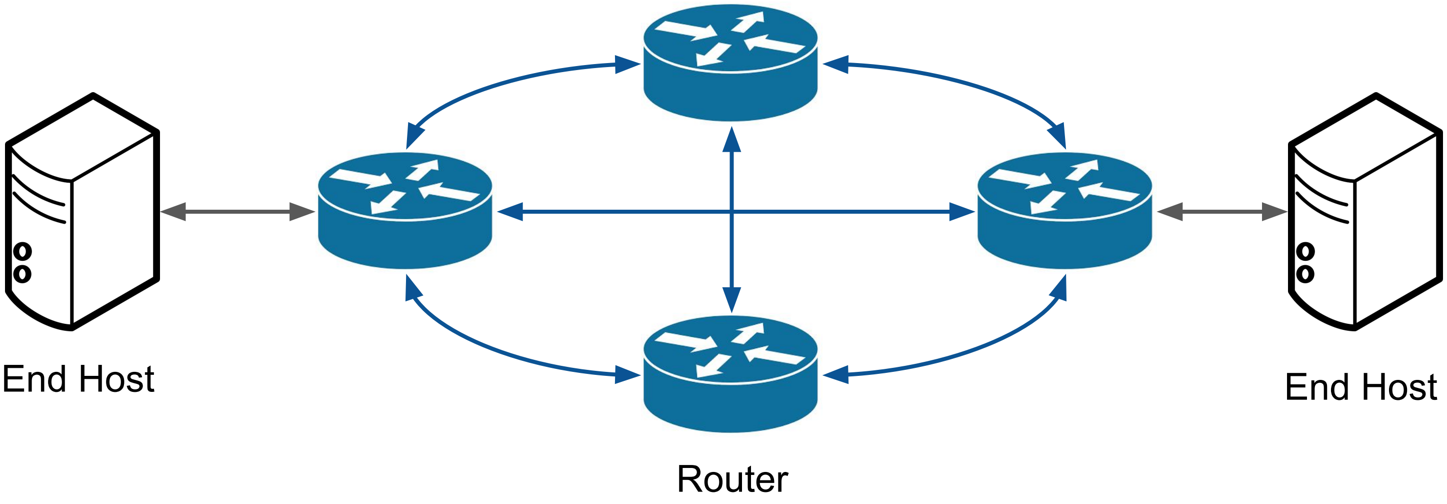
Towards a programmable network

CS 695

Department of Computer Science & Engineering
Indian Institute of Technology Bombay

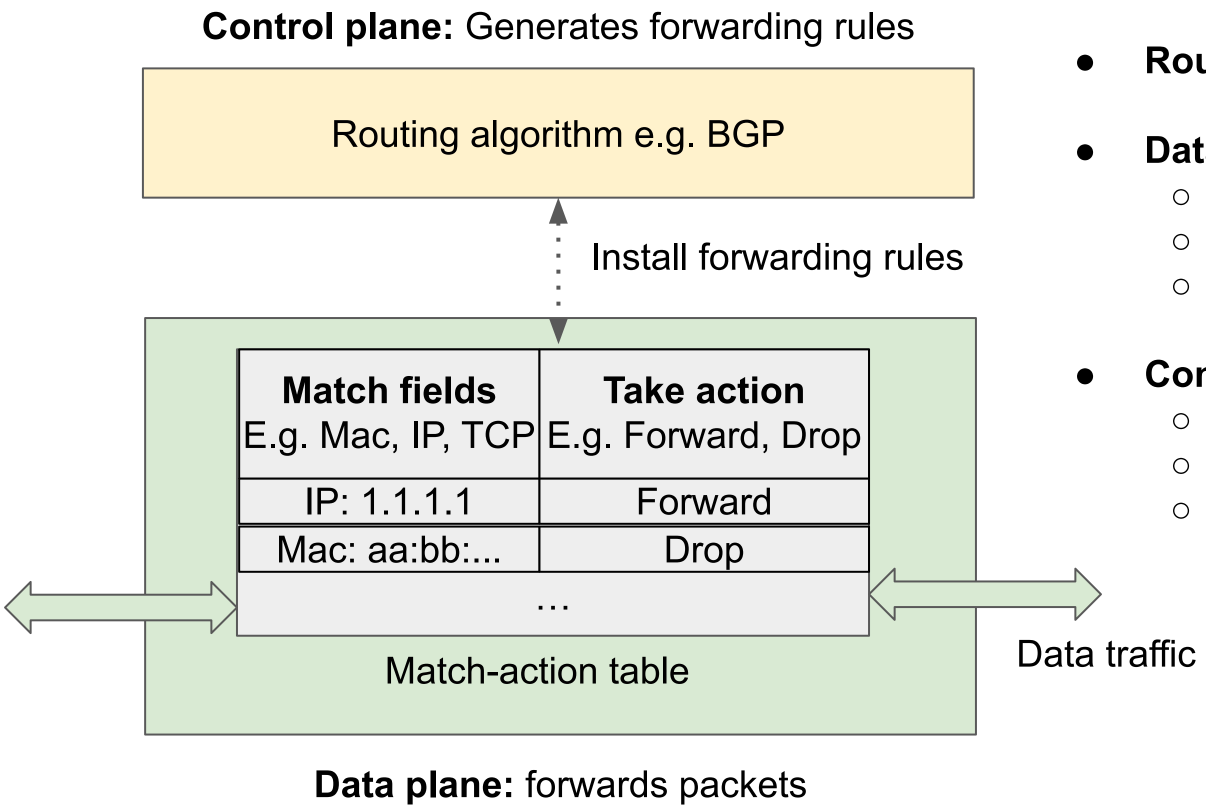
By, Abhik Bose
Guided By: **Prof. Purushottam Kulkarni**
Mar 10, 2023

