修　士　論　文　の　和　文　要　旨

研究科・専攻	大学院　情報システム学研究科　情報ネットワークシステム学専攻　博士前期課程		
氏　　　名	蕭　昊駒 (Xiao Haoju)	学籍番号	1552012
論 文 題 目	A Study on Packet Classification Using the P4 Language (P4 言語を用いたパケット分類アルゴリズムに関する研究)		

要　　旨

パケット・クラシファイアとは、コンピュータネットワークにおいてネットワーク機器に到着したパケットをグループに分類するメカリズムである。特定の処理のためにパケットを区別して分離する必要があるサービス、例えば、ファイアウォールやサービス品質などのカスタマイズネットワークサービスなどを提供するためにルータでのパケットを分類するのは極めて重要である。

パケット分類に関するアルゴリズムがいくつかの研究で提案されている。分類の性能を向上するため、決定木、ヒューリスティックなどを利用した提案がある。しかし、その性能評価は主にハードウェア実装に基づいていたので、アルゴリズムの設計方法、データ構造などソフトウェルーターに適用できない恐れがある。

近年、ネットワークプロトコル、ターゲット非依存という特徴をある P4 言語が開発された。P4 言語は幅広いのデータプレーンをプログラミングできるように、ネットワークの基本機能に関する表現力豊かな文法設計されています。仮想ネットワーク機能（VNF）に対する研究が流行っている背景のなか、P4 言語用いてソフトウェアにおけるパケット分類の実装を研究する必要がある。

本研究では、今までネットワークのパケット分類に関するアルゴリズムが P4 言語文法による実装を検討する。P4 抽象転送モデル中で利用可能なプログラミングフローを議論し、パケット分類の改善に適しているデータ構造を示した。また、異なるアルゴリズムとデータ構造を用いて、P4 ソースコードからコンパイルされたソフトウェアルーターの性能評価を行った。

平成２８年度修士論文

A Study on Packet Classification Using the P4 Language

P4 言語を用いたパケット分類アルゴリズムに関する研究

大学院情報システム学研究科情報ネットワークシステム学専攻

学籍番号：	1552012
氏名：	蕭　昊駒 (Xiao Haoju)
主任指導教員：	Ved Kafle 客員准教授
指導教員：	加藤　聡彦
指導教員：	大坐畠　智
提出年月日：	平成２９年１月２６日(木)

Abstract

Packet classification is crucial for routers to provide customized network services such as firewall and quality of service that require to distinguish and isolate packets for specific processing. In order to classify packets belonging to a certain traffic flow, routers have to perform a search on a set of filters by examining packet header fields.

A number of algorithms concerning packet classification have been proposed in previous studies. However, their evaluations are mostly based on hardware implementation. The P4 language, a higher-level language for Programming Protocol-independent Packet Processors is gaining popularity recently and has the characteristics of flexible expression of network functions that are useful in packet classification.

In this thesis, we present our study on the implementation of packet classification using the P4 language. In particular, we discuss the programming flows available in the P4 language description and illustrate the data structures are suitable for improving the packet classification. We have implemented software routers compiled from P4 source code using different algorithms. We evaluate the performance of the virtual machine on which the software router is running regarding CPU utilization, memory cost and latency. Additionally, we compare different counting mechanisms of our proposed idea. Experiment results show that our proposal has obvious advantage of lower computation cost and reduced latency with slightly more memory space consumption.

Acknowledgements

I would like to extend my deepest appreciation to my supervisor Professor Ved Prasad Kafle, a senior researcher at the National Institute of Information and Communications Technology (NICT). Without his persistent guidance and encouragement on my research, this thesis would not have materialized.

I would also like to thank the P4 Language Consortium, which is an organization promotes the development and provides great materials for my research.

Moreover, I express my gratitude to Antonin Bas, a major contributor to the P4 language's repository on GitHub.com, along with all members of the P4 language's E-mailing list. They offered help and motivations on setting up my experimental environment.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter 1

INTRODUCTION

As modern networks involve heterogeneous services and exponentially increasing data volume, the techniques to ensure network service quality are in high demand. A major approach to guarantee network services is through network traffic engineering. An important objective of Internet traffic engineering is to facilitate reliable network operations [1]. This can be done by embracing policies emphasizing packet routing table and switching packets from one incoming port to particular outgoing port.

Packet classification was formally thought necessary only at network access points and used mainly for firewall and security applications. However, the recent tread indicates that it is equally valuable for traffic engineering and various network services. Nonetheless, due to the complexity of the filtering, packet classification is often a performance bottleneck in network infrastructure [2].

The P4 language [3], which the acronym for Programming Protocol-independent Packet Processor, was proposed as a high-level language for network devices, i.e. the data plane in the Software-Defined Networking. It is characterized by its (1) Reconfigurability, (2) Protocol independence and (3) Target independence, providing opportunities for flexible network architecture innovations.

In this thesis, we are going to evaluate different packet classification algorithms implemented in the P4 language. Since P4's compiler might already have certain optimization on data structure, some classification algorithms may not offer much improvement as claimed in their original paper. Based on our preliminary evaluation, we will propose new algorithm, which is thought to be suitable for efficient classification under P4's architecture.

The rest of the thesis is organized as follows. We briefly discuss related works in chapter 2. In chapter 3, we demonstrate the architecture of P4 language and how data packets are processed in the P4 behavior model. In chapter 4, we illustrate P4's major components in details and show the challenges and theoretical improvement of its implementation in P4. We present our proposal on packet classification using the P4 language in chapter 5. Experimental environment setup and evaluation results will be discussed in chapter 6. Finally, we draw our conclusion in chapter 7.

Chapter 2

RELATED WORKS

2.1 Extending SDN to the Data Plane

Software Defined Network (SDN) [4] [5] is changing the way networks are designed and managed. It has two defining characteristics. First, SDN separates the control plane (which decides how to handle the traffic) from the data plane (which forwards traffic according to the decisions that the control plane makes). Second, SDN consolidates the control plane, so that a single software control program controls multiple data-plane elements [6].

The previous research on extending SDN to the data plane [7] points out that applications, with new queuing strategies, scheduling techniques, and endpoint control protocols, might require customized switch functions, such as a different trade-off between throughput and delay, and an emphasis on throughput or delay variation above other metrics. The idea is to extend SDN control functions to the data plane, and to allow network operators to align queuing and scheduling behavior with application requirements. Although the study focuses on hardware switches instead of software ones, it indicates that data-plane should be programmable for better flexibility.

The RMT architecture [8] is an example of programmable forwarding planes for switches. The research in this area includes the Tiny Packet Program interface [9] for low-latency network monitoring and control, PLUG [10], providing a programmable set of lookup modules, and a proposal to add an FPGA for programmable queue management and scheduling.

2.2 Packet Processing Languages

NetASM [11] is an intermediate representation for programmable data planes. NetASM is a device-independent language that is expressive enough to act as the target language for compilers for high-level languages, yet low-level enough to be efficiently assembled on various device architectures.

