Introduction to P4 and In-Network Computing

1. An introduction to P4

1.1. What is PISA?

PISA (Protocol Independent Switch Architecture) is a single pipeline forwarding architecture. A typical PISA switch is composed of a programmable parser before each pipeline, a deparser after each pipeline, ingress pipeline, traffic manager (not programmable), and egress pipeline as shown in Fig.1. The programmable parser declares how the headers should be recognized and their order in the packet. The ingress and egress pipelines define the match-action tables and the exact processing algorithm using ALUs (Arithmetic and Logical Units), and they are the actual packet processing units. Match-action tables match the header based on a set of rules that is controlled by control plane and performs the corresponding action on the packet. Actions use primitives to customize the packet header or metadata. The programmable deparser indicates the output packet format on the wire.

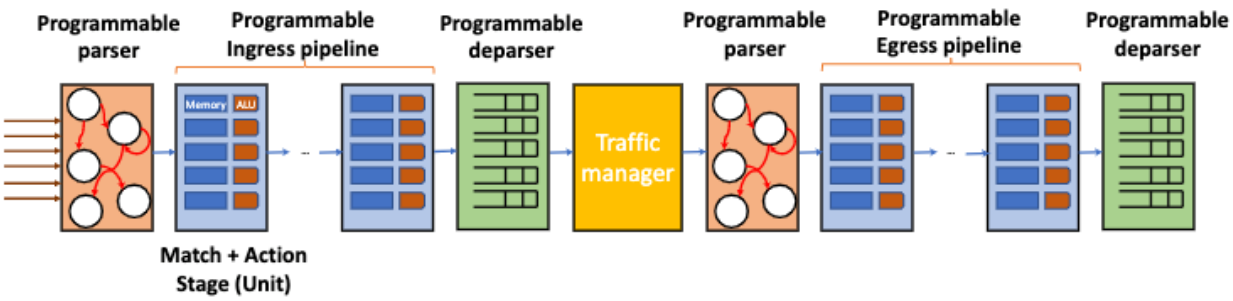

Fig.1: Abstract machine model of programmable switch architecture

1.2. V1model

The V1Model 4 [1] is the architecture commonly used with the behavioural model, called BMv2. It defines the programmable structure of the pipeline.

The BMv2 framework offers resources to implement a particular target (like a Simple Switch target, also referred to as simple_switch). The Simple Switch target is not the only target you can support. BMv2 can be used to support other targets, like P4Pi.

[1] https://github.com/p4lang/behavioral-model

1.3. What is P4?

P4 [2] is a high-level domain specific language for customizing the network protocol, which can express how packets are processed by the data plane of a programmable forwarding element. Example such elements are a hardware or software switch, network interface card, router, data processing unit (DPU), or a network appliance. There are two versions of P4: $P4_{14}$ and $P4_{16}$. $P4_{14}$ is deprecated, and currently $P4_{16}$ is commonly used and will be used during the course.

The core abstractions provided by the P4 language are:

- **Header types** describe the format (the set of fields and their sizes) of each header within a packet.
- **Parsers** describe the permitted sequences of headers within received packets, how to identify those header sequences, and the headers and fields to extract from packets.

- **Tables** associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multi-variable decisions.
- **Actions** are code fragments that describe how packet header fields and metadata are manipulated. Actions can include data, which is supplied by the control-plane at runtime.
- **Match-action units** perform the following sequence of operations:
    - Construct lookup keys from packet fields or computed metadata,
    - Perform table lookup using the constructed key, choosing an action (including the associated data) to execute, and
    - Finally, execute the selected action.
- **Control flow** expresses an imperative program that describes packet-processing on a target, including the data-dependent sequence of match-action unit invocations. Deparsing (packet reassembly) can also be performed using a control flow.
- **Extern objects** are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behaviour is hard-wired (e.g., checksum units) and hence not programmable using P4.
- **User-defined metadata:** user-defined data structures associated with each packet.
- **Intrinsic metadata:** metadata provided by the architecture associated with each packet — e.g., the input port where a packet has been received.

