Validation of P4 tools using Test Framework

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

Contents

List of Figures

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". Practically speaking a NIC is a PCIe card that plugs into a server or storage box to enable connectivity to an Ethernet network. Traditionally, the NIC processing is performed by the CPU, 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. At times the definition of a SmartNIC is focused entirely on the implementation, as proposed by few vendors. Least said, the 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.[1]

Programming Protocol-independent Packet Processors (P4) is a declarative language for expressing how packets are processed by the pipeline of a network forwarding element, such as a switch, NIC, router or network function appliance. Depending on the flexibility and complexity of the appliance, the data plane of such systems can run the from fixed-function ASICs (such as those used in L2 switches) to fully programmable, general purpose CPUs found in whitebox implementations of routers, web proxies and firewalls, also known as NPU's. The designers of P4 state that it is a programming language intended for packet processors, irrespective of the packet processing device- Hardwired ASICs or NPUs, provided they can interpret P4.

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 perform three basic operations: packet parsing, match/action operations, and packet reassembly. P4 provides coding constructs that make describing these

operations easy to understand. And because P4 is an actual language, users have the ability to define the packet header structures the parser will extract. This is a vast improvement over protocols like OpenFlow which can only parse the headers for supported network protocols. Ethernet, IP or TCP headers are supported officially by OpenFlow. If there is a need to parse an NVGRE packet, OpenFlow needs to add support for that protocol. Contrasting this with P4, P4 allows to specify the header stack of the packets that are flowing, and parse it. This ability is known as protocol independence. Moreover, the ability to write new P4 source code and rules, compile it and push it to the data plane of a system 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 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.

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, 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.

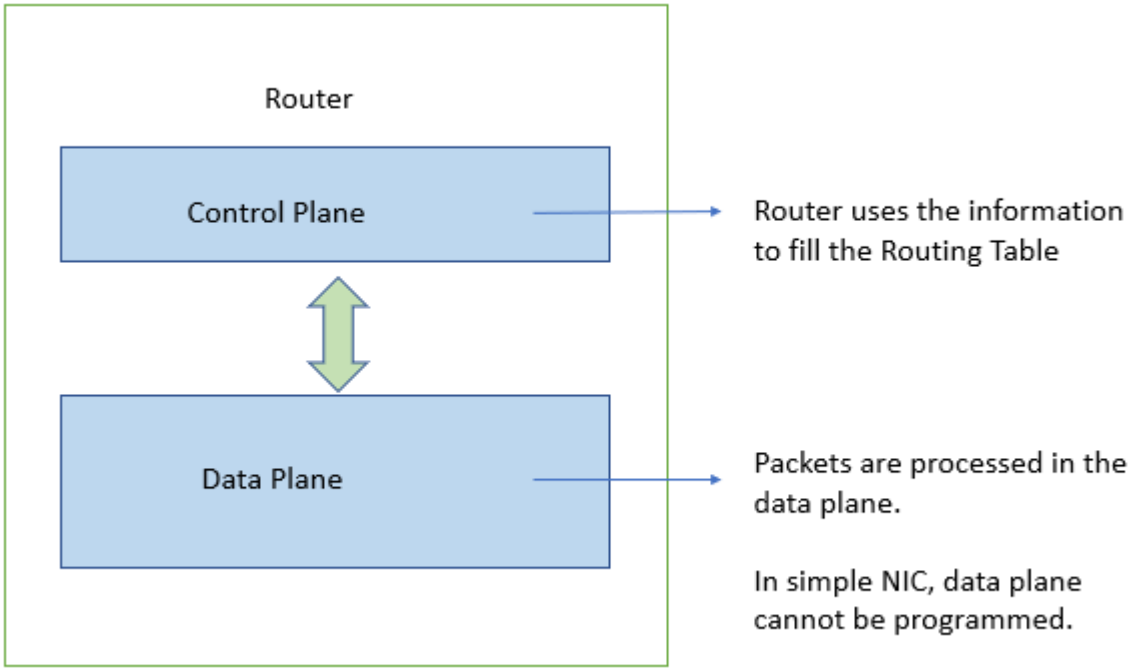

Figure 2.1: Networking Planes

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.[4] 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 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.