Protocol-oblivious forwarding (POF) [12] [13] was proposed as a key enabler for highly flexible and programmable SDN. It aims to remove any dependency on protocol-specific configurations on the forwarding elements and enhance the data-path with new stateful instructions to support genuine software defined networking behavior.

The P4 language [3] is relatively a new concept at the time. It aims to provide a description of customized packet processing functionality for configurable switches. Some studies had been carried out using this flexible mechanism. A generator [14] of high-speed packet parser suitable for FPGAs. The generators output is a synthesized VHDL code that performs packet parsing as defined by the P4 program. A mapper [15] converts logical tables defined by the programmer in P4 language to physical tables on the switch for various architecture. Another study [16] reveals that a P4 program can render promising performance on such architecture by match+action parallelism with the General Purpose Graphics Processing Unit (GPGPU) accelerator.

Use cases of P4 complied forwarding elements include In-band Network Telemetry (INT) [17] [18], a new abstraction that allows data packets to query switch-internal state such as queue size, link utilization, and queuing latency.

2.3 Linear Packet Classification Approach

The most simplest and straightforward classification algorithm is the linear search. Its data structure is simple and tables are easy to be organized or updated. It has the advantage of $O(N)$ storage requirement, but it also has non-negligible $O(N)$ memory access time cost. In theory, by organizing classification rules into a number of pipeline stages, memory access cost can be reduced by a small factor (i.e. $O(N/d)$). Among some classification algorithms, linear search is also popularly utilized in their final processing stages [19][20][21].

2.4 Decision Tree Packet Classification Approach

2.4.1 The Tree Structure

The decision tree approach is implemented by using tree data structure, where subsets of classification rules reside on the leaves. Branching decisions are made on each nodes during the traversing process. Once a certain leaf is reached, an appropriate filtering is performed. Some algorithms related to this approach view filters with d fields as a subset of *d-dimensional* space. Thus cuttings in multi-space are functionally equivalent to branching decisions in tree structure.

Generally speaking, given a tree structure specified with b branching decisions, the searching time cost is $O(b)$ while the memory cost is $O(2^b)$ theoretically. But the actual memory requirement varies greatly with the branching complexity of the data structure. And as filters are located by traversing from the root to the leaf in a serial operation, it is not possible to implement decision tree with parallelism. However, if we have a pipeline structure where each pipeline stage has independent memory interface, it might yield greater throughput.

Another problem concerning this approach is that different filtering might follow different type of search algorithms, e.g. longest prefix match, arbitrary range match and exact match, which can complicate the tree structure. A common solution to this problem is to convert match conditions to a bit vectors with arbitrary bit masks, that is, bit vectors where each bit may be a 1, 0, or a mask [22].

Grid-of-Tries [23] is a packet classification algorithm that applies a decision tree approach. It performs optimally especially for processing filters of two prefix fields. For instance, for filters that are defined by destination and source address prefix, *Grid-of-Tries* outperforms the directed acyclic graph (DAG) [24] technique in that some redundant tree structure is reduced by allowing multiple accesses to leaves (filters). Though this optimization does not cut all structural redundancy, it eliminates filter replication on leaf nodes by storing filters in one place and utilizes pointers for access. Given that F is the number of filters and B is the number of bits specified in the destination and source fields, *Grid-of-Tries* has a memory cost at $O(FB)$ and a time cost at $O(B)$. While *Grid-of-Tries* is efficient in classification on two-address prefix, it cannot complicate the search with additional fields.

2.4.2 Extended Grid-of-Tries

Extended Grid-of-Tries (EGT) [25] supports multiple field search without the need of multiple instance of data structure, and offers more flexible pointers to target filter set compared with *Grid-of-Tries*. EGT has pointers that can direct the search to all possible matching filters, instead of only one kind of matching manner (e.g. destination and source prefix).

2.4.3 Hierarchical Intelligent Cuttings

Hierarchical Intelligent Cuttings (HiCuts) [19] introduces the cutting concept for packet classification from a geometrical viewpoint. Each filter in the filter set defines a d-dimensional rectangle in d dimensional space.

HiCuts approach has to handle the filter set beforehand to build a tree structure with leaves containing subsets of filters. The traversing decision is based on header field information. Various heuristics at nodes where branching decisions are made, reducing the complexity, redundancy and memory cost of the tree structure as a result.

2.4.4 Modular Approach

A flexible framework for packet classification [20] takes a modular approach consisting three stages: an index jump table, search trees, and filter buckets. The classifications are initiated by examining certain fields of the packet header to produce an index in the jump table. And the search continues to the search tree if selected entry has a valid pointer, or filter actions are applied otherwise. Branching decisions are made on each tree node based on specified header fields. When a filter bucket is reached, filters are chosen from the filter set (located in the bucket) in a manner of linear search. This approach has the advantage of balancing tree structure and access cost by placing frequently used filters in the bucket.

2.4.5 HyperCuts

HyperCuts algorithm [21] is a decision tree algorithm that tries to control the depth of the tree structure. It was improved upon HiCuts algorithm [19] and modular classification approach [20]. But unlike HiCuts, each node in the HyperCuts decision tree represents a k-dimensional hypercube, i.e. each branching node can have multiple cuts on the multidimensional search space. HyperCuts also encode pointers in an efficient way by forcing cuttings to create uniform sub-space, which is one of the reasons why multidimensional cuttings are possible without significant memory cost.

2.4.6 Fat Inverted Segment Trees

Fat Inverted Segment (FIS) Trees [26] uses independent field search for packet classification. FIS Trees view filter set from a geometry viewpoint and map filters into multidimensional space. The algorithm starts by constructing a FIS tree on one dimension. For each node with non-empty filter set, a FIS tree is built with a minimum interval formed by the projection of the next dimension.

2.5 Discussion

Previous works on packet classification are mostly focus on examining the packet headers with different approaches. In the case of hardware implementation, such low-level optimization could yield significant improvement. However, for software routers running on various infrastructures, packet-level might not give much performance advantages. Since their network behaviors are often described in high-level languages, low-level data structure could be perfectly optimized by corresponding compiler. In this thesis, while we consider the possible benefit of previous approaches, much emphasis is put on traffic flow level classification. In other words, we seek classification improvements from a specific traffic flow approach.

Chapter 3

THE P4 LANGUAGE

3.1 General Introduction

P4 is a high-level language for programming protocol-independent packet processors, expressing how packets are processed by the pipeline of a network forwarding element such as a switch, NIC, router or network function appliance [27].

It is designed to allow the development of packet forwarding data planes in the SDN architecture [6]. In contrast to a general purpose language such as C or python, P4 is a domain-specific language with a number of constructs optimized around network data forwarding. P4 is an open-source, permissively licensed language and is maintained by a non-profit organization called the P4 Language Consortium [28]. P4 has the ambition of being used to program the data plane of many different networking targets. It aims to provide the following features [3]:

- **Target Independence** P4 programs are designed to be implementation-independent, meaning they can be compiled according to many different types of network infrastructure such as general-purpose CPUs, FPGAs, system(s)-on-chip, network processors, and ASICs. These different types of machines are treated as P4 targets, and each target must be provided along with a compiler that maps the P4 source code into a target switch model.