[2] Bosshart, Pat, et al. "P4: Programming protocol-independent packet processors." ACM SIGCOMM Computer Communication Review 44.3 (2014): 87-95.

1.4. P4 syntax and semantics

P4 core library

The P4 language specification defines a core library that includes several common programming constructs. All P4 programs must include the core library and also v1model in P4P:

```
# include <core.p4>

# include <v1model.p4>
```

An example program skeleton is shown in the next page.

```
#include <core.p4>
#include <v1model.p4>
/* HEADERS */
struct metadata { ... }
struct headers {
  ethernet_t   ethernet;
  ipv4_t       ipv4;
}
/* PARSER */
parser MyParser(packet_in packet,
        out headers hdr,
        inout metadata meta,
        inout standard_metadata_t smeta) {
  ...
}
/* CHECKSUM VERIFICATION */
control MyVerifyChecksum(in headers hdr,
                         inout metadata meta) {
   ...
}
/* INGRESS PROCESSING */
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta)
{
  ...
}
```

P4$_{16}$ types

Basic Types

- `bit<n>`: Unsigned integer (bitstring) of size n
- `bit` is the same as `bit<1>`
- `int<n>`: Signed integer of size n (>=2)
- `varbit<n>`: Variable-length bitstring

Header Types: Ordered collection of members

- Can contain `bit<n>`, `int<n>`, and `varbit<n>`
- Byte-aligned
- Can be valid or invalid
- Provides several operations to test and set validity bit: `isValid()`, `setValid()`, and `setInvalid()`
- `Typedef`: Alternative name for a type

Other types:

- **Struct**: Unordered collection of members (with no alignment restrictions)
- **Header Stack:** array of headers
- **Header Union:** one of several headers

Example usage:

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
header ethernet_t {
  macAddr_t dstAddr;
  macAddr_t srcAddr;
  bit<16>   etherType;
}
header ipv4_t {
  bit<4>    version;
  bit<4>    ihl;
  bit<8>    diffserv;
  bit<16>   totalLen;
  bit<16>   identification;
  bit<3>    flags;
  bit<13>   fragOffset;
  bit<8>    ttl;
  bit<8>    protocol;
  bit<16>   hdrChecksum;
  ip4Addr_t srcAddr;
  ip4Addr_t dstAddr;
}
```

Declaring and Initializing Variables

In $P4_{16}$ you can instantiate variables of both base and derived types. Variables can be initialized including the composite types. Constant declarations make for safer code. Infinite width and explicit width constants.

P4$_{16}$ Parsers

Parsers are functions that map packets into headers and metadata, written in a state machine style. Every parser has three predefined states:

- start
- accept
- reject

Other states may be defined by the programmer. In each state, execute zero or more statements, and then transition to another state (loops are OK, but not too many).

Example:

```
/* User Program */
parser MyParser(packet_in packet,
        out headers hdr,
        inout metadata meta,
        inout standard_metadata_t std_meta) {
  state start {
     packet.extract(hdr.ethernet);
     transition accept;
  }
}
```

Select statement

$P4_{16}$ has a select statement that can be used to branch in a parser, similar to case statements in C or Java, but without “fall-through behavior”—i.e., break statements are not needed.

In parsers it is often necessary to branch based on some of the bits just parsed. For example, etherType (Ethernet header type field) determines the format of the rest of the packet. Match patterns can either be literals or simple computations such as masks.

Example:

```
state start {
  transition parse_ethernet;
}
state parse_ethernet {
  packet.extract(hdr.ethernet);
  transition select(hdr.ethernet.etherType) {
    0x800: parse_ipv4;
    default: accept;
  }
}
```

$P4_{16}$ Controls

A user can declare variables, create tables, instantiate externs, etc. in P4 control.
Functionality is specified by code in **apply** statement. A user can represent all kinds of processing that are expressible as a directed acyclic graph (DAG):

- Match-Action Pipelines

- Deparsers
- Additional forms of packet processing (updating checksums)

Interfaces with other blocks are governed by user- and architecture-specified types (typically headers and metadata).

Control in P4 is similar to C functions (for those familiar with the language), but without loops and pointers.

Example:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta)
{
  /* Declarations region */
  bit<48> tmp;
  apply {
    /* Control Flow */
    tmp = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;
    hdr.ethernet.srcAddr = tmp;
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

Reflector

A reflector is a use case where the desired behavior is to bounce the packet back out on the physical port that it came into the switch on. This requires swapping source and destination MAC addresses within the header, and updating the output port to be the source port:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta)
{
  action swap_mac(inout bit<48> src,
                  inout bit<48> dst) {
    bit<48> tmp = src;
    src = dst;
    dst = tmp;
  }
  apply {
    swap_mac(hdr.ethernet.srcAddr,
             hdr.ethernet.dstAddr);
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

Simple action

Actions are very similar to functions. They can be declared inside a control or globally. In actions:

- Parameters have type and direction
- Variables can be instantiated inside
- Many standard arithmetic and logical operations are supported
    - +, -, *
    - ~, &, |, ^, >>, <<
    - ==, !=, >, >=, <, <=

    - No division/modulo
- Some non-standard operations are supported:
    - Bit-slicing: [m:l]

      Bit Concatenation: ++

Example:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta)
{
  action swap_mac(inout bit<48> src,
                  inout bit<48> dst) {
    bit<48> tmp = src;
    src = dst;
    dst = tmp;
  }
  apply {
    swap_mac(hdr.ethernet.srcAddr,
             hdr.ethernet.dstAddr);
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

Tables

Tables are the fundamental unit of a Match-Action pipeline. A table:

- Specifies what data to match on and match kind
- Specifies a list of *possible* actions
- Optionally specifies a number of table **properties**
    - Size

- Default action
- Static entries
- etc.

Each table contains one or more entries (rules). An entry contains:

- A specific key to match on
- A single action that is executed when a packet matches the entry
- Action data (possibly empty)

Match Kinds

The type match_kind is special in P4. It defines how a key is matched to a table entry.

The standard library (core.p4) defines three standard match kinds

- Exact match – The key and the entry are identical
- Ternary match – The key and the entry are the same, but a bit can take three values: 0, 1, don’t care (*).
- LPM match – Longest prefix match (see the introduction to networks presentation).

The architecture that we use (v1model.p4) defines two additional match kinds:

- Range – The key falls within a range defined by the entry
- Selector – Allows to select actions defined outside the tables (externs). You will not be using this type.

Other architectures may define (and provide implementation for) additional match kinds.

The following shows how match kinds are defined within architectures (not as part of user programs):

```
/* core.p4 */
match_kind {
    exact,
    ternary,
    lpm
}
/* v1model.p4 */
match_kind {
    range,
    selector
}
/* Some other architecture */
match_kind {
    regexp,
    fuzzy
}
```

Applying Tables in Controls

The following shows how a table can be applied within a control block:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t
standard_metadata) {
  table ipv4_lpm {
    ...
  }
  apply {
     ...
     ipv4_lpm.apply();
     ...
  }
}
```

Table Initialization

It is possible to initialize the values of a table using hard coded values, and shown in the following example:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t
standard_metadata) {
  table ipv4_lpm {
    ...
  }
```

```
    apply {
        ...
    }
    const entries = {
     /*{ dstAddr, srcAddr, vlan_tag[0].isValid(),
    vlan_tag[1].isValid() } : action([action_data])*/
            { 48w000000000000, _,  _, _ } :
    malformed_ethernet(ETHERNET_ZERO_DA);
            { _, 48w000000000000, _, _ } :
    malformed_ethernet(ETHERNET_ZERO_SA);
            { _, 48w010000000000 &&& 48w010000000000, _,
    _ } : malformed_ethernet(ETHERNET_MCAST_SA);
            { 48wFFFFFFFFFFFF, _, 0, _ } :
    broadcast_untagged();
            { 48wFFFFFFFFFFFF, _, 1, 0 } :
    broadcast_single_tagged();
            { 48wFFFFFFFFFFFF, _, 1, 1 } :
    broadcast_double_tagged();
            { 48w010000000000 &&& 48w010000000000, _, 0,
    _ } : multicast_untagged();
            { _, _, 0, _ } : unicast_untagged();
            { _, _, 1, 0 } : unicast_single_tagged();
    }
}
```

P4$_{16}$ Deparsing

A Deparser assembles the headers back into a well-formed packet. It is expressed as a control function:

- No need for another construct!
- packet_out extern is defined in core.p4: emit(hdr): serializes header if it is valid

For example:

```
/* From core.p4 */
extern packet_out {
  void emit<T>(in T hdr);
}
/* User Program */
control DeparserImpl(packet_out packet,
                     in headers hdr) {
  apply {
    ...
    packet.emit(hdr.ethernet);
    ...
  }
}
```

1.5. Stateless and Stateful Objects

Stateless Objects have no memory, and are reinitialized with every packet. This includes variables (metadata), packet headers, packet_in, packet_out.

Stateful Objects have memory, and keep their state between packets. This includes tables and externs (external functions). Examples of stateful externs supported by the v1 architecture are Counters, Meters, Registers, and Selectors.

2. P4Pi: P4 on Raspberry Pi

P4Pi is a low cost, open source hardware platform for teaching and research. P4Pi enables designing and deploying P4-based network devices using the Raspberry Pi board.

[3] https://github.com/p4lang/p4pi

Working with a P4 program in P4Pi:

Bmv2 is the default switch target in P4Pi, and the v1model architecture is included with the p4c compiler. To run a P4 program in P4Pi, you need to follow the following steps:

1. Compile a P4 program

   ```
   $ p4c --target bmv2 --arch v1model --std p4-16 test.p4
   ```

   The compiled P4 program `test.p4` will generate a new file called `test.json`

2. Run the compiled program

   ```
   $ sudo simple_switch –i 0@eth0 test.json
   ```

Indicate the network interface to send traffic in the switch. In the example, the network interface `eth0` is considered as port 0 in the switch. `test.json` is the compiled P4 program.

It is also possible to use CLI to configure the match-action table when running your P4 program in P4Pi:

```
$ simple_switch_CLI
```

Then you should be able to see P4 runtime (control plane):

```
$ RuntimeCmd:
```

Useful commands in P4 runtime:

- Show all tables

```
$ RuntimeCmd: show_tables
```

- Show the information of the specific table

```
$ RuntimeCmd: table_info [table_name]
```

- Show configured table entries in the specific table