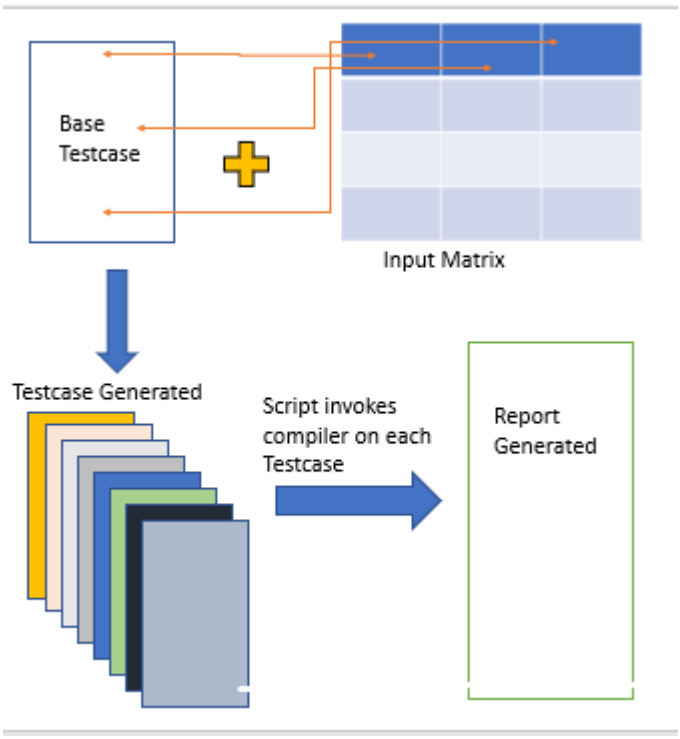

Figure 3.1: Testcase Generation Script Work Flow

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.

3.2 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.[5] 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.[5] 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();
    }
```

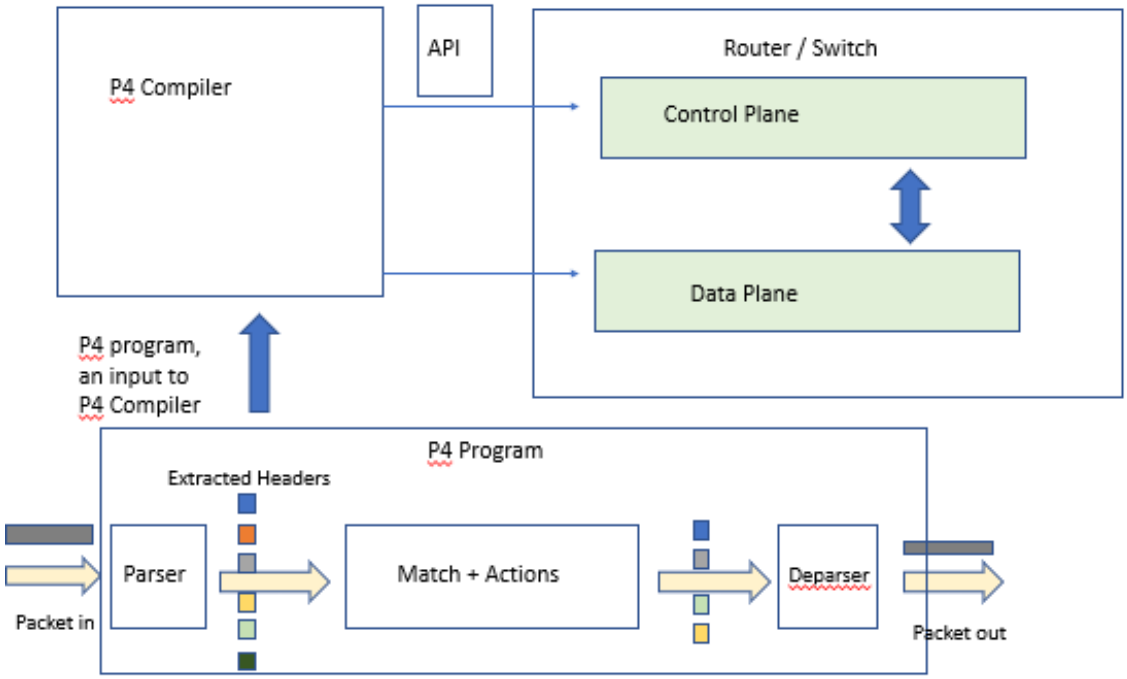

Figure 3.2: Compiler Data Flow

3.3 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 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

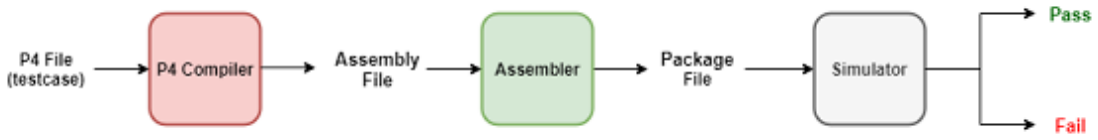

Figure 3.3: Validation Framework

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. **Automating the testcase generation, and report generation**

With the python automation script the testcases 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.

ii. **Test Framework Improvement**

Report generation of Test Cases listed in alphabetically sorted order: The Test Framework was modified to collect all the files in the directories and the sub-directories that are relevant to architecture, store them in a dictionary, sort them in alphabetical order and forward it to the testing and validation functions.

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

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

Chapter 5

Future Work

i. End-to-end explanation of packet processing in a Simple Switch/NIC using P4 Programmable blocks.

ii. Test case development for other programmable Blocks/Tools.

iii. Involvement into Compiler Block Development and Bug fixes.

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] 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/.

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

0