- **Protocol Independence** P4 is designed to be protocol-independent, meaning that the language has no native support even for common protocols such as IP, Ethernet, TCP, VxLAN, or MPLS. Instead, the P4 programmer describes the header formats and field names of the required protocols in the program, which are in turn interpreted and processed by the compiled program and target device.

- **Reconfigurability** The protocol independence and the abstract language model allow for reconfigurability P4 targets by changing the way they process packets after they are deployed. This capability is traditionally associated with forwarding planes built on general-purpose CPUs or network processors.

3.2 Proposal Background

P4 is a language for it resolves the problem of increasing header fields recognized by OpenFlow [29] standard and enables programmer to easily redefine the forwarding behavior of network devices regardless of the specifics of the underlying hardware. That further implies greater portability of software router compiled in P4 language.

Software-defined networking (SDN) is an approach to computer networking that allows network administrators to programmatically initialize, control, change, and manage network behavior dynamically via open interfaces [5] and abstraction of lower-level functionality. SDN is commonly associated with the OpenFlow [29] protocol, which enables network controllers to determine the path of network packets across a network of switches.

The OpenFlow Protocol serves as an interface to provide an open and standard way for a controller to communicate with a switch [29]. OpenFlow is the de-facto standard to control the network, originally targeting local area networks, and data centers in particular. The development of OpenFlow tries to cover all scenarios in networking, but this approach comes at a cost of complexity. For instance, the original list of OpenFlow supported fields has now grown to more than 40 protocols as shown on Table 1, mostly due to packet encapsulation that must be processed by switches when looking for the label.

Version	Date	Header Fields
OF 1.0	Dec. 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb. 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec. 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun. 2012	40 fields
OF 1.4	Oct. 2013	41 fields

Table 1: Fields recognized by the OpenFlow standards

Rather than repeatedly extending the OpenFlow specification, future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through an open interface [3]. As a consequence, the P4 language was proposed, raising the level of abstraction for programming the network by instructing the switch how to operate, rather than be constrained by a fixed switch design.

3.3 Basic Components

P4 addresses the configuration of a forwarding element. Using relatively simple syntax, P4 allows defining five major aspects of packet processing.

3.3.1 Header Definitions

P4 requires each header instance to be declared explicitly prior to being referenced. Header types describe the layout of fields and provide names for referencing information. There are two sorts of header instances: packet headers and metadata. Usually, packet headers are identified from the packet as it arrives at ingress while metadata holds information about the packet that is not normally represented by the packet data such as ingress port or a time stamp.

Most metadata are simply per-packet state and are used like Registers (discussed in Section 4.2) in P4 while processing a packet. However, some metadata may have special significance to the operation of the switch. For example, the queuing system may interpret the value of a particular metadata field when choosing a queue for a packet. P4 acknowledges these target specific semantics, but does not attempt to represent them.

Packet headers (declared with the header keyword) and metadata (declared with the metadata keyword) differ only in their validity. Packet headers maintain a separate valid indication, which may be tested explicitly. Metadata are always considered to be valid.

3.3.2 The Parser

P4 defines the parser as a state machine, describing the conceptual logic used to traverse packet headers from start to end, extracting field values accordingly. A node in the parse graph may be purely a decision node and not bound to a particular header instance, or a node may process multiple headers at once.

The parser produces the *Parsed Representation* of the packet on which match+action pipelines operate. Match+action may update the Parsed Representation of the packet by modifying field values and by changing which header instances are valid; the latter results in adding and removing headers.

In P4, each state is represented as a parser function. A parser function may exit in one of the following four ways:

- A transition to another parser function by specifying the name of a parser function is executed.

- Terminates parsing and begins match+action processing by calling the indicated control function.
- An explicit or implicit error occurs.

3.3.3 Table Definitions

Tables are the most fundamental units of the Match+Action pipeline. Tables in P4 define the type of lookup to perform (e.g. exact match, prefix match, range search), the input fields to use, the actions that may be applied, and the size of each table. Associated with each entry is an indication of an action to take should the entry match. If no entry is found that matches the current packet, the table is said to ”miss”; in this case a default action for the table may be applied.

3.3.4 Action Definitions

In P4, actions are declared imperatively as functions. These function names are used when populating the table at run time to select the action associated with each entry. They often compose compound actions from a set of primitive actions.

3.3.5 The Control Program

The control program organizes the layout of tables within ingress and egress pipeline, and the packet flow through the pipeline. At configuration time, the control flow (in what order the tables are to be applied) may be expressed with an imperative program. The imperative program may apply tables, call other control flow functions or test conditions.

3.4 Abstract Forwarding Model

P4 abstract forwarding model describes the operations of the P4 machine from a high-level point of view. For an incoming packet, the Parser produces a *Parse Representation* on which match+action tables operate. During the ingress pipeline processing, an *Egress Specification* is generated by match+action tables. After the *Egress Specification* being processed by *Queuing Mechanism*, instances of the packet are created and sent to *Egress Pipeline*. Once the packet has entered the *Egress Pipeline*, its destination is assumed fixed except that the packet is dropped during *Egress Pipeline* processing. Finally, the packet with a new header formed by *Parse Representation* (a process also known as Deparsing) is transmitted.

Chapter 4

CLASSIFICATION IMPLEMENTATION IN P4

This chapter discusses the data structure that can facilitate packet classification from a P4 description. We will demonstrate some of the characteristics of metadata in Section 4.1. Section 4.2 describes how to construct particular stateful memories. And we will also mention the feasible control flow for implementing some previous algorithms.

4.1 Metadata

Along with packet headers, metadata is another type of header instance. Metadata usually stores information that is not commonly represented in packet data, e.g. ingress ports or a timestamp. Carried by the metadata bus (Figure 1), most metadata are used for indicating per-packet state, and it is accessible across tables within processing pipeline. Metadata are considered stateless, because they are reinitialized for each packet. Some metadata may have special significance to the operation of the device, but that is not irrelevant to our study.