Computer network overview



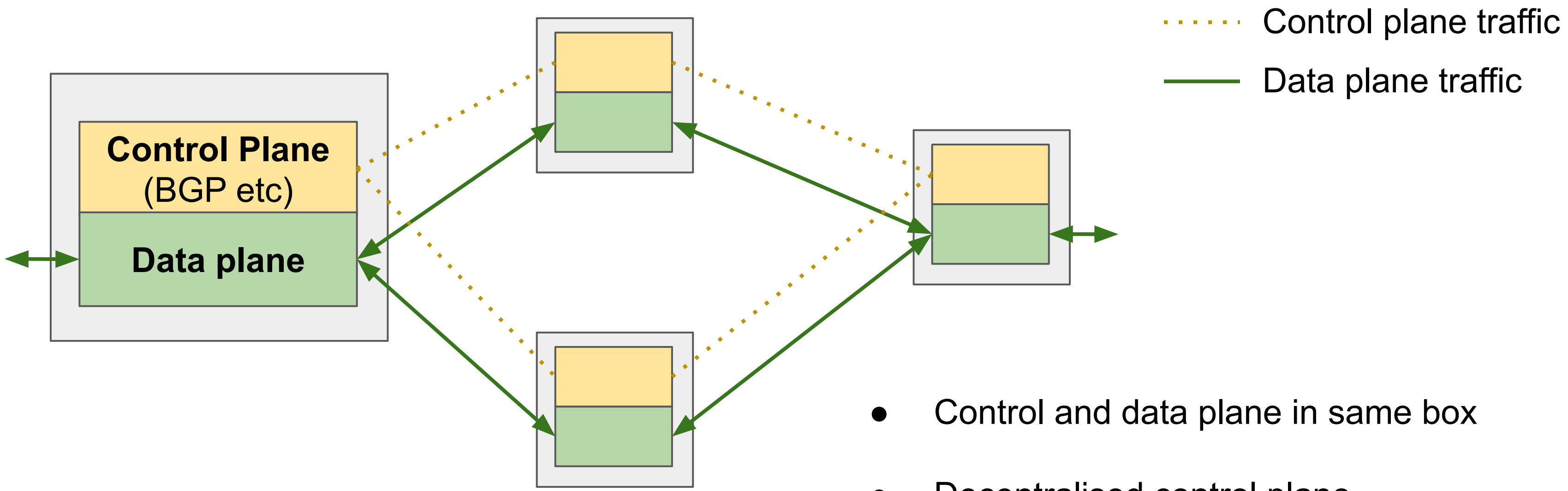
Routers connect end hosts and forward data packets at a high rate

