Development and Validation of P4 tools

Project report submitted in partial fulfillment

of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

by

Shubham Pandey

(Roll Number: 18CS4134)

based on internship carried out

at Intel India Technologies Pvt. Ltd.

Submitted to

Asso. Prof. Sajal Mukhopadhyay

Department of Computer Science and Engineering

National Institute of Technology Durgapur

Department of Computer Science and Engineering
National Institute of Technology Durgapur

June 04, 2020

Certificate of Examination

Roll Number: *18CS4134*
Name: *Shubham Pandey*
Project Title: *Development and Validation of P4 tools*

We the below signed, after checking the project report mentioned above and the official record book (s) of the student, hereby state our approval of the project report submitted in partial fulfillment of the requirements of the degree of *Master of Technology* in *Computer Science and Engineering* at *National Institute of Technology Durgapur*. We are satisfied with the volume, quality, correctness, and originality of the work.

Sajal Mukhopadhyay	Tandra Pal
Principal Supervisor	Head of the Department

Dedication

Dedicated to my Family and every Mentor of my life.

Signature

Declaration of Originality

I, *Shubham Pandey*, Roll Number *18CS4134* hereby declare that this project report entitled *Development and Validation of P4 tools* presents my original work carried out at Intel India Technology Pvt. Ltd. as a postgraduate student of NIT Durgapur.

I declare that this written submission represents my work in my own words and where others ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

June 04, 2020
NIT Durgapur

Shubham Pandey

Acknowledgment

First and foremost, I would like to express my deep sense of profound gratitude to my college mentor **Dr. Sajal Mukhopadhyay**, Associate Professor, Dept. of CSE, NIT Durgapur for his constant support and allowing me to have my final year internship at Intel India Technology Pvt. Ltd.

I must also thank the HoD, **Dr.Tandra Pal**, faculty members, and staffs of the Department of Computer Science and Engineering, NIT Durgapur for their kind help and support.

I would like to acknowledge the support of my Manager **Balachandra Sambasivam** and all the team members who helped me grow intellectually and socially since last one year at Intel Bengalore.

I must acknowledge the extended support of my fellow M.Tech scholars and friends **Anup Srivasatava, Ramesh Kumar, Sanjeev Kumar, Kiran Purohit, Sweta Rawat, Prachi Mehare, Aniruddho** and **Arpan.**

June 04, 2020
NIT Durgapur

Shubham Pandey
Roll Number: 18CS4134

Abstract

NIC stands for a Network Interface Card that makes it possible to connect a computer device over a network. It may be a wired or wireless network but a computer can not connect to any other computer without NIC.This NIC has three functional planes data plane, control plane and management plane. Basically those planes are the set of algorithms. Every plane has its own job. Data plane handles the processing of the incoming packets and control plane handles decides how these packets should be handled. Network Operating System is the software that controls these planes. It is known to the network engineer that they can define the behavior of control plane but not the data plane. This means that we can not handle how to process a packet using traditional methods. So, there is a long need to have the ability to defined the behavior of data plane.

In this report we'll look at the SmartNIC that allows packet processing to take place at the NIC itself, not on the CPU itself. To acheive this, we will use the Programming Protocol Independent Packet Processing(P4) Language which will enable us to define the data plane behavior. An open source P4 compiler is used, and with P4 we are able to define our own programmable components which decide how to process an incoming packet.

This report also shows a simple switch programmed in P4 and its runtime files that uses some api to help control plane add / delete / update entries. Having the ability to control packet processing is a long pending item for network engineers and because P4 language and its runtime are built to be protocol-independent, hardware-independent it appears to have a bright future in networking.

This report also includes my work in P4 Compiler validation and shows the important components of the P4 Compiler test framework and the development of test cases.

Keywords*: *SmartNIC*; *P4 language*; *P4 Compiler*; *P4 Runtime*; *Validation*; *Command line Options.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Intel's Network Division Group develops Ethernet solutions for Data Centers. The team assigned to me works on the development of firmware and software stack for customized SmartNICs for Cloud Service Providers. NIC stands for a "Network Interface Card". A NIC is practically a PCIe card that plugs into a server or storage box to enable connectivity to an Ethernet network. Traditionally, the CPU does the NIC processing, but a **SmartNIC** goes beyond simple connectivity, and implements network traffic processing on the NIC itself not on the CPU. This likewise cuts down the load on the CPU. Sometimes the definition of a SmartNIC is focused entirely on the implementation, as proposed by few vendors. The SmartNIC is, to say the least, fully programmable and allows total flexibility in its data plane, courtesy P4 language for programming flexible data plane elements because it is incredibly flexible and allows data plane innovation.[1]

Programming Protocol-independent Packet Processors (P4) is a declarative language used to describe how packets are transmitted via a network forwarding device pipeline, such as a switch, NIC, router or network function appliance. It was created as a next step in Software-Defined Networking (SDN) evolution. The data plane of these systems will run from fixed-function ASICs (such as those used in L2 switches) to fully programmable, general-purpose CPUs used in router, web proxy and firewall implementations, also known as NPU's, depending on the functionality and complexity of the device. P4 designers state that it is a programming language for packet processors, irrespective of the packet processing unit- Hardwired ASICs or NPUs, as long as they can understand P4. It is a high level Domain-Specific Language (DSL), dedicated to network interface programming. It allows to specify the format of packets (protocol's headers) to be recognized by network devices and actions to be performed on incoming packets (forwarding, headers modification, adding protocol header, etc).Nevertheless, the P4 language is not consumed directly by the network device, but for a particular platform, it must be compiled into the source code (some target-specific binary).