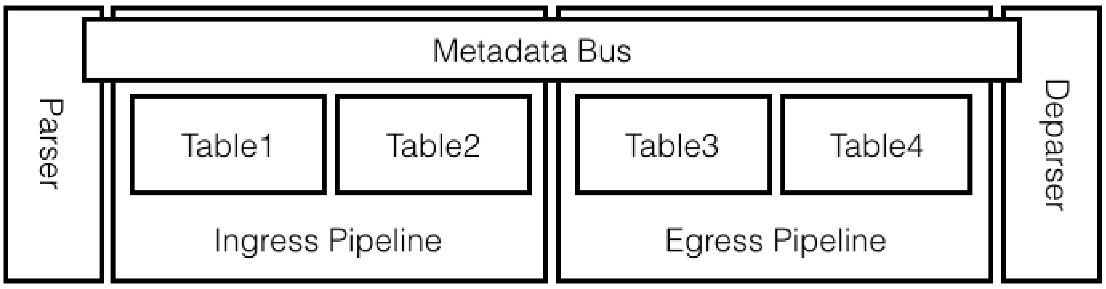

Figure 1: Metadata Bus

4.2 Stateful Memories

Stateful memories in P4 include Counters, Meters and Registers, also know as a *cell* individually. They are categorized as stateful because they can keep their information across packets. There are two key attributes associated with these objects: *static* and *direct*. The *static* attribute indicates that the cell is bound to a specific table, or accessible to all tables if the attribute is absent. The present of attribute *direct* means that each entry in the table is allocated with one instance of specified cell. For our work, Counters and Registers are relevant for classification implementation. Counters are incremented by one unit for each packet, while Registers can be used for a more general purpose to store arbitrary data. Our classification implementations rely heavily on Registers because of its versatility.

4.2.1 Counters

With the attribute *direct* specified, a direct counter may be declared like this:

```
counter direct_counter {
    type : packets_and_bytes;
    direct : target_table;
}
```

The description above declares a set of *direct_counter* counters attached to the table named *target_table*. The *packets_and_bytes* attribute indicates that the counter is comprised of two sub-counters internally, and each sub-counter is incremented by the packet length and by one respectively. In this case, no actions is needed for handling the counting operations since they are automatically incremented once the corresponding entry is matched. And this counter is not allowed to be referenced by primitive actions, otherwise compiler errors would incur.

Counters can also be declared with *indirect* attribute, which means the counter is globally accessible among multiple tables. A sample table with indirect counter may be declared as follows:

```
counter indirect_counter {
    type : packets_and_bytes;
    instance_count : INSTANCE_NUMBER;
}
```

Generally, different table entries, even they do not reside in the same table, may refer to the same counter. This process is also called *indirect access*.

4.2.2 Registers

Registers are stateful memories whose values can be read and written in actions. They are like counters, but can be used in a more general way to keep state. Registers are declared with a width attribute that indicates the bit-width of each instance of the register. A sample declaration may looks like this:

```
register static_register {
    width: 32;
    static: target_table;
    instance_count: 1024;
}
```

4.3 The Flow Control for Classification

Match+action tables are organized in an imperative program named Control Flow Program in P4. The imperative program can direct the execution of tables, and perform test conditions or call other control flow. Standard control flow functions include *ingress* control function and *egress* control function. Though programmers might define customized control function to improve readability, no performance advantage can be gained since the compiler would fatten the code.

Since packet processing is determined by a sequence of match+action tables and control flow can direct the order of tables execution, it is clear that packet classification algorithms can be implemented by defining tables and programming control flow. In this thesis, however, the tables' definition is not of much concern, because we believe flow control would play a larger role in refining classification than particular kind of filtering. Modifications on packet header can happen in both ingress pipeline and egress pipeline, but the difference is that egress port is determined at the end of ingress pipeline and assumed not to be altered in the egress pipeline unless recirculation occurs.

Though the imperative control flow representation is a convenient way to specify the logical forwarding behavior of a switch but does not explicitly call out dependencies between tables or opportunities for concurrency [3].

4.3.1 Linear Flow Control

The simplest and straightforward organization of match+action tables is linear flow control, as shown in Figure 2. In the imperative control flow, the execution of tables are implied by *apply* instruction statement. Linear organization means the control flow moves to next statement sequentially and unconditionally. And in any case of table miss is encountered, the data packet is assumed dropped, terminating the search flow. The linear design comes with the advantage of less implementation cost and therefore its deployment speed has an edge on production services, especially when we consider that hardware-computing resources are often abundant. On the other hand, when we have a large number of rules on each table and a non-negligible number of tables, the search time cost can grow linearly, which results in low performance in a heavy network flow.

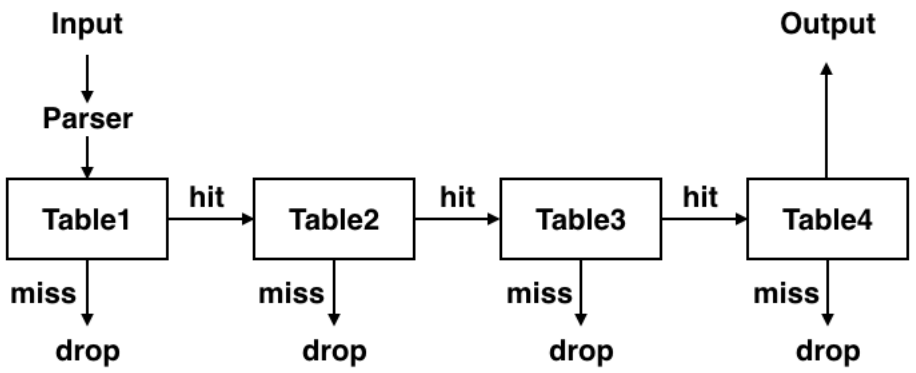

Figure 2: Linear flow control in P4

4.3.2 Decision Tree Flow Control

Decision tree implementation especially the branching decision might be complicated by the fact that a filter may specify various types of searches, e.g. a mix of the longest prefix match, arbitrary range match, and exact match conditions [2]. Since branching is the key for decision tree implementation, several syntax patterns allowed in P4 language should be studied beforehand.

In the flow control program, branching can be achieved by evaluating the information concerning a data packet, e.g. header fields and metadata. Packet header fields are predefined in a P4 source file and extracted by parser before a packet is entering match+action stage. Metadata is another kind of header instance, it holds state information about a packet that is not normally represented by packet data, e.g. ingress port and timestamp. Also, branching evaluation can be based on stateful memories, i.e. counters, meters, and register, which can store information and stay persistent across packets. The Figure 3 illustrates this category of branching decision. The packet processing flow from *table0* to either *table1*, *table2*, or *table3*, based on the evaluation of header fields, metadata or stateful memories (counters, meters, or register). In the coding implementation, such evaluations can be achieved by using nested if/else statement.

Another category of branching is determined by which table action is taken, as shown as an example in Figure 4. In P4, actions are declared imperatively as functions, which are used when populating the table at run time to select the action associated with each entry. Note that even

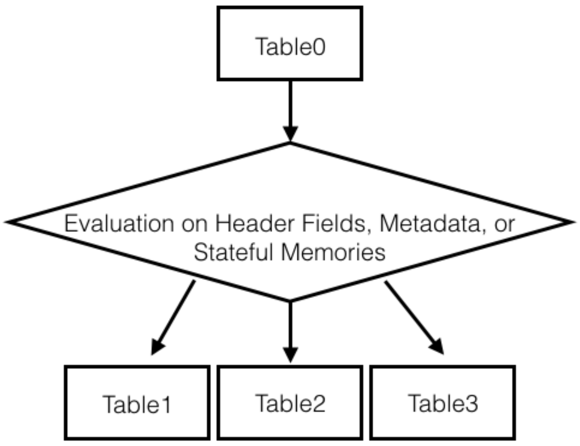

Figure 3: Nested if/else Branching

if no entry matches in a table, a specified action can also be taken, which is commonly named default action.