```
$ RuntimeCmd: table_dump [table_name]
```

- Configure new table entries

```
$ RuntimeCmd: table_add [table name] [action name]
[table entry] => [action] [priority]
```

3. An Introduction to In-Network Computing

In-network computing is a emerging research area, focusing on computing within the network. More specifically, it refers to the execution of programs which are typically running on end-hosts within network devices. The network devices already exist within the networked-system and are already used to forward traffic. Therefore, in-network computing means there is no need to add new devices to the network, as the system already uses switches and network interface cards (NICs).

In the past, network devices were fixed-functional and supported only the functionality defined by their manufacturer. As network devices evolved, more and more programmability was introduced into the data plane, with commercial devices making their debut around 2015. In contrast to the past, today's programmable network devices allow users to implement their own functionality.

In the beginning, P4 was used mainly to define new protocols and networking related functionality. However, researchers have quickly started to build upon the language and platforms to port more complex functionality to the network.

To date, in-network computing was implemented on three classes of devices: FPGAs, SmartNICs and switch-ASIC.

3.1. Advantages of in-network computing

The main promise of in-network computing is performance in terms of throughput and latency. Today, many network devices support sub-microsecond latency, with low variance in non-oversubscribed scenarios. And as in-network computing refers to processing within the

network, it means that transactions are terminated within their path rather than reach an end-host, saving the latency introduced by the end-host and any network devices along the way from the in-network computing node to the end-host. Reduced latency is important in various scenarios. For example, high frequency trading firms try their best to build low latency trading systems which can lead to higher profits. Time sensitive applications, such as the control of robotic arms in on a manufacturing floor, also require low latency.

The second performance advantage, throughput, is a property of packet processing rate. Switch ASICs process nowadays up to ten billion packets per second, and therefore potentially support billions of operations per second per offloaded application. This class of switches is designed as pipelines, continuously moving data without stalls. In most cases, even if one operation (packet) is stalled (queued), e.g., while competing on shared resources (congestion), other packets continue to be served. Applications implemented using in-network computing have demonstrated x10,000 performance improvement compared with their host-based counterparts.

An unexpected benefit of in-network computing is power efficiency. While the power-per-watt benefit of accelerators is a known secret, network switches are notoriously regarded as power hungry. Furthermore, they are not power proportional, drawing significant power even when idle. However, if you consider operations-per-Watt, network switches are a lot more attractive, supporting millions of operations per Watt, meaning for some applications have x1,000 higher efficiency than software-based solutions. To illustrate, a million key-value store queries will “cost’’ less than one Watt on a switch. Since network switches are part of a user’s network, most of the power consumption is already paid

by packet forwarding, and the overhead of in-network computing is small, in the order of several percent of the overall switch power consumption.

Research has demonstrated that in-network computing can provide x10000 throughput improvement, x100 latency reduction and x1000 power saving per operation. In a word, the overhead of in-network computing is minimal as no extra space, cost or idle power are required. Furthermore, in-network computing reduces the load on the network by terminating user-generated data before it gets to the host, saving CPU cycles and freeing up resources.

[4] Tokusashi, Yuta, et al. "The case for in-network computing on demand." EuroSys 2019.

3.2. In-network computing applications

In-network computing has been applied to several classes of applications.

- Network functions, e.g., load balancer, NAT, or DNS server.
- Caching, using the network device to quickly reply with cached values.
- Data reduction and data aggregation, using the network device to aggregate or batch data from a number of sources, and to reduce the amount of data sent from the network device onward.
- Coordination, e.g., consensus algorithms.
- Cross-disciplinary use-cases, such as accelerating stream processing or storage systems.

3.3. Challenges

In-network computing is promising, but there are still a lot of challenges ahead. Two important questions are the benefits of in-network computing when traffic is encrypted, and the security risks presented by in-network computing. In addition, the architecture of network devices does not easily lend itself to machine learning applications. While systems running machine learning can benefit from acceleration within the network, running the training within the network has proven to be difficult so far.

In-network computing also faces several large technical challenges. The biggest challenge is probably being able to abstract the network-hardware from programmers. While P4 is a declarative language, it still operates at the packet-level. Ideally, programmers will be able to code using higher level abstractions. The language also lacks support for stateful operations, with current solutions being target-specific. Furthermore, to achieve high performance, programmers must be aware of the hardware target and leverage its capabilities in their code.

Porting between different network-hardware targets is not an easy task, and often requires a significant number of changes to the code. Porting the same code between heterogeneous targets, such as CPU, GPU, switch ASIC will be one step further.