One of the primary goals of P4 is to keep the language **hardware independent**[2]. P4 is meant to describe/specify the behavior of the data plane, but not how the data plane is actually implemented. The job of interpreting P4 behavioral code and generating a lower-level, device-specific implementation falls to the manufacturers of the target systems. Developers

write P4 source code and rules, compile the P4 to a target-specific representation that is used to program the data plane.

The other two primary goals of P4 are **protocol independence** and **reconfigurability**[2].Most network data planes do three simple operations: parsing packets, match / action operations, and reassembly of packets. P4 includes coding structures that make it easy to understand defining such operations. And since P4 is an actual language, users are able to specify the structures of the packet header that the parser must extract. This is a major improvement over protocols like OpenFlow which can only parse the headers for network protocols that are supported. OpenFlow officially supports the Ethernet, IP, or TCP headers. If an NVGRE packet needs to be parse, OpenFlow will need to add support for that protocol. In comparison, P4 allows the header stack of the flowing packets to be specified, and parsed. This ability is known as protocol independence. In addition , the ability to write, compile, and transfer new P4 source code and rules into the data plane of a framework already deployed in a production environment is known as reconfigurability. P4 allows not only to simply update the data in forwarding table, but also restructuring/redesigning the table itself.

Chapter 2

Background Study

Standard Telecommunications Architecture comprises three planes- starting from the bottom, the Data Plane, followed by the Control Plane, and lastly, at the top, the Management Plane. A Plane is defined as a group of algorithms. These algorithms are actually different in terms of the traffic they process, the kind of performance requirements they have, the methodologies that they are designed with and even the kind of hardware they are often run on are different and are independently from each other and they interact through well-defined interfaces. The software that controls the planes is often called a Network Operating System (NOS).[3] Since past decade, the Data Plane algorithms are not a part of the NOS, as they are encapsulated in the Application Specific Integrated Circuits (ASICs) to increase the speed of packet processing. The classical telecommunications architecture can be divided into data plane, control plane and management plane. Data plane is the layer, where data packets are being processed and forwarded, while the control plane decides how these packets should be handled.[4] The network designers have their own idea of the NOS; they can customize it in their own way- the Control Plane, and the Management Plane, but not the Data Plane. The packet processing in the hardware is restricted, and happens as per the data sheet of the chip. Hence, there was a need to make the data plane programmable, to empower the network designers to process the packets the way they want, which gave rise to P4. Thus, using the P4 language you can specify which packet's headers can be processed and which actions can be performed on packets. The data plane programming was the missing link in the software-based network systems, where control plane and management plane are programmable already.[4]

SmartNIC is a reprogrammable networking interface that runs at high performance in high-scale clouds data center servers to boost performance by offloading CPUs in servers by performing network datapath processing. [1] Performing network offloads on the NIC is an old concept with many traditional NICs supporting offload of functions like checksum and segmentation. However, with the recent tectonic shift in cloud data center networking driven by software-defined networking (SDN) a new class of offload NIC is needed - namely the SmartNIC. From technology as well as business perspective, there are strong points for adopting SmartNICs. The complexity of the server-based networking data plane has increased dramatically with the introduction of overlay tunneling protocols such as VXLAN,

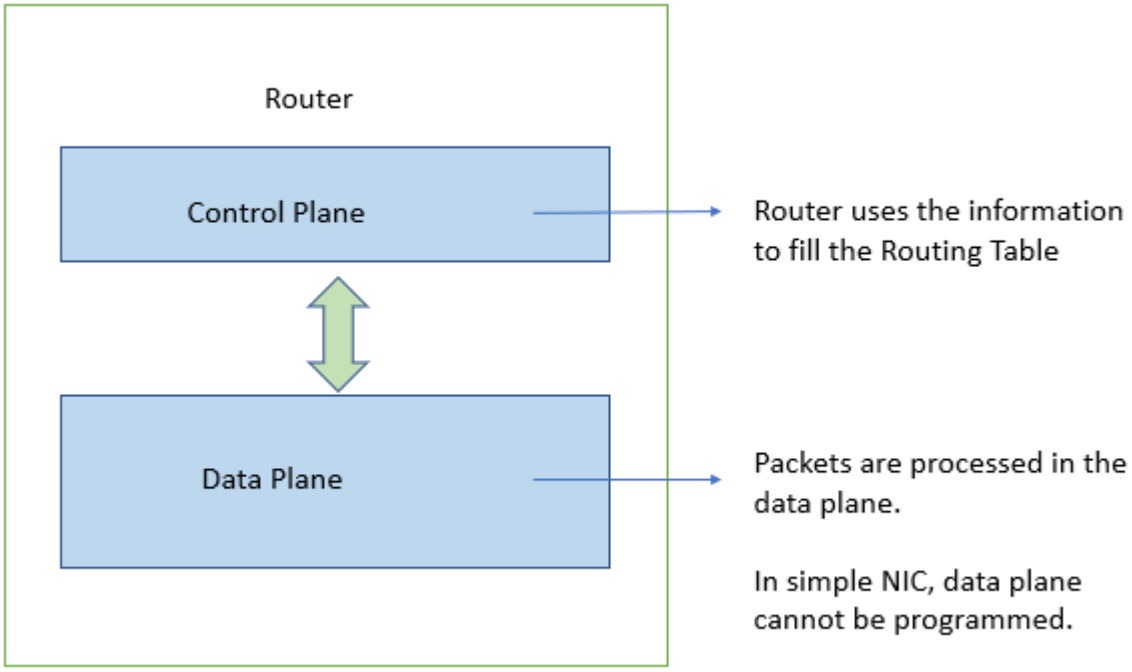

Figure 2.1: Networking Planes

and virtual switching with complex actions. Cloud Service Providers are rolling out new services and scaling their services. Hence the change in the networking logic is equivalent to the speed of software. SmartNICs are all about Cloud and Cloud cadence is very aggressive. Products have to be developed aggressively without delay. Hence to survive in a competitive market, products have to be delivered on a cadence i.e. continuously put next generation products on the roadmap.