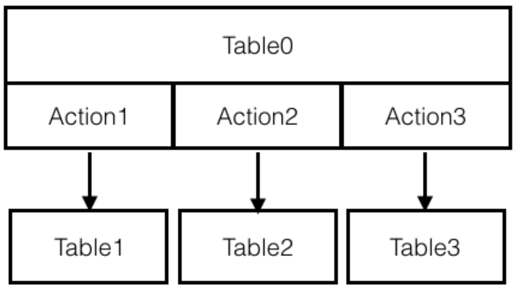

Figure 4: Branching Based on Table Actions

The second kind of branching depends on table hit or miss event, as illustrated in Figure 5. In the process of search inside a table, if an entry is found to be matched with the incoming packet, a *table hit* event is triggered; while if no entry is found that matches the current packet, the table is said to have a *table miss* event. Note that table entries are configured at run time, and they are not part of the P4 source code.

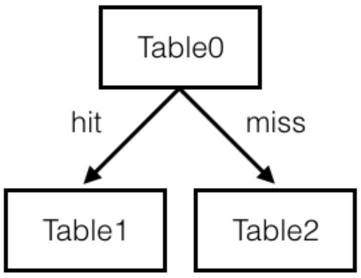

Figure 5: Branching Based on Table Hit/Miss

Chapter 5

OPTIMIZATION

5.1 Problem Statement

The packet classification algorithms discussed so far focus on data packet level, i.e. classifying every data packets based on their header information. However, it is common for a traffic flow to exist for a while, which means there are abundant packets with the same header data. Thus, classification on packet-level would incur significant processing redundancy.

5.2 General Proposed Idea

Our optimization towards classification is paying more attention to the concept of traffic flow and heuristics for classification, so as to reduce the processing redundancy found in packet-level classification.

Heuristics stand for strategies using readily accessible information to control problem-solving processes in man and machine [30]. A use case of heuristics is to make algorithms faster on searching problems where a preliminary best choice is far less costly than complete search. In our study, though it is not possible to certainly predict header information of an incoming packet, we can use indicators that based on available data at the time.

The general idea (Figure 6) is counting incoming packets belonging to the same traffic flow until a certain threshold is reached, where a flow is considered recognized. Once the flag (i.e. *flag_flow_detected*) indicating whether any flow is recognized is set to *true*, subsequent packets will be processed in the flow verification module. If the packet is verified as belonging to one of recorded traffic flows, it will be processed under a short-cut way, i.e. being modified with corresponding pre-stored egress specification and sent to the egress pipeline. For other cases in our proposed logic, packets will be handle with regular packet classification algorithms.

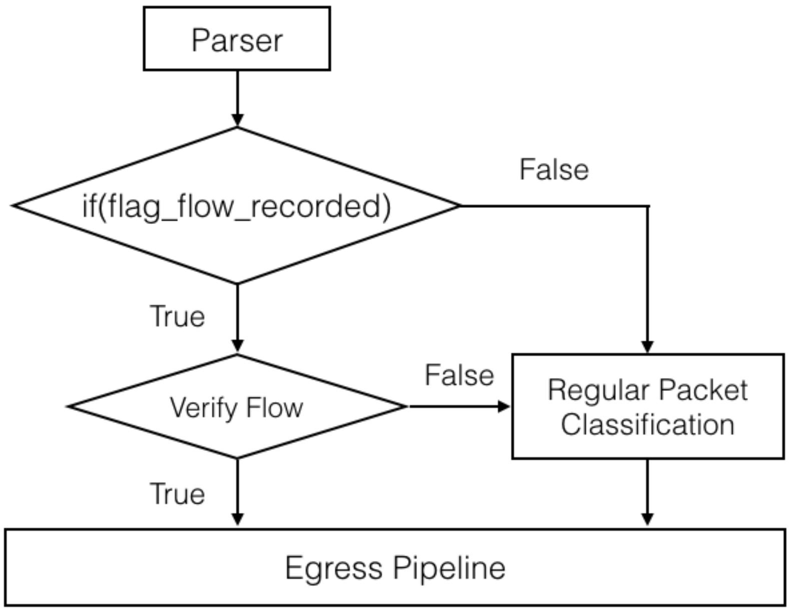

Figure 6: General Logic of New Proposal

5.3 Recognizing a Traffic Flow

There are various ways to detect and determine certain traffic flow using P4, and we will discuss two methods both of which utilize the stateful memories in P4. Stateful memories include counters, meters, and registers; their values can be persistent across packets, which differ from characteristics of meta-data.

5.3.1 Using Direct Counters for Counting

The first implementation method is using direct counter attached to a specific table as shown in Table 2. The counter allows keeping a measurement of every table match value that occurs in the data plane. If the counter is declared with the direct attribute, one instance of the counter is associated with each entry in a referenced table. In this case, no action needs to be given for the counter increment; they are automatically updated whenever the corresponding entry is matched. If any counter had reached a pre-configured threshold, a continuous data flow is assumed.

The major positive aspect of using direct counter is that it is quite easy for implementation, and we can get an accurate count on each entry matching. However, if an incoming packet does not belong to any entry specified in the table, the direct counter will not work. And inserting new entries when the router is running would involve the interaction with control plane, slowing the processing time significantly. Thus we do not consider using control plane software to address this issue.

Entries	Match Conditions	Counters
Entry 1	192.168.1.0/24	counter 1
Entry 2	192.168.2.0/24	counter 2
Entry 3	192.168.3.0/24	counter 3

Table 2: A sample table with direct counter

5.3.2 Using Bloom Filter for Counting

The second implementation method is utilizing Bloom filter to detect a traffic flow. A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Bloom filters allow false positives but the space savings often outweigh this drawback when the probability of an error is controlled [31][32].

Figure 7 illustrates the bloom-filter implementation. In P4's ingress control program, once a packet finished being parsing and sent to the ingress pipeline, we will calculate a hash value based on source address, destination address and protocol field of IPv4 protocol, along with source port number and destination port number of TCP protocol. Knowing that a register data structure is referenced by its array name and index, we convert the hash value to a valid index value with a modular operation. Then we check the register value located in computed index if it has reached a pre-configured threshold. We will increment the register value for a few incoming packets, and a new traffic flow is assumed once the threshold is crossed. Also, to reduce to chance of possible hash collision, which will result in false increment operations, we utilize two hash functions in our implementation at the same time.

5.4 A Short-cut Packet Handling

Once a new traffic flow is recognized, a proper short-cut processing is needed for efficient handling on subsequent packets. Since after being processed by ingress pipeline, a packet's destination is assumed fixed, we focus our attention on ingress pipeline only in improving classification.

