ipv4.protocol;
        printt("next proto 0x%x\n", nw_proto);
} else if (eth_proto == bpf_htons(ETH_P_ARP) ||
           eth_proto == bpf_htons(ETH_P_RARP)) {
        struct arp_rarp_t *arp;

        printt("parse arp/rarp\n");

        /* the struct arp_rarp_t is wired format */
        arp = &hdrs.arp;
        if (skb_load_bytes(skb, offset, arp, sizeof(hdrs.arp)) < 0)
{
            err = ovs_header_too_short;
            printt("ERR: load byte %d\n", __LINE__);
            goto end;
        }
        offset += sizeof(hdrs.arp);
        hdrs.valid |= ARP_VALID;
```

Length checks in red

Header extraction in blue

Protocol switch code in green

```
if (arp->ar_hrd == bpf_htons(ARPHRD_ETHER) &&
    arp->ar_pro == bpf_htons(ETH_P_IP) &&
    arp->ar_hln == ETH_ALEN &&
    arp->ar_pln == 4) {
    printt("valid arp\n");
                 : invalid arp\n");
                 data;
}
} else if (eth_proto == bpf_htons(ETH_P_IPV6)) {

        struct ipv6hdr ip6hdr;     /* wired format */

        if (skb_load_bytes(skb, offset, &ip6hdr, sizeof(ip6hdr)) <
0) {
            err = ovs_header_too_short;
                 RR: load byte %d\n", __LINE__);

                 eof(struct ipv6hdr); /* wired format */
                 IPV6_VALID;

        printt("parse ipv6\n");

        memcpy(&hdrs.ipv6.flowLabel, &ip6hdr.flow_lbl, 4); //FIXME
        memcpy(&hdrs.ipv6.srcAddr, &ip6hdr.saddr, 16);
        memcpy(&hdrs.ipv6.dstAddr, &ip6hdr.daddr, 16);

        nw_proto = ip6hdr.nexthdr;

        if (ipv6_has_ext(nw_proto)) {
            printt("WARN: ipv6 nexthdr %x does not supported\n",
nw_proto);
            // need to update offset
        }

        printt("next proto = %x\n", nw_proto);
}
```

P4 parser for OVS

```
#define ETH_P_8021Q     0x8100 /* 802.1Q VLAN Extended Header  */
#define ETH_P_8021AD    0x88A8 /* 802.1ad Service VLAN     */
#define ETH_P_ARP       0x0806
#define ETH_P_IPV4      0x0800
#define ETH_P_IPV6      0x86DD

header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

parser parse_ethernet{
        extract(ethernet);
        return select(latest.etherType) {
        ETH_P_8021Q: parse_vlan;
        ETH_P_8021AD: parse_vlan;
                ETH_P_ARP: parse_arp;
                ETH_P_IPV4: parse_ipv4;
                ETH_P_IPV6: parse_ipv6;
                default: ingress;
        }
}

parser parse_ipv4 {
        extract(ipv4);
        return select(latest.protocol) {
                IPPROTO_TCP: parse_tcp;
                IPPROTO_UDP: parse_udp;
                IPPROTO_ICMP: parse_icmp;
                default: ingress;
        }
}
```

```
parser parse_ipv6 {
        extract(ipv6);
        return select(latest.nextHdr) {
                IPPROTO_TCP: parse_tcp;
                IPPROTO_UDP: parse_udp;
                IPPROTO_ICMP: parse_icmp;
                default: ingress;
        }
}

table ovs_tbl {
        reads {
                /* Avoid compiler optimizes out, although
           we are not using it at all */
        ethernet.dstAddr: exact;
        vlan.etherType: exact;
                ipv4.dstAddr: exact;
        ipv6.dstAddr: exact;
        icmp.typeCode: exact;
        tcp.dstPort: exact;
        udp.dstPort: exact;
                md.in_port: exact;
                tnl_md.tun_id: exact;
        }
        actions {
                nop;
        }
}
```

P4 parser for OVS

```
#define ETH_P_8021Q     0x8100 /* 802.1Q VLAN Extended Header  */
#define ETH_P_8021AD    0x88A8 /* 802.1ad Service VLAN     */
#define ETH_P_ARP       0x0806
#define ETH_P_IPV4      0x0800
#define ETH_P_IPV6      0x86DD

header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

parser parse_ethernet{
        extract(ethernet);
        return select(latest.etherType) {
        ETH_P_8021Q: parse_vlan;
        ETH_P_8021AD: parse_vlan;
                ETH_P_ARP: parse_arp;
                ETH_P_IPV4: parse_ipv4;
                ETH_P_IPV6: parse_ipv6;
                default: ingress;
        }
}

parser parse_ipv4 {
        extract(ipv4);
        return select(latest.protocol) {
                IPPROTO_TCP: parse_tcp;
                IPPROTO_UDP: parse_udp;
                IPPROTO_ICMP: parse_icmp;
                default: ingress;
        }
}
```

Header definitions

Parser and switch

No IPv4 header length handling?