One of the most important reasons why the world needs SmartNICs is that modern workloads and data center designs impose too much networking overhead on the CPU cores. With faster networking, the CPU utilizes too much of its valuable cores classifying, tracking, and steering network traffic.[5] These expensive CPU cores are designed for general purpose application processing, and the last thing needed is to consume all this processing power simply looking at and managing the movement of data. Renting out more cores for application processing is where the real value creation occurs. Hence there is an active need to lower Data Center networking 'tax' i.e. the number of cores running in the infrastructure that are not used by the application. SmartNIC reduces infrastructure overhead in large scale DCs by perform network offloads, hence business model is improved by renting out more cores.

Chapter 3

My Contribution

3.1 P4 for programming the data plane.

SmartNIC is Fully Programmable and allows for total flexibility in its Dataplane, courtesy P4 language for programming flexible data plane elements as it is incredibly flexible and allows innovation in the data plane.

i. **P4 Language**

P4 stands for Programming Protocol-Independent Packet Processors is a programming language for controlling packet forwarding planes in networking devices, such as routers and switches. P4 is distributed as open-source, permissively licensed code, and is maintained and developed by the P4 Language Consortium, a not-for-profit organization 6 years ago in 2013. P4 is designed to specify only the data plane functionality of the target. P4 programs also partially define the interface by which the control plane and the data-plane communicate, but P4 cannot be used to describe the control-plane functionality of the target.[6] In a traditional switch the manufacturer defines the data-plane functionality. The control-plane controls the data plane by managing entries in tables. A P4-programmable switch differs from a traditional switch. The data plane functionality is not fixed in advance but is defined by the a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program and has no built-in knowledge of existing network protocols. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

ii. **Benifits of P4**

a. **Flexibility**: P4 allows many of the packet processing policies to the hand of users that makes it flexible.

b. **Expressiveness**: P4 can help the user in representing the hardware independent packet processing algorithm in basic operations and table loopkups. Also the same program can be taken to different hardwares.

c. **Software engineering**: P4 programs provide important benefits such as type checking, information hiding, and software reuse. Previously all these things were exposed to the hardware vendors.

d. **Debugging**: Manufacturers can provide software models of an architecture to aid in the development and debugging of P4 programs.

iii. **Header Format**

The design begins with the specification of header format. Several domain specific languages are proposed for this. P4 borrows number of ideas from them. In general each header is specified as an ordered list of field names together with their widths. These header fields can be declared using structures similar to what is done in C proramming language. Following is an example to declare Ethernet and Ipv4 header.

```
header ethernet {
        fields{
                dst_addr : 48;
                    // mac address of 48 bits
                src_addr : 48;
                ethertype : 16;
        }
    }

header ipv4{
        fields{
                version : 4;
                ihl : 4;
                diffserv : 8;
                totalLen : 16;
                identification : 16;
                flags: 3;
                fragOffset : 13;
                ttl : 8;
                protocol : 8;
                hdrChecksum : 16 ;
                srcAddr : 48;
                dstAddr : 48;
        }
    }
```

iv. **Packet Parser**

P4 assumes the underlying switch can implement a state machine that traverses packet headers from start to finish, extracting field values as it goes. State machine where state can be user defined. The extracted field values are sent to the match+action tables for processing. P4 describes this state machine directly as the set of transitions from one header to the next.[6] Each transition may be triggered by values in the current header. Parsing starts in the start state and proceeds until an explicit stop state is reached or an unhandled case is encountered (which may be marked as an error). Upon reaching a state for a new header, the state machine extracts the header using its specification and proceeds to identify its next transition. The extracted headers are forwarded to match+action processing in the back-half of the switch pipeline. Parser takes the data packet as input and depending upon the how parser is defined the following headers are extracted from the packet. It is state machine with many states. There are 3 fixed states of the parser Start, Accept and Reject.

```
parser start{    // beginning state
        ethernet;
    }

Parser ethernet {
        Switch(ethertype) {
                Case 0x8100 : ipv4;
                default : accept;
        }
    }

Parser ipv4{
        Switch(version) {
                Case 2 : accept;
                Default : reject;
        }
    }
```

v. **Table Specification**

The programmer describes how the defined header fields are to be matched in the match+action stages and what actions should be performed when a match occurs. The table is defined using the keyword table followed by the table name. The reads attribute declares which fields to match. The match can be possible for more than one

field. The actions attribute lists the possible actions which may be applied to a packet by the table. An entry is a statement that takes input and acts on it if matched. The number of inputs to a matching is equal to the no of fields specified in the reads. Action can be both used defined or any predefined action. The max size attribute specifies how many entries the table should support. Following is an example of a table

```
table ipv4_table{
        reads{
                ethernet.dst_Addr;
                ipv4.checksum;
        }
    max_size :  2000;
    Actions{
                X1, Y1 : some_action();
                X2, Y2 : some_action();
        }
    }
```

vi. **Action Specification**

Actions are the methods that user can specify to make the compiler do what action he wants when it matches the value in the match action unit. There are some primitive actions defined in the P4 language but it also allows user to define his own user defined action. User can use these primitive actions or new actions can be added as per the user need. Keyword action is used to define any new user defined action. Following is an example of an action.

```
    // primitive actions
set_field (ethernet.ethtype, 2);
increment();

    //user defined actions that uses primitive actions
action user_defined_action(){
    set_field (ethernet.ethtype, 2);
    increment();
    }
```