Work of network routers: Control plane and data plane



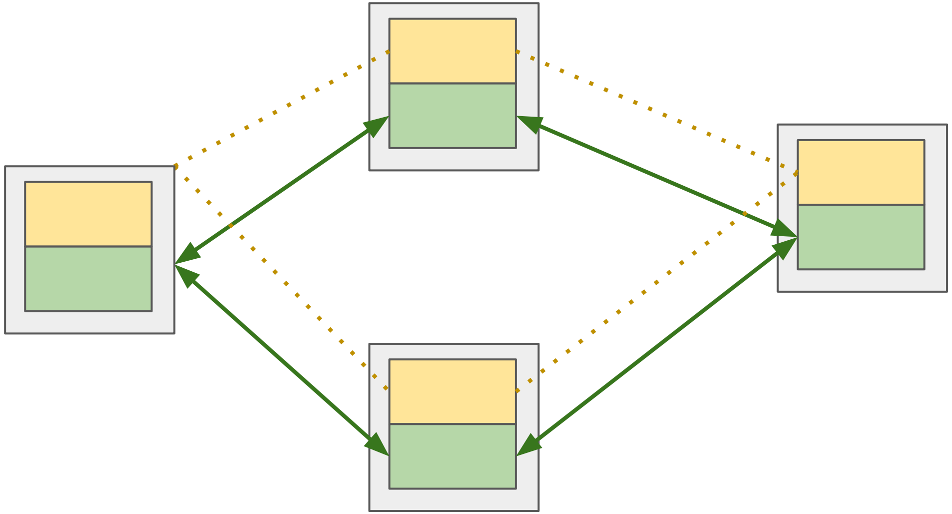
- **Router**: Control plane and data plane
- **Data plane:**
  - Forwards data
  - Match action table
  - Match packet headers, call action
- **Control plane:**
  - Run routing protocols
  - Generates match-action rules
  - Installs rules in data plane

Traditional computer network architecture



- Control and data plane in same box
- Decentralised control plane
  - Communicates using open source protocol e.g. BGP
- Proprietary control and data plane implementation

Traditional computer network limitations



Implement a new control plane protocol for intrusion detection (Say IGP)

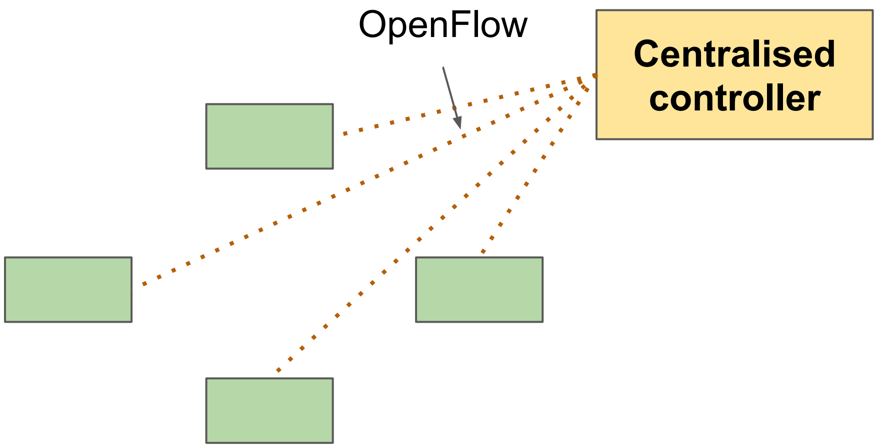
- Write IGP in all vendor specific languages
- Upload IGP to all routers
- Develop an inter router control communication protocol for IGP

Difficulties

- Time consuming, error prone, downtime
- IGP for new vendor X must be written before using their switch
- Resource limitation on router control plane
- All vendors may not support writing such new control plane protocol

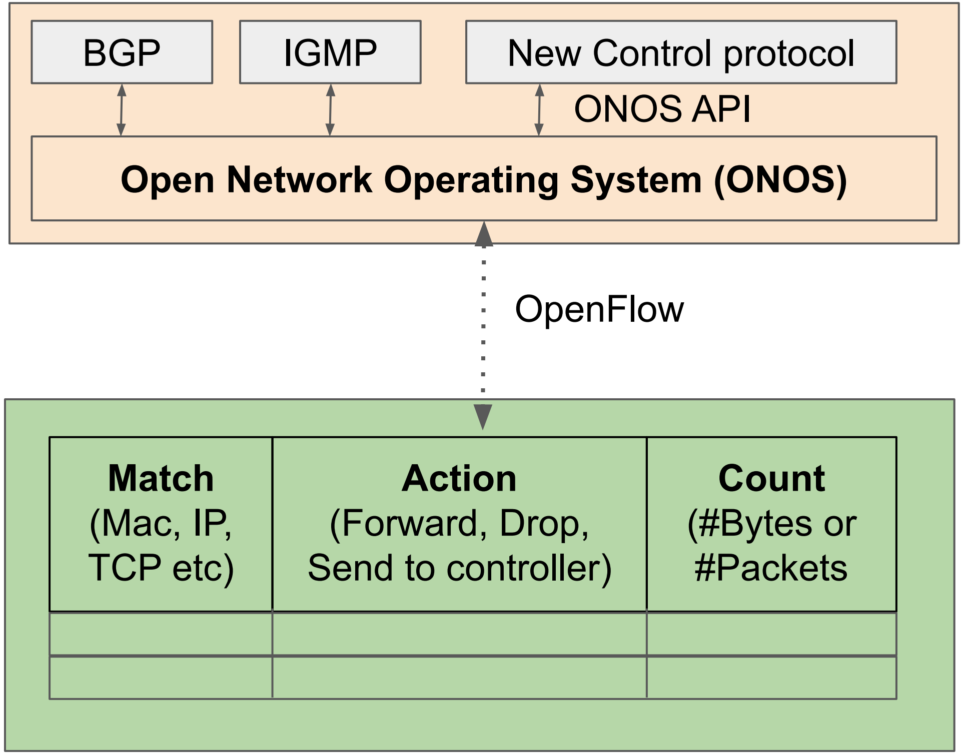
Time consuming, error prone and difficult to scale