In the case of using direct counter, we store the egress specification (i.e. egress port, MAC address) along with a hash value based on some header fields to a register data structure (Table 3) when counters meet the threshold on the routing table. For the following packets, we firstly compute their hash value to check if they belong to the traffic flow recorded previously. While in

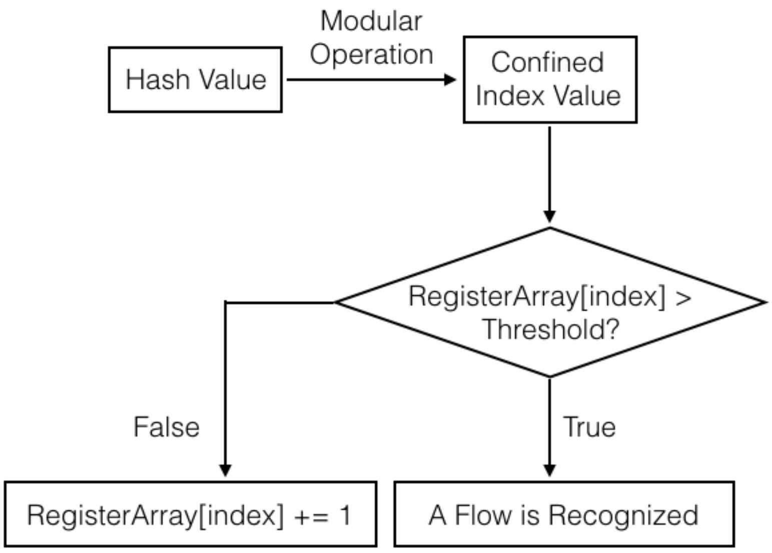

Figure 7: A Bloom Filter Structure for Counting Packets

the case of using Bloom-filter, the register data structure is quite similar to the previous scenario except that the index is calculated based on hash value, and to avoid hash collisions, we deploy two hash functions to identify a certain flow and the packet counting at the same time. Then we modify packet's egress specification with the egress port value stored in the register if the corresponding hash value matches, or neglect the packet if no match is found and process the packet normally. With an early-configured egress specification, the packet can bypass further tables' filtering and finish being processed in the ingress pipeline.

Index	0	1	2
Flow	Flow 0	Flow 1	Flow 2
Hash Value	Hash_0	Hash_1	Hash_2
Egress Port	Port 0	Port 1	Port 2

Table 3: Register data structure for storing egress port when using counter

In the next chapter, we will evaluate the two implementation approaches of our proposal, along with several classification algorithms discussed in the related works.

Chapter 6

EXPERIMENT

6.1 Environment Introduction

6.1.1 The P4 Behavior Model

The P4 Behavior Model is a software switch open-sourced by the P4 Language Consortium Community [28]. The first version (p4c-behavioral: github.com/p4lang/p4c-behavioral), which has been deprecated, was originally designed as a compiler to generate executable software switch for each P4 program. It assumed a fixed abstract switch model with two pipelines (ingress and egress). Later, the second version (github.com/p4lang/behavioral-model) (a.k.a BMv2, shown in Figure 8) was published to take over its predecessor in that it is target-independent and, therefore, has static advantages, i.e. an executable switch program that can interpret any P4 program and behavior accordingly. Thus, the BMv2 can be used to reproduce the behavior of networking devices easily by programming in P4 language.

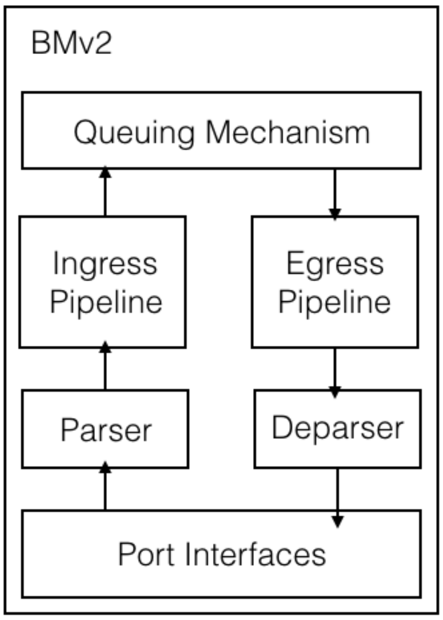

Figure 8: The P4 Behavior Model Version 2 (BMv2)

6.1.2 Experiment Work Flow

Besides for the implementation of P4 abstract forwarding model, BMv2 also includes a program-independent command line for populating the match + action tables. As depicted on Figure 9, the BMv2 has a controller that acts as the server side and the BMv2 CLI that used as the client side. They are connected with a TCP socket under the Apache Thrift Framework [33].

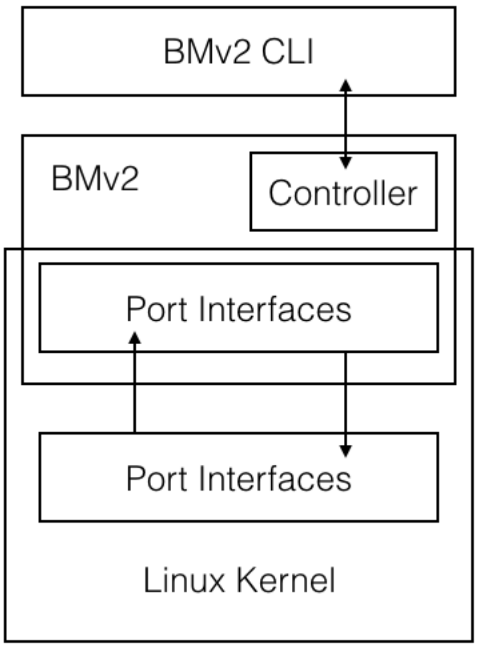

Figure 9: General Environment Setup

In order to enable the BMv2 to communicate with outside network, we use the virtual Ethernet interfaces in the Linux Kernel (Figure 9). More specifically, we manage to bind some interfaces in a way shown in Figure 10. One group of interfaces is attached to the BMv2, while the other is exposed to the outside world to handle incoming and outgoing network traffic.

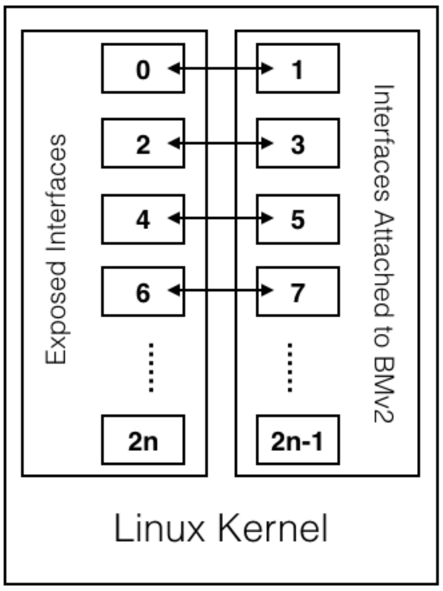

Figure 10: Virtual Interfaces Binding

Once the BMv2 is up and running as a software router on a virtual machine with the specification shown on Table 4, we can initiate the network traffic flow. Basically, as shown in Figure 11, we deploy a packet generator for sending packets and a packet sniffer for traffic analysis. And we chose Scapy [34] as a tool for achieving both tasks, because Scapy is a relatively powerful interactive packet manipulation program, supporting various protocols, and performs very well with many other specific tasks that most other tools cannot handle.