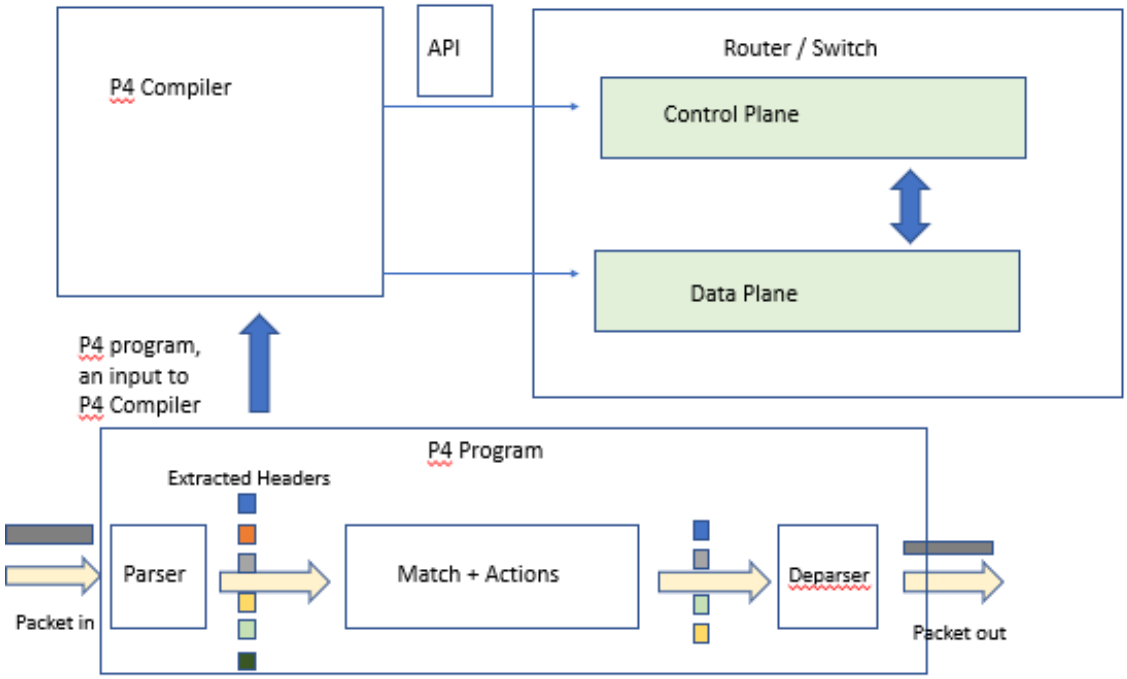

Figure 3.1: Compiler Data Flow

3.2 A very simple Switch.

A switch is a device that connects other devices into one computer network. Multiple data cables is plugged into a switch to allow communication between different networked devices. Switches control the data flow through a network by only transmitting a received network packet to the one or more computers the packet is intended for. Each networked device that is connected to a switch can be identified by its network address, allowing the switch to direct traffic flow to optimize network security and performance. As an example to illustrate the architecture features, consider implementing very simple switching in P4.

3.2.1 Architecture Description

The "Very Simple Switch" (VSS) is the name given to the architecture. VSS accepts packets from one of eight input Ethernet ports, via a recirculation channel, or from a port directly connected to the CPU. VSS has a single parser that feeds into a single match-action pipeline feeding into a single deparser. After exiting the deparser, packets are emitted through one of 8 output Ethernet ports or one of 3 "special" ports[7]:

• Packets sent to the "CPU port" are sent to the control plane

• Packets sent to the "Drop port" are discarded

• Packets sent to the "Recirculate port" are re-injected in the switch through a special input port.

The white blocks in the figure 3.2 are programmable, and the user must provide a corresponding P4 program to specify the behavior of each such block. The red arrows

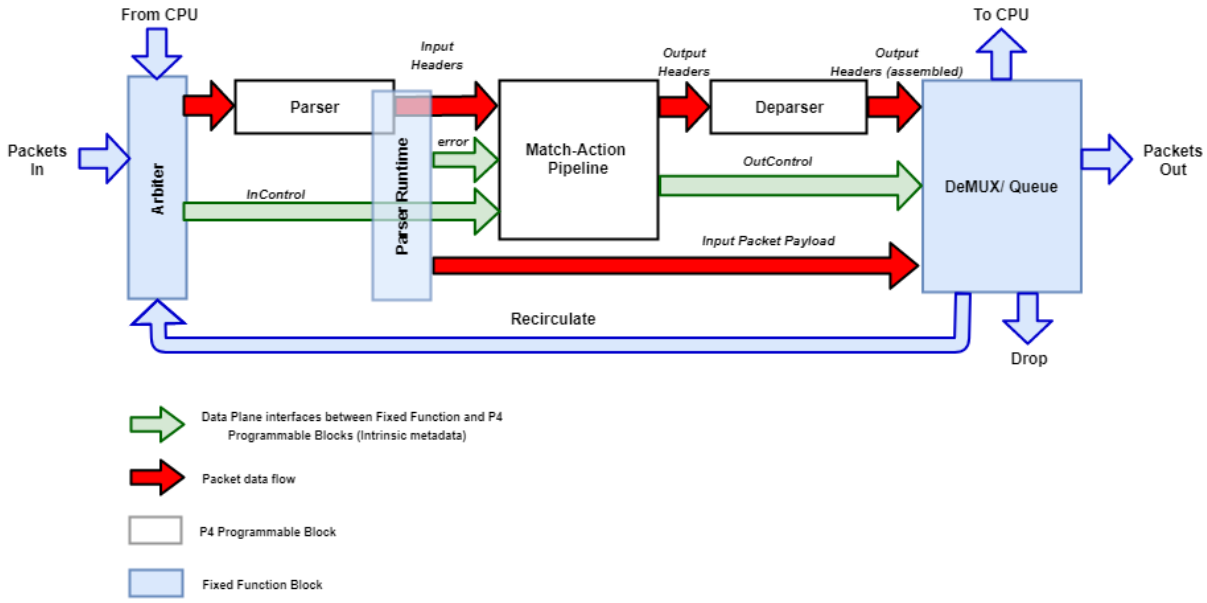

Figure 3.2: A Very Simple Switch Architecture.