```
parser parse_ipv6 {
        extract(ipv6);
        return select(latest.nextHdr) {
                IPPROTO_TCP: parse_tcp;
                IPPROTO_UDP: parse_udp;
                IPPROTO_ICMP: parse_icmp;
                default: ingress;
        }
}

table ovs_tbl {
        reads {
                /* Avoid compiler optimizes out, although
                   we are not using it at all */
        ethernet.dstAddr: exact;
        vlan.etherType: exact;
                ipv4.dstAddr: exact;
        ipv6.dstAddr: exact;
        icmp.typeCode: exact;
        tcp.dstPort: exact;
        udp.dstPort: exact;
                md.in_port: exact;
                tnl_md.tun_id: exact;
        }
        actions {
                nop;
        }
}
```

Metadata

BP4 parser

```
static int ipv4_proto_nofrag(const iphdr *iph)
{
        if (ip_is_fragment(iph)) {
                /* Don't parse into any fragments */
                return BP4_STOP_OKAY;
        }

        return iph->protocol;
}

static size_t ipv4_len(const iphdr *iph)
{
        return iph->ihl * 4;
}

static int ipv6_proto(struct ipv6hdr *ip6h)
{
        return ip6h->nexthdr;
}


struct bp4_proto_node parse_ipv4_no_frag = {
        .name = "IPv4 no frag",
        .min_len = sizeof(struct iphdr),
        .ops.len = ipv4_len,
        .ops.next_proto = ipv4_proto_nofrag,
};

struct bp4_proto_node parse_ipv6 = {
        .name = "IPv6",
        .min_len = sizeof(struct ipv6hdr),
        .ops.next_proto = ipv6_proto,
};
```

```
static void ether_meta_data(const struct ethhdr *eth,
                            struct meta_data_frame *frame)
{
          frame->eth_addrs.dst = eth->h_dest;
          frame->eth_addrs.src = eth->h_src;
}

static void ipv4_meta_data(const struct iphdr *iph,
                           struct meta_data_frame *frame)
{
        frame->is_fragment = ip_is_fragment(iph);
        frame->ttl = iph->ttl;
        frame->nproto = iph->protocol;
        frame->addr_type = BP4_KEY_IPV4_ADDRS;
        frame->v4_addrs,src = iph->saddr;
        frame->v4_addrs,DST = iph->Daddr;
}

static void ipv4_meta_data(const struct ipv6hdr *ip6h,
                           struct meta_data_frame *frame)
{
        frame->ttl = ip6h->hop_cnt;
        frame->nproto = ip6h->next_proto;
        frame->addr_type = BP4_KEY_IPV6_ADDRS;
        frame->v4_addrs,src = ip6h->saddr;
        frame->v4_addrs,DST = ip6h->Daddr;
}
struct bp4_entry ether_table[] = {
        { ETH_P_IP, &ipv4_node },
        { ETH_P_IPV6, &ipv6_node },
        { -1, NULL },
};


BP4_MAKE_PARSE_NODE(ether_node, parse_ether, ether_meta_data, NULL,
                    ether_table);
BP4_MAKE_PARSE_NODE(ipv4_node, parse_ipv4, ipv4_meta_data, NULL,
ipv4_table);
BP4_MAKE_PARSE_NODE(ipv6_node, parse_ipv6, ipv6_meta_data, NULL,
ipv6_table);
```

BP4 parser

From BP4 parser library

```
static int ipv4_proto_nofrag(const iphdr *iph)
{
        if (ip_is_fragment(iph)) {
                /* Don't parse into any fragments */
                return BP4_STOP_OKAY;
        }

        return iph->protocol;
}

static size_t ipv4_len(const iphdr *iph)
{
        return iph->ihl * 4;
}

static int ipv6_proto(struct ipv6hdr *ip6h)
{
        return ip6h->nexthdr;
}


struct bp4_proto_node parse_ipv4_no_frag = {
        .name = "IPv4 no frag",
        .min_len = sizeof(struct iphdr),
        .ops.len = ipv4_len,
        .ops.next_proto = ipv4_proto_nofrag,
};

struct bp4_proto_node parse_ipv6 = {
        .name = "IPv6",
        .min_len = sizeof(struct ipv6hdr),
        .ops.next_proto = ipv6_proto,
};
```

Next protocol

IPv4 header length arithmetic

Protocol nodes