Software Defined Network (SDN) key principles



- Control and data plane physically separated
- Centralised network controller
- Open source communication protocol (OpenFlow) for control and data plane communication

Control and data plane communication protocol (OpenFlow)



SDN compatible switch

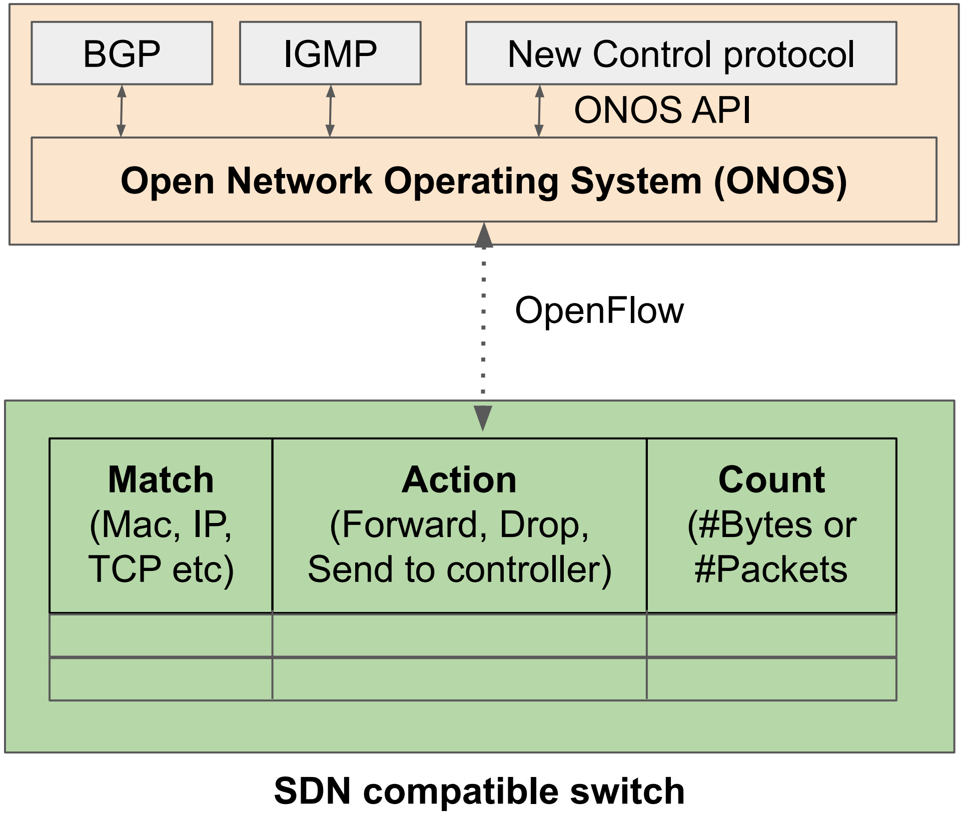
SDN compatible switch

- Can match standard fields
- Action: Forward, drop or send to controller
- Statistics: packet and byte count for each rule

Centralised controller

- Commonly runs ONOS
- Configures rules and acquires statistics from and to data plane using OpenFlow
- Control plane applications are written using ONOS API