Provider	Google Cloud Platform
Operating System	Ubuntu 16.04 LTS
CPU	Virtual CPU * 2
Memory	7.5 GB

Table 4: Virtual Machine Specification

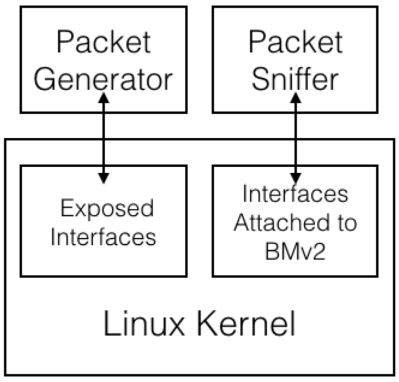

Figure 11: Packets Generator and Sniffer

6.2 Traffic Flows for Flows for Evaluation

We designed a network flow scenario (shown in Table 5) where the software router (BMv2) need to classify and send to 5 different outgoing interfaces accordingly. Obviously, we direct some flows to the same destination because we believe they are usually categorized to a similar network service, e.g. both Flow 1 and Flow 2 belong to web services, and we often have emailing traffic like Flow 4 and Flow 5. Five interfaces of the software router are assigned with the MAC addresses appeared on Table 5. Note that flows are sent from interfaces other than outgoing interfaces for simplicity. Here the MAC addresses of incoming interfaces are not shown explicitly, because packets classification does not consider them in our work. For our evaluation experiment, all flows are generated simultaneously and continuously unless stated otherwise.

Flow No.	Source IP Address	Source Port Number	Protocol	Destination MAC Address
1	10.1.1.1/24	80	TCP	00:00:00:00:00:01
2	10.1.1.1/24	443	TCP	00:00:00:00:00:01
3	10.1.1.2/24	22	TCP	00:00:00:00:00:02
4	172.16.1.1/24	25	TCP	00:00:00:00:00:03
5	172.30.1.254/24	993	TCP	00:00:00:00:00:03
6	192.168.10.1/24	161	UDP	00:00:00:00:00:04
7	192.168.10.2/24	123	UDP	00:00:00:00:00:05

Table 5: Traffic Flows for Evaluation

6.3 Comparison on Classification Algorithms

6.3.1 Evaluation Metrics

Our evaluation on the performance of the P4 software router will be based on CPU and memory usages on virtual machine, showing the amount of computation power required on different scenarios. We calculate the latency of classifying packets in different cases. We define the latency as the timestamp discrepancy between when a packet enters the ingress pipeline and when a packet exits the ingress pipeline, while the egress pipeline is outside the scope of our work. Also, by changing the threshold where a flow is recognized, we observe how this parameter affects the performance based on aforementioned metrics respectively.

We focus on three classification algorithms, i.e. linear approach, decision tree approach and the proposed algorithm in this thesis. We collected data once the traffic flows were initiated, and calculated an average value of a bunch of data within a time frame (e.g. 20 milliseconds). The collection process terminated when 20 time frames passed. Note that CPU and memory usages data are collected through the top program on Linux, which provides a dynamic real-time view of a running system.

6.3.2 Evaluation on CPU usage

Figure 12 shows the virtual machine CPU usage under three different classification algorithms. We can tell that the computation cost under linear approach and decision tree approach are almost the same, while linear approach performs slightly better. A possible explanation is that the decision tree flow structure may have complicated the processing, making it less efficient. The Flow-level approach incurs high CPU usage initially, but perform better in the later stages. Since classifying the first few packets does not have any benefits from computing efficiency while incurring packet-recording overheads, it is expected to have higher CPU usage. However, we can see that the short-cut processing does show some advantages after recognizing traffic flows later on.

While the experiment results shown in Figure 12 was carried out with the flow recognition threshold of 100 packets, Figure 13 and Figure 14 show the results of extended experiment by setting the threshold to 50 and 25 packets, respectively. Since our proposal requires relatively higher computation in the early stage, we expect this cost could be lower if the threshold is set to a smaller value. The results support this assumption. In the early stage (time frame ¡ 6), our proposed algorithm yields slightly better result by setting the threshold to 50 packets. This benefit is more obvious if the threshold is set to 25 packets. On the other hand, with altered threshold, CPU utilization does not show much difference either for linear approach or decision tree approach classification.

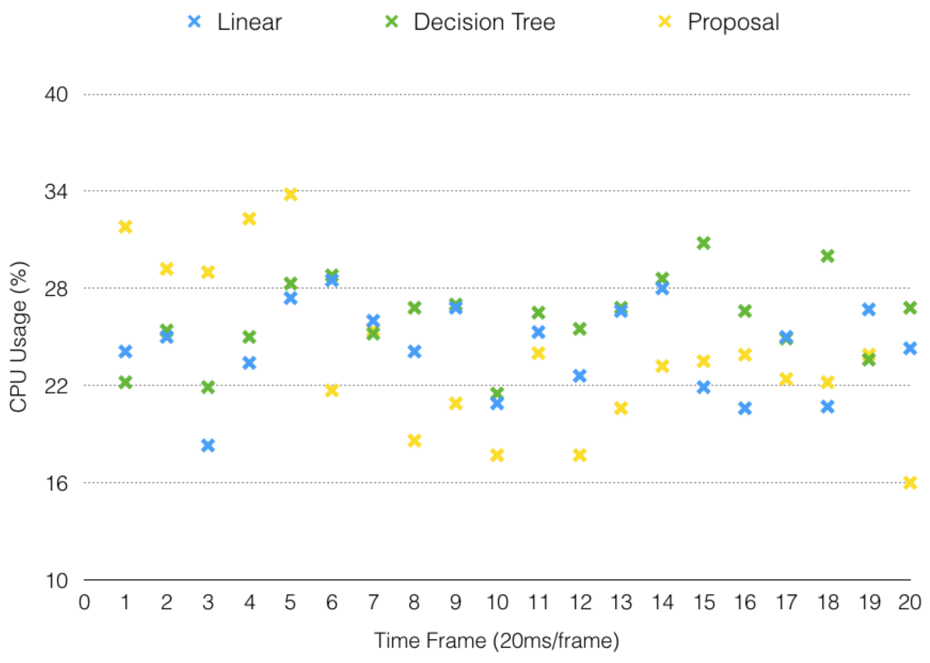

Figure 12: CPU Usage Comparison (threshold = 100 packets)