Debugging tools will play a crucial role in any future success of in-network computing. While there are several formal verification tools, building debuggers that fit network-device architectures and pipelines moving data is hard.

As in-network computing evolves, more challenges arise. For example, is it possible to run multiple applications over the same network device? How do you isolate resources? And what is the difference between virtualisation on a CPU and on a network device?

4. In-Network Computing State of the Art

4.1. In-network Telemetry

Network telemetry has emerged as a mainstream term to refer to the network data collection and consumption techniques. Telemetry is an automated process for remotely collecting and processing network information. Network telemetry has been widely considered an ideal means to gain sufficient network visibility with better scalability, accuracy, coverage, and performance than traditional network measurement technologies. In-band network telemetry is an emerging representative of network telemetry, which has received extensive attention in both academia and industry in recent years. Different from traditional network measurement and software-defined measurement, in-band network telemetry combines packet forwarding with network measurement. In-band network telemetry collects the network status by inserting metadata into packets as the packet traverses through multiple switching nodes.

[5] Tan, Lizhuang, et al. "In-band network telemetry: A survey." Computer Networks 186 (2021): 107763.

4.2. In-Network Caching

Caching is often applied in the computing world to reply faster to popular queries. Popular items receive far more queries than others, and the set of “hot items” changes rapidly due to popular posts, limited-time offers, and trending events. This skew can lead to severe load imbalance, which results in significant performance degradations.

In-Network Caching leverages the power and flexibility of programmable switches to cache data in the network. Caching hot items in the network is a natural solution to provide performance and strong consistency guarantees for in-memory key-value stores under highly skewed and dynamic real-world workloads, where caching is an effective technique for alleviating load imbalance

[6] Liu, Ming, et al. "Incbricks: Toward in-network computation with an in-network cache." ASPLOS 2017.
[7] Jin, Xin, et al. "Netcache: Balancing key-value stores with fast in-network caching." SOSP 2017

4.3. In-network DDoS Detection

A distributed denial-of-service (DDoS) attack is a malicious attempt to disrupt the normal traffic of a targeted server, service or network by overwhelming the target or its surrounding infrastructure with a flood of Internet traffic. The most obvious symptom of a DDoS attack is a site or a service suddenly becoming slow or unavailable.

The key concern and difficulty in mitigating a DDoS attacks is differentiating between attack traffic and normal traffic. In the modern Internet, DDoS traffic comes in many forms. The traffic can vary in

design from un-spoofed single source attacks to complex and adaptive multi-vector attacks.

In-network DDoS defence maps the defence primitives to run on programmable network devices and when necessary, on server software for effective defence.

[8] https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/
[9] Zhang, Menghao, et al. "Poseidon: Mitigating volumetric ddos attacks with programmable switches." NDSS 2020
[10] Liu, Zaoxing, et al. "Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches." USENIX Security 2021

4.4. In-network Computing for Distributed Systems

A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires coordinating processes to reach consensus, or agree on some data value that is needed during computation. In-band consensus focuses on how to take advantage of the programmable forwarding plane to accelerate consensus protocols.

[11] Dang, Huynh Tu, et al. "P4xos: Consensus as a network service." IEEE/ACM *Transactions on Networking 28.4 (2020): 1726-1738.*

4.5. In-network Aggregation

Cloud network deployments have found the pace of distributed machine-learning (ML) training hard to match, skewing the ratio of computation to communication towards the latter. Since parallelization techniques like mini-batch stochastic gradient descent (SGD) training alternate

computation with synchronous model updates among workers, network performance now has a substantial impact on training time.

In-network aggregation primitives can accelerate distributed ML workloads and can be implemented using programmable switch hardware. Aggregation reduces the amount of data transmitted during synchronization phases, which increases throughput, diminishes latency, and speeds up training time.

[12] Sapio, Amedeo, et al. "In-network computation is a dumb idea whose time has come." HotNets 2017.
[13] Lao, ChonLam, et al. "ATP: In-network Aggregation for Multi-tenant Learning." NSDI 2021.

4.6. In-network Machine Learning

In-network Machine Learning (ML) can be defined as the partial or full offloading of ML algorithms to run within network devices. Strictly speaking, the scope of in-network ML can be limited to the forward classification process of ML algorithms being offloaded to the data plane, while the training part remains on the host (including accelerators) or in the control plane. In-network ML algorithms follow an offline training, online (in-band) inference pattern. Feature extraction can be done either by parsing within the data plane or customized headers. A mapped ML model is typically implemented within the Match-Action pipeline, and the decision can be stored in a header or turned into an action within the network device.

[14] Changgang Zheng et al. "IIsy: Practical In-Network Classification." arXiv:2205.08243 (2022).
[15] Changgang Zheng et al. "Automating In-Network Machine Learning." arXiv:2205.08824 (2022).