Benefits of SDN



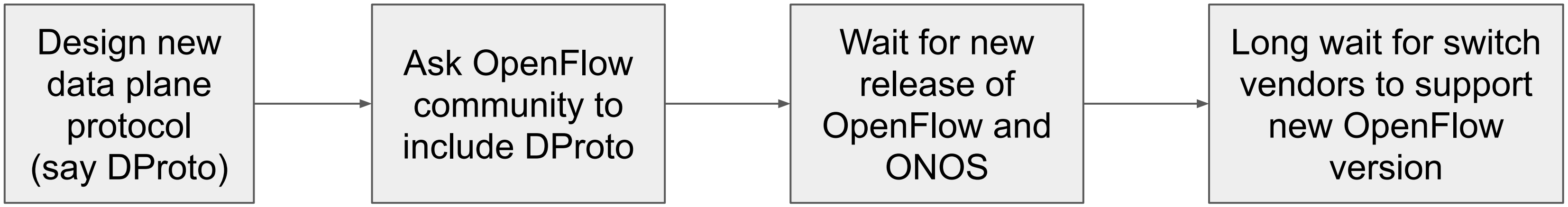
- All network applications can be written using ONOS API
- All control protocols can access statistics acquired by ONOS. Reduce control traffic
- Scalable controller (on cloud)
- Global network view at controller
- Easy to develop, maintain new control plane protocols
- No update at SDN switch for new control protocols
- Easy to add more switches (scalable)
- Less downtime
- Network vendors don't need to open source their SDN switch implementation

Limitations of SDN and solution approaches

- Single point of failure at controller
  - Use fault tolerant hardware e.g. RAID based disk
  - Cloud based controller, use VM based failure handling
  - Open research area

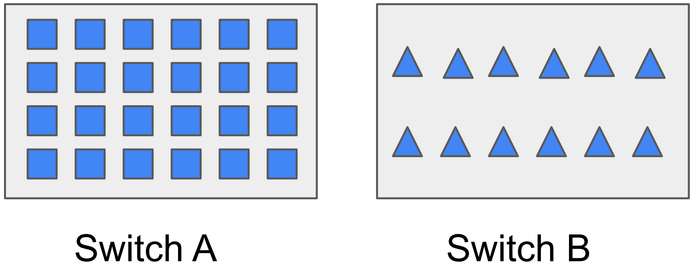
- Data plane is still not programmable (Next Slide..)

Let’s add a new data plane protocol to SDN



- OpenFlow initially released with 12 protocols support, expanded to 46 within 4 years with many releases
- SDN dataplane is not scalable
- **Solution:** Let’s make the dataplane programmable too

Need for a high level data plane programming language



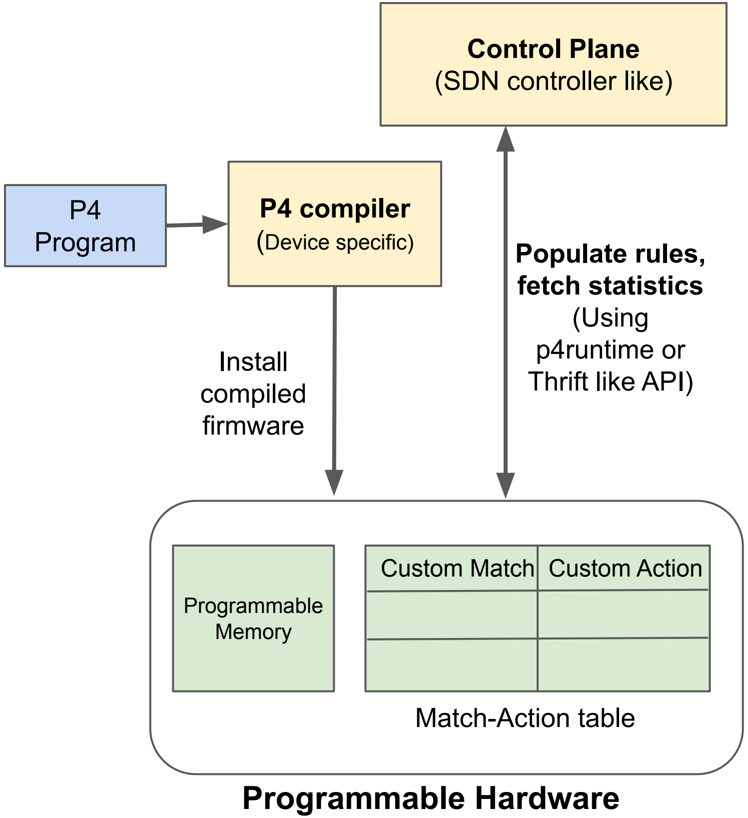
- Network devices have different architectures
  - E.g. ASIC, FPGA, SoC etc
- Programming them using device specific language is difficult

Why not C/C++, JAVA, Python?