indicate the flow of user-defined data. The blue blocks are fixed-function components. The green arrows are data plane interfaces used to convey information between the fixed-function blocks and the programmable blocks—exposed in the P4 program as intrinsic metadata.[7] VSS also has a number of fixed-function blocks whose behaviours are as follows:

a. **Arbiter Block**

   It receives packets from one of the Ethernet physical input ports, from the control plane, or from the recirculation port for data. The block compute and verifies the Ethernet trailer checksum for packets obtained from Ethernet ports. If the checksum does not match, it discards the packet. If the checksum matches, then it will be removed from the packet payload. The arbiter block sets the ”inCtrl.inputPort” value after receiving a packet which is an input to the parser with the name of the input port where the packet was originating[7].

b. **Parser runtime block**

   The Parser block runtime parser works with the parser in concert. It provides the match-action pipeline with an error code based on the parser behavior, and provides demux block with information about the packet payload (e.g., the size of the remaining payload data)[7]. If the parser completes the processing of a packet, the match-action pipeline is invoked with the associated metadata as inputs (packet headers and user-defined metadata).

c. **DeMux block**

   DeMUX block ’s core functionality is to receive the outgoing packet headers from the

deparser to assemble them into a new packet, and to send the result to the appropriate output port. The output port is specified by the outCtrl.outputPort value, which is set by the pipeline for match-action[7]. Sending the packet to the drop port will cause the packet to disappear. Sending the packet to an Ethernet output port numbered between 0 and 7 causes it to be emitted on the physical interface. Sending a packet to the output CPU port causes transfer of the packet to the control plane. Sending the packet to the recirculation port on the output allows it to appear at the input recirculation port.

3.2.2 Architecture in P4 Language

P4 program provides a declaration of VSS, as it would be provided by the VSS manufacturer. The declaration contains several type declarations, constants, and finally declarations for the three programmable blocks. The programmable blocks are described by their types; the switch programmer has to provide implementation of those blocks.
The file core.p4 defines some standard data-types and error codes. So we'll include the file int the header. "error" is a built-in P4 type for holding error codes. Also we will declare the output ports which have their values fixed so they are constants. The declaration extern Checksum16 defines an external object whose services can be invoked in order to calculate checksums.

```
parser Parser<H>(packet_in b,
                 out H parsedHeaders);
```

The parser reads its data from a packet in, a predefined external P4 object which is declared in the core.p4 library. The parser writes its output (the keyword out) into argument parsedHeaders.

```
control Pipe<H>(inout H headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl);
```

The pipeline receives three inputs: headers, a parser error parseError, and metadata for the inCtrl function. The pipeline writes its outputs into outCtrl, and it will change the headers in order for the deparser to consume.
The next declaration is for Deparser which takes parser output headers from the parser and assembles them into an output packet. Implementation of the deparser is given next.

```
control Deparser<H>(inout H outputHeaders,
                    packet_out b);
```

The top-level package is called VSS; for programming a VSS, a package of this type must be instantiated by the user (shown in the next section)

```
package VSS<H>(Parser<H> p,
               Pipe<H> map,
               Deparser<H> d);
```

3.2.3 Implementation in P4

The VSS declaration package has three complex parameters, respectively of the types Parser, Pipe, and Deparser; which are exactly the statements we have just mentioned. For these parameters to program the target, one has to supply values. Following is the implementation of the Very Simple Switch Architecture.

Parser is a state machines that is used to extract headers fro the incoming packet. Depending on the header fields different parser state are reached or otherwise packet is discarded with an error message. We have defined a two state machine that starts with extracting header ethernet, depending on the ethType next state ipv4 is achieved.

```
parser Parser(packet_in b, out Parsed_packet p) {
    Checksum() ck;

    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
         //if value matches go to next state
            0x0800: parse_ipv4;
        }
    }

    state parse_ipv4 {
        b.extract(p.ip);
        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
        // Verify that packet checksum is zero
        verify(ck.get() == 16w0, error.IPv4ChecksumError);
        transition accept;
    }
}
```

Other user programmable block is the match action pipeline which takes parsed header from the parser and depending upon the match value it performs the specified action. Tables contain actions, in our example we have used two actions one that set the output destination

port and other that sets the next hop destination address. Table chooses one action from the two depending upon the destination address of current ip header.

```
control Pipe(inout Parsed_packet headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl) {
    bit<32> nextHop;
    action Drop_action() {
        outCtrl.outputPort = DROP_PORT;
    }
    action Set_nhop(IPv4Address ipv4_dest, PortId port) {
        nextHop = ipv4_dest;
        headers.ip.ttl = headers.ip.ttl - 1;
        outCtrl.outputPort = port;
    }
   table ipv4_match {
       key = { headers.ip.dstAddr: lpm; }
       actions = {
            Drop_action;
            Set_nhop;}
       size = 1024;
       default_action = Drop_action;
     }
}
```

The last is the Deparser which takes parsed packet from the Match Action Pipeline and emits the either to the Control Plane or drops the packet. Also a VSS package is required to be instantiated to call the above three programmable units.

```
control Deparser(inout Parsed_packet p, packet_out b) {
    Checksum() ck;
    apply {
        b.emit(p.ethernet);
        b.emit(p.ip);     }
}

VSS(Parser(), Pipe(), Deparser()) main;
```

3.3 Command Line Options of P4 Compiler

The open source p4 compiler comes with many command line options but some of very useful commmand line options are below. In general, command-line compiler options may appear in a single invocation of a compiler in any order. The results of certain choices, however, depend on the order that appears in the command line and how it is paired with other similar options[8]. The compiler allows you to use multiple options even when they can clash. To define command-line options for the compiler, you can use the environment variable. The command line options take precedence over the options specified in the environment variable[8].