```
: void ether_meta_data(const struct ethhdr *eth,
                       struct meta_data_frame *frame)
{
        frame->eth_addrs.dst = eth->h_dest;
        frame->eth_addrs.src = eth->h_src;
}

static void ipv4_meta_data(const struct iphdr *iph,
                           struct meta_data_frame *frame)
{
        ragment = ip_is_fragment(iph);
        = iph->ttl;
        to = iph->protocol;
        _type = BP4_KEY_IPV4_ADDRS;
        ddrs,src = iph->saddr;
        ddrs,DST = iph->Daddr;
}

meta_data(const struct ipv6hdr *ip6h,
                           struct meta_data_frame *frame)

l = ip6h->hop_cnt;
roto = ip6h->next_proto;
dr_type = BP4_KEY_IPV6_ADDRS;
frame->v4_addrs,src = ip6h->saddr;
frame->v4_addrs,DST = ip6h->Daddr;
}
struct bp4_entry ether_table[] = {
P, &ipv4_node },
PV6, &ipv6_node },
L },


BP4_MAKE_PARSE_NODE(ether_node, parse_ether, ether_meta_data, NULL,
                    ether_table);
BP4_MAKE_PARSE_NODE(ipv4_node, parse_ipv4, ipv4_meta_data, NULL,
ipv4_table);
BP4_MAKE_PARSE_NODE(ipv6_node, parse_ipv6, ipv6_meta_data, NULL,
ipv6_table);
```

BP4 parser

From BP4 parser library

```
static int ipv4_proto_nofrag(const iphdr *iph)
{
        if (ip_is_fragment(iph)) {
                /* Don't parse into any fragments */
                return BP4_STOP_OKAY;
        }

        return iph->protocol;
}

static size_t ipv4_len(const iphdr *i
{
        return iph->ihl * 4;
}

static int ipv6_proto(struct ipv6hdr *ip6h)
{
        return ip6h->nexthdr;
}


struct bp4_proto_node parse_ipv4_no
        .name = "IPv4 no frag",
        .min_len = sizeof(struct ip
        .ops.len = ipv4_len,
        .ops.next_proto = ipv4_prot
};

struct bp4_proto_node parse_ipv6 = {
        .name = "IPv6",
        .min_len = sizeof(struct ip
        .ops.next_proto = ipv6_prot
};
```

Metadata extraction

Protocol switch tables

Build parse nodes, parse graph

```
static void ether_meta_data(const struct ethhdr *eth,
                            struct meta_data_frame *frame)
{
        frame->eth_addrs.dst = eth->h_dest;
        frame->eth_addrs.src = eth->h_src;
}

static void ipv4_meta_data(const struct iphdr *iph,
                           struct meta_data_frame *frame)
{
        frame->is_fragment = ip_is_fragment(iph);
        frame->ttl = iph->ttl;
        frame->nproto = iph->protocol;
        frame->addr_type = BP4_KEY_IPV4_ADDRS;
        frame->v4_addrs,src = iph->saddr;
        frame->v4_addrs,DST = iph->Daddr;
}

static void ipv4_meta_data(const struct ipv6hdr *ip6h,
                           struct meta_data_frame *frame)
{
        frame->ttl = ip6h->hop_cnt;
        frame->nproto = ip6h->next_proto;
        frame->addr_type = BP4_KEY_IPV6_ADDRS;
        frame->v4_addrs,src = ip6h->saddr;
        frame->v4_addrs,DST = ip6h->Daddr;
}
struct bp4_entry ether_table[] = {
        { ETH_P_IP, &ipv4_node },
        { ETH_P_IPV6, &ipv6_node },
        { -1, NULL },
};


BP4_MAKE_PARSE_NODE(ether_node, parse_ether, ether_meta_data, NULL,
                    ether_table);
BP4_MAKE_PARSE_NODE(ipv4_node, parse_ipv4, ipv4_meta_data, NULL,
ipv4_table);
BP4_MAKE_PARSE_NODE(ipv6_node, parse_ipv6, ipv6_meta_data, NULL,
ipv6_table);
```

End slide