- All features supported by them can't be implemented at data plane
- A data plane specific language is more efficient

Solution: A new language namely **Programming Protocol-independent Packet Processors (P4)**

Programmable data plane approach and P4

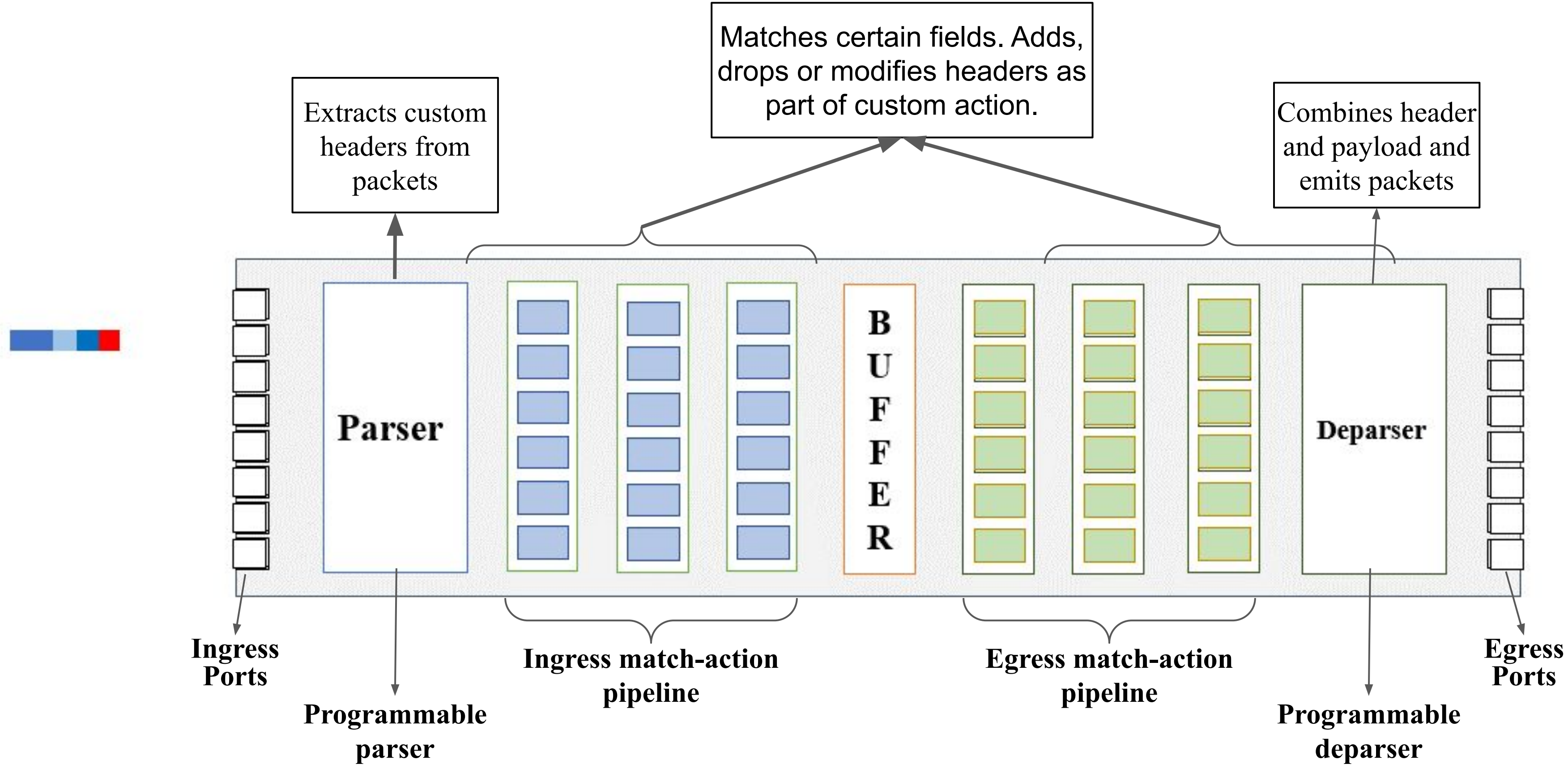


- **P4 Programmable hardware**
- **Features**
  - Custom Header parsing, custom match action
  - On-NIC programmable memory
  - Custom computation
  - Device specific features
- **P4 runtime or other APIs to configure custom match action tables at runtime**

Pros: Offloading application processing to programmable hardware is cost effective and improves performance

Limitations: Limited expressiveness, limited memory

P4 portable switch architecture



P4 Programming example (Continued..)

```
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> ethType;
}

header arp_t {
    bit<16> htype;
    bit<16> ptype;
    bit<8> hp_addr_len;
    bit<8> protocol_len;
    bit<16> op_code;
    bit<48> senderMac;
    bit<32> senderIPv4;
    bit<48> targetMac;
    bit<32> targetIPv4;
}
```