Command	**Use**
-h, –help	show this help message and exit
-b TARGET, –target TARGET	specify target device
-a ARCH, –arch ARCH	specify target architecture
-D PREPROCESSOR-DEFINES	define a macro to be used by the preprocessor
-E	Only run the preprocessor
-e	Skip the preprocessor
-I PATH	Add directory to include search path
-o PATH, –output PATH	Write output to the provided path
–p4runtime-files P4RUNTIMEFILES	Write the P4Runtime control plane API description (P4Info) to the specified files (comma-separated list); format is detected based on file suffix. Allowed suffixes are .txt, .json, .bin.
–p4runtime-format binary,json,text	Choose output format for the P4Runtime API description (default is binary). [Deprecated; use '–p4runtime-files' instead]
–std p4-14,p4-16	Treat subsequent input files as having type language.

Table 3.1: Commmand line options and their use.

Steps to compiler a P4 Program.
In the commands below, the -b option selects bmv2 (Behavioral Model Version 2) as the target, which is the software switch that we will use to run the P4 program. The general command format :

```
p4c –b <target name> –I <include_file> –o <output_dir name>

p4c –b bmv2 test.p4 –o test.bmv2
```

This compiler generates a directory test.bmv2 which contains a file test.json which the generated "executable" code which is run by the software switch. The P4 compiler also give us a command line options to get Runtime files for our P4 program. We will talk about P4 Runtime in the next section.

3.4 P4 Runtime

SDN is unfolding its boundaries in the age of softwarization and industries are focusing on taking networking control over the top of the pyramid – from sophisticated hardware to software application. P4 has already taken over the bottom of the pyramid (forwarding plane), by offering down to ASIC network programmability[9]. Control Plane is network device brain. It has the specific application which can learn and populate data plane specified tables. The control plane or software which is used to configure the packet's control movement was on the same device in traditional network devices.Because of these, for any changes, the network admin has to configure the switch by separately connecting to each device. To add more complexity, each chip vendor provides their own proprietary interfaces/APIs for controlling data plane. To make network administrators' life easy,

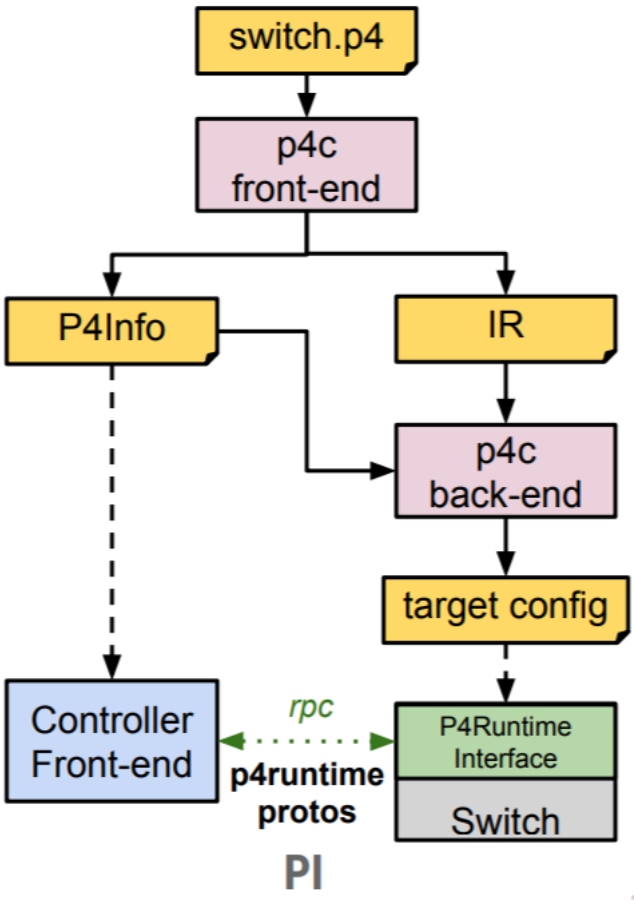

Figure 3.3: P4 Runtime Workflow

the aim was to separate networking devices' Control and Data Planes. But still, it was impossible to use single control plane APIs to control switch ASICs from different chip vendors because of different interface from different vendors. OpenFlow is the first standard open source communication protocol between the control plane and data plane by ONF (Open Networking Foundation) to promote adoption of SDN. OpenFlow is a solution to SDN networks which provides a standard interface between controller and switch in terms of flows. Flows contains match fields, priority and actions which are organized in tables. The provided APIs are generic, not target-dependent, thus making them suitable for any OpenFlow compatible control plane software. There are various limitations with OpenFlow. It was designed keeping in mind the fixed switches which makes it unscalable to new protocols in future. They are independent of target, but based on protocol[4].

With P4 programmable data plane, there were no standard for control plane interface. Few vendors have built their own proprietary tools to auto-generate APIs that can be populated from P4 software or JSON tables. But there was no standard on how those APIs should be specified. P4 Runtime architecture makes it independent of all protocols and subsequent forwarding switches. The same API can be used to control different switches supporting different protocols. P4 Runtime API have support for Managing behavior of data plane by adding, deleting, modifying, displaying entries in match-action tables. P4 compiler auto-generates the APIs needed for tables to be filled in. P4 language can be used to define the forwarding pipeline, and the forwarding logic can be managed and modified using P4 Runtime[4]. With P4, SDN controllers have the ability to redefine tables, entries, parser, match actions and the logic for packet processing. Thus providing complete control of the network.