Can define any new header

```
#define TYPE_IPV4 0x0800
#define TYPE_ARP 0x0806

state parse_ethernet {
    pkt.extract(hdr.ethernet);
    transition select(hdr.ethernet.ethType){
        TYPE_IPV4 : parse_ipv4;
        TYPE_ARP : parse_arp;
        default : accept;
    }
}

state parse_arp {
    pkt.extract(hdr.arp);
    transition accept;
}

state parse_ipv4 {
    pkt.extract(hdr.ipv4);
    transition accept;
}
```

Programmable parser

P4 example program

```
table arp_tbl {
    // This will be matched from incoming packet
    key = {
        hdr.arp.targetIPv4 : exact;
    }

    // Any one of the actions can be called
    // Based on the rule configured by controller
    actions = {
        arp_act;
        no_option;
    }
}
```

Programmable match-action table

```
action arp_act(bit<48> ifaceMac){
    // Sending packet back to incoming port
    ig_tm_md.ucast_egress_port = ig_intr_md.ingress_port;

    // Changing Ethernet headers
    hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;
    hdr.ethernet.srcAddr = ifaceMac;

    // Changing ARP headers
    hdr.arp.op_code = 2; // Arp req = 1, ARP reply = 2
    hdr.arp.targetMac = hdr.arp.senderMac;
    hdr.arp.senderMac = ifaceMac;

    // Swaping sender and target IPv4
    ig_md.arpTargetIPv4_temp = hdr.arp.targetIPv4;
    hdr.arp.targetIPv4 = hdr.arp.senderIPv4;
    hdr.arp.senderIPv4 = ig_md.arpTargetIPv4_temp;
}
```

Programmable action

```
apply {
    if (hdr.ethernet.isValid() && hdr.arp.isValid()){
        arp_tbl.apply();
    }
}
```

Programmable apply section (the main application logic)

Current research directions in programmable data plane

Programmable Dataplane

Hardware development

- FPGA
- ASIC
- Software Switch (BMv2)

Compiler design

- P4 language
- Domino language (SIGCOMM '16)

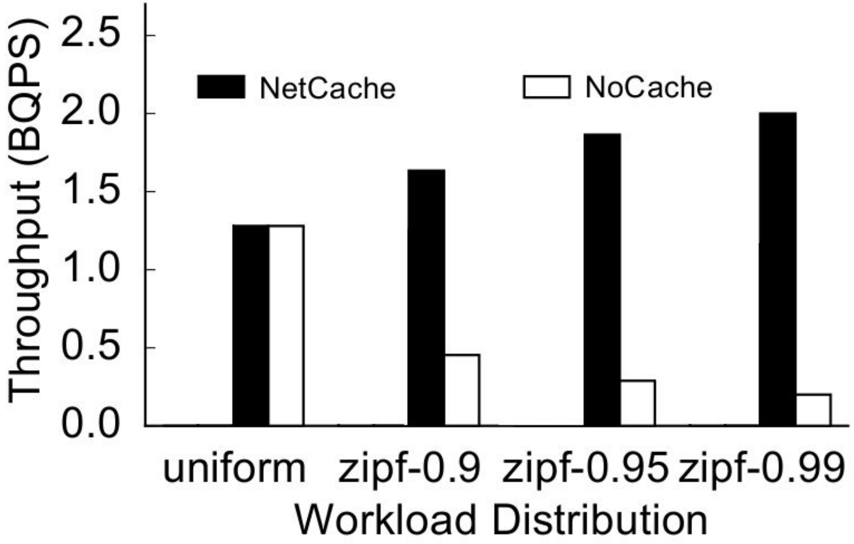
Application design & offloading

- In-network computation (NetCache, NetChain)
- Fault tolerant routing (Blink)
- Telemetry (Marple)
- Consensus (NetPaxos)
- Load-balancing

Programmable dataplane use case: In-network computation

Offload computation on programmable hardware

- **CPU load reduction**
  - Checksum calculation offloading
  - TCP connection setup and teardown offloading
- **In network cache**
  - Increases throughput, reduces latency
  - Handles skewed workload



Programmable dataplane use case: Network failure handling

Route selection and intelligent routing decision at data plane.