3.4.1 Runtime file using P4 Compiler

P4 compiler is able to generate a runtime file that an be used to fill up tables by the control plane. The file format and the name of the file can be specified by the user using commmand line option. The default file format is JSON file format. For the above table ipv4-match we will generate the runtime files.

```
p4c –p4runtime–format json –p4runtime–files rt.json file.p4
```

This command takes two command line options one –p4runtime-format tells the compiler about the type of runtime file to be generated and second –p4runtime-files tell the name of the runtime file to be generated. Both these commands must be included to get the runtime file from P4 Compiler. Below is the insight of the runtime file generated for table ipv4-match in JSON format where all the information about the actions and tables is stored. All actions and tables have an id associated wih them that is refered whenever an action is called or a table is applied. Another field called name is associated with every action and table i.e. the name used in p4 program. Tables also have a match field and match type. Whenever this

match field matched for an entry a specific action is taken. Actions are refered inside the table using their id.

```
actions {
    preamble { id: 16805608
        name: "drop" }
}
actions {
    preamble { id: 16799317
        name: "set_nhop" }
    params { id: 1
        name: "dstAddr" }
    params { id: 2
        name: "port" }
}
tables {
    preamble { id: 33574068
        name: "ipv4_lpm" }
    match_fields { id: 1
        name: "header.ip.dstAddr"
        match_type: LPM }
    action_refs { id: 16799317 }
    action_refs { id: 16805608 }
    action_refs { id: 16800567 }
    size: 1024
}
```

3.5 Automation of Testcase Generation using Python scripting.

Testcase is the set of condition or variable under which the tester can determine whether the system satisfies the requirement or not. Testing must be done thoroughly to check nearly all the functionalities of the module. Automation of testcase generation was done by me using the python scripting language. This automation helps to reduce the burden of Manual Tescase generation where tester has to manually create the testcase and that takes nearly 2 days to generate all the testcases. But with this automation script this laborious work was reduced to few hours.

Figure below shows the flow of the script to Automating the testcases generation.

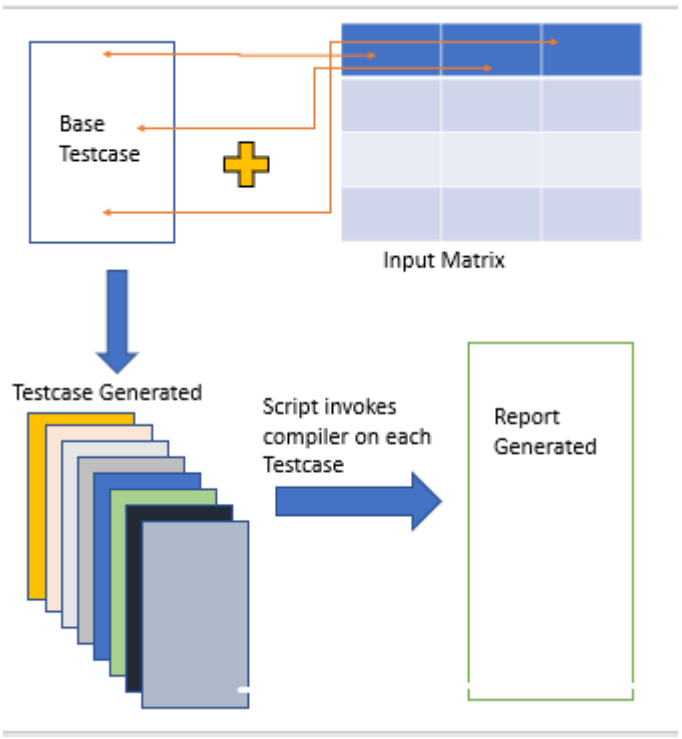

Figure 3.4: Testcase Generation Script Work Flow

In figure 3.1 the script takes one base testcase and one input matrix. The values in the input matrix are the values that user desires to fill in the base testcase. The script start with the taking values of the input matrix row wise and insert these values into the newly created .p4 file. No of testcase generated are equal to the number of the rows of the input matrix. For each testcase the base testcase open and each line of base testcase is read and value is insert from the input matrix if any python variable is found. Each testcase carries the information about it like its description, error message, and its behavior. This script takes each testcase and invokes P4 compiler on it. If the behavior of the testcase is matched then testcase is added as an Pass testcase in the report and if the some unwanted behavior is seen then testcase is added as a fail testcase. Thus along with the testcases, a text file is generated the has a record of pass and fail testcases against the current compiler.

3.6 Validation of P4 tools.

The heart of the P4 Language is its compiler. The architecture of P4 Compiler helps to make it target independent by separating language and the target model. P4 has a frontend open source and a backend loose to make it independent of target. Each chip vendor can implement its own compiler backend to map to their hardware architecture. The architecture of P4 (figure 1) also helps to insulate hardware details by allowing chip companies to define their own model and then write the P4 backend to support the same.

Above Figure 3.2, shows the data flow of P4 compiler. The open source P4 compiler

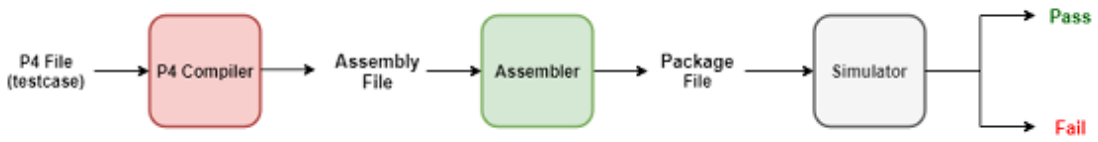

Figure 3.5: Validation Framework

is known by the name P4C. The P4C has Open Source Fronend of the compiler and the a loose Backend that can be used by vendors to wirte their own backend and get support for their own archietecure. In the above figure we pass the P4 program as input to the Parser that parses the P4 program and generates an Intermediate Representation also known as IR, a data structure to strore the extracted information from the P4 program. This IR information is then passed to the Frontend that has its own IR and is passed to the Midend that does its jobs and undergoes many user-defined passes. The Backend is made flexible so that output is used depending on user requirement. In the figure it is mentioned that Backend can be made to give either C code as output or JSON file as output or some other target specific code depending upon the user or vendor using the P4.

The packet first arrives to programmable parser which is a very simple finite state machine that starts to convert the byte stream that arrived into it into the parsed representation or set of headers. For example, it first extracts Ethernet header, then it looks at the Ethertype field and decides whether to extract an IPv4 header or extract an IPv6 header. It finishes extracting those headers and then s are passed to the programmable match action pipeline which performs the same operation all the time which is to perform a match on something and execute an action. The packet headers and metadata which are the intermediate results get transformed after each stage, and finally when the flow of control is at the last stage, there are only so many headers which probably needs to output onto the wire. So the packet goes into the last unit called the Deparser which basically assembles those headers and sends them out.

In P4, the compiler back-end is target dependent, which in our case is the SmartNIC. The P4 Compiler is being actively developed to support the Programmable blocks described above namely the Parser, Match-Action Pipeline, and the Deparser. The compiler converts P4 language file into an equivalent assembly language file, native to Intel's hardware products. The testing and validation of the P4 Compiler for functional correctness, stability,

and code coverage is assigned to me. This involves writing test cases in P4 language which will test different parameters of the hardware, test for error-checking, and test for code coverage of the compiler. More than a thousand test cases are developed to enforce the same, which is done via a test framework written in python. The framework (figure 3.3) takes P4 files and compiles them to create Assembly files. These Assembly files are converted to Package files via Intel Assembler, which is then fed into Intel's Simulator Tool. The Simulator initializes the registers from the Package file. The resultant output shows the success/failure of test cases

Chapter 4

Conclusion

i. **Functioning of a Simple Switch in P4**

This project makes a very clear understanding of the functioning of a simple switch inn P4. Successfully writing the architecture and definition of programmable blocks in P4 language also compiling and validating the generated assembly through test framework. This switch brings us the usefulness and user friendliness of the P4 language.

ii. **P4 Runtime for Control Plane**

P4 Compiler comes with a functionality to generate a runtime file that has information about the P4 program and helps control plane to add and update entries in the tables. Thus we have a complete overview of how to write a P4 program to its compile and runtime outputs. this clearly shows how powerful P4 language is.

iii. **Automating the test case generation, and report generation**

With the python automation script the test cases generation that used to take 3 to 4 day is now reduced to few hours. Tester just need to make a good base case and provide the input matrix with values and rest is taken care by the python script. I have successfully used the script with the existing test framework and it saved a lot of time and effort.

iv. **Test Framework Improvement**

Every good software is created only when it goes through rigorous and thorough testing stages. Intel Validation ensures that same for P4 Language compiler with addition of hundreds and thousands of test cases with every new control block. Not only for the backend of compiler but also for the command line options, test framework was improved time to time.

v. **On time delivery of modules without missing deadlines**

Being a part of ongoing project of Intel Technologies, I was able to successfully deliver many blocks without failing the deadlines.

Chapter 5

Future Work

P4 language is very new to the field of computer science. The first conference on P4 took place in2015 and the first stable release was made in 2019. P4 is a domain-specific language that will be used for years to come in the network world and many other fields are yet to be explored.

i. End-to-end explanation of packet processing in a Realtime Network or real Switch/NIC using P4 Programmable blocks.

ii. P4 is not only use to program the data plane but efforts are being made to expand its scope in network security also. Since packet processing is involved network engineers are trying to push boundaries to get an early defense mechanism for many security attacks[10].

iii. An automated network validation can be done using Machine Learning or Reinforcement Learning[11].

References

[1] Tausanovitch, N., 2016. What makes a nic a smartnic, and why is it needed? Netronome, https://www.netronome.com/blog/what-makes-a-nic-a-smartnic-and-why-is-it-needed/.

[2] Consortium, T. P. L., 2017. P416 language specification. P4.org, https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[3] Stephens, B., 2016. Your programmable nic should be a programmable switch. Programmable NICs, https://www.cs.uic.edu/~brents/docs/panic.hotnets18.pdf.

[4] Shah, K., 2019. P4 runtime. P4 Runtime : Future of SDN, https://volansys.com/p4-runtime-future-of-sdn/.

[5] Deierling, K., 2018. Defining the smartnic: What is a smartnic and how to choose the best one. Mellanox, https://blog.mellanox.com/2018/08/defining-smartnic/.

[6] McKeown, N., 2014. "P4: Programming protocol-independent packet processors". *ACM SIGCOMM Computer Communication Review,* **44**(3), July, pp. 88 – 95.

[7] Rijsman, B., 2019. Getting started with p4. Getting started with P4, https://p4.org/p4/getting-started-with-p4.html.

[8] Command, 2017. Command line options. Order of compiler command-line options, http://www.keil.com/support/man/docs/armcc/armcc_chr1359124195419.htm.

[9] Abdi, S., 2017. P4 program-dependent controller interface. P4 Workshop, https://p4.org/assets/p4-ws-2017-p4-program-dependent-api-for-sdn-applications.pdf.

[10] Levensalor, R., 2019. Attacks detection using p4. P4 and netwokr security, https://www.cablelabs.com/10g-integrity-the-docsis-4-0-specification-and-its-new-authentication-and-authorization-framework.

[11] Shukla, A., 2019. Validation with p4 using rl. P4 meets ML and RL, https://dl.acm.org/doi/10.1145/3341216.3342206.

0