- **Failure recovery** at data plane
  - Traces failure from TCP retransmission
  - Faster than control plane driven recovery
- **Content based routing**
  - Treat packets based on custom headers
  - **E.g.** NetChain treats read & write request differently

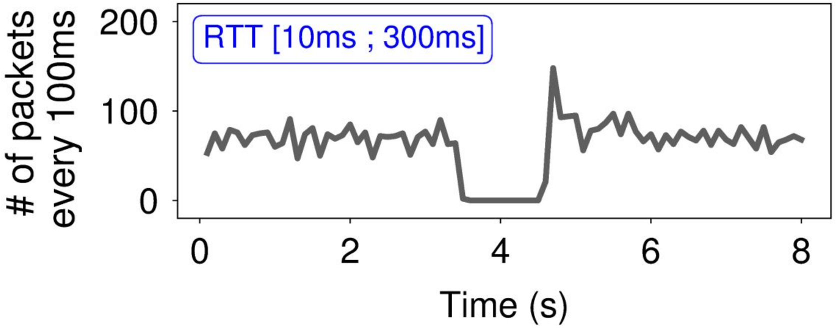


Figure: Blink recovers connectivity within 1.1 second of failure, completely at data plane

Programmable dataplane use case: Network telemetry

Network telemetry: Active monitoring of health and statistics of network

- **Software vs fixed function hardware**
  - Software - Expressive but inefficient
  - Hardware - Efficient but less expressive
- **Programmable data plane based** telemetry
  - Expressive as well as efficient

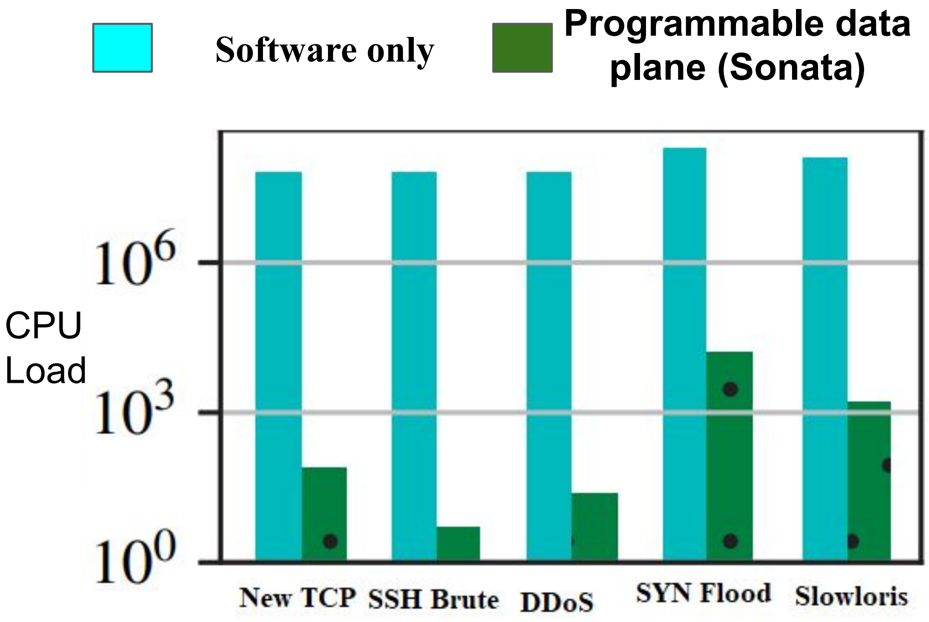


Figure: CPU workload: Software vs programmable hardware

Programmable data plane limitations

- **Costly** compared to fixed function switches
  - 32X100G NETBERG AURORA 710 (BAREFOOT TOFINO) - 7500 USD
  - Similar non-programmable switch - Approximately 1000 USD

- **Limited resources** at data plane
  - Limited CPU resource, limited on-chip memory.
  - Limited programmability & strict packet processing pipeline e.g. loop execution not supported

- Slow control plane operations e.g. populating match-action tables, loading program etc.

What should be offloaded?

- Require **high throughput and/or low latency** . E.g. In network KV store.
- Application **should be fairly simple**.
  - Should NOT store too much state. E.g. KV-store should cache hot items only.
  - Should NOT do complex calculations (e.g. division etc).
  - Should NOT use complex programming logic (e.g. loop etc.) or complex data structures.
- Should support modularisation and partial offloading.
  - e.g. In TCP protocol stack only connection setup and teardown
- Should NOT communicate too much with control plane.
- Should NOT require global network view.
- Fault tolerant or low-fault rate.

1. HotOS '19: Proceedings of the Workshop on Hot Topics in Operating Systems, May 2019, Pages 209–215
https://doi.org/10.1145/3317550.3321439