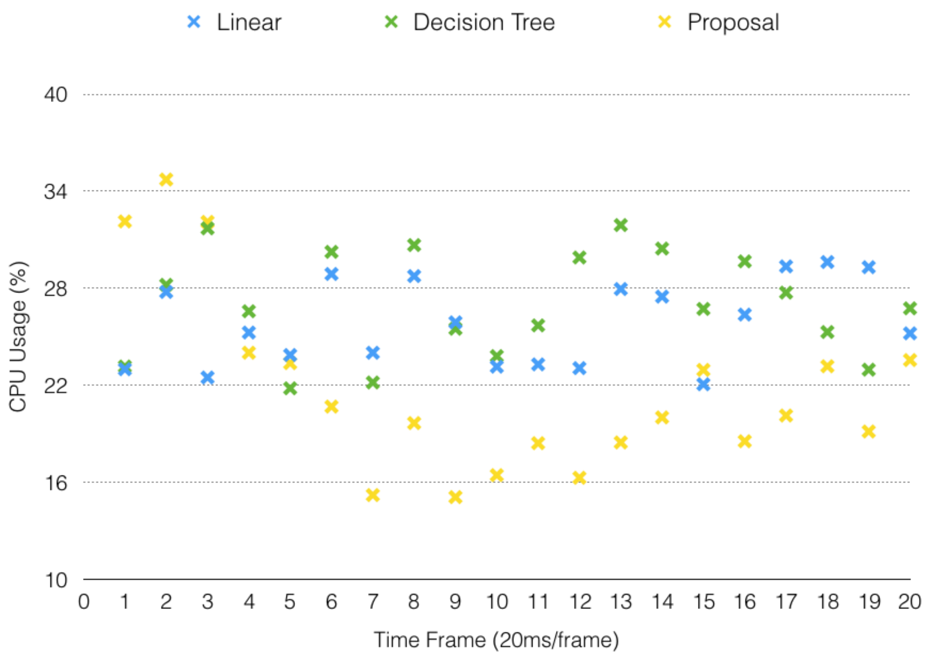

Figure 13: CPU Usage Comparison (threshold = 50 packets)

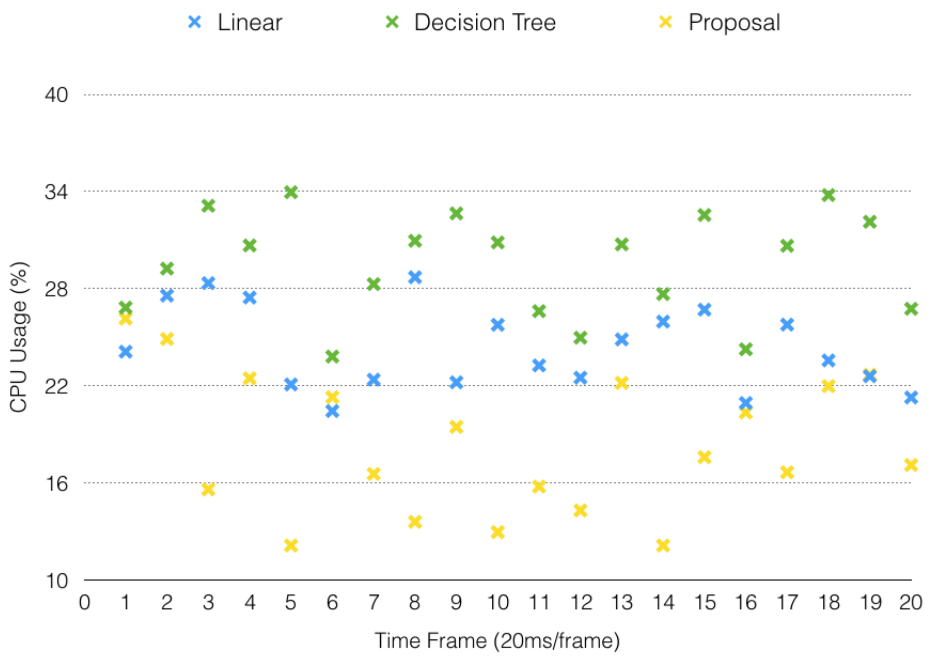

Figure 14: CPU Usage Comparison (threshold = 25 packets)

6.3.3 Evaluation on Memory Usage

Memory cost is another metric we would like to evaluate. Though it is reasonable to argue that data structures in our proposal may have high space cost, we believe it is an acceptable trade off if the actual memory cost does not have huge surge compared with other algorithms.

We conduct experiments by setting the threshold to three different scenarios as depicted in Figure 15, Figure 16 and Figure 17. For each threshold case, we conclude that there are not many differences between the linear approach and the decision tree approach. Nonetheless, it is obvious that the flow-level approach has more memory cost on average. A reason for this result could probably be that stateful memories (i.e. Counters and Registers in this case) do cost non-negligible memory resource.

However, the memory cost of our proposed algorithm stays quite consistent across all threshold cases, which means threshold setting value is not a factor that affects the memory cost directly. We assume refining data structure would be a better way for such improvement.

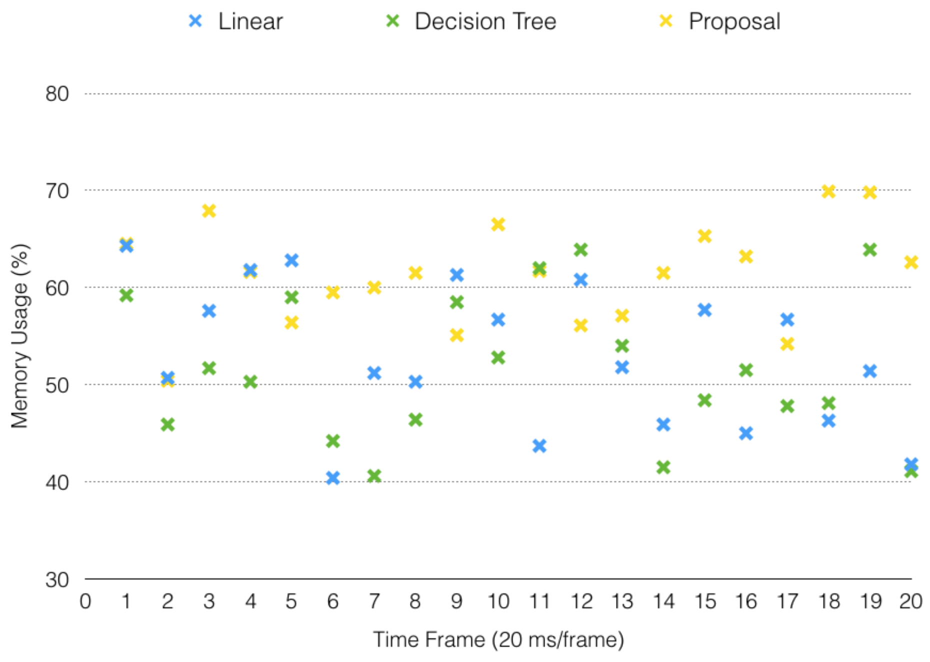

Figure 15: Memory Usage Comparison (threshold = 100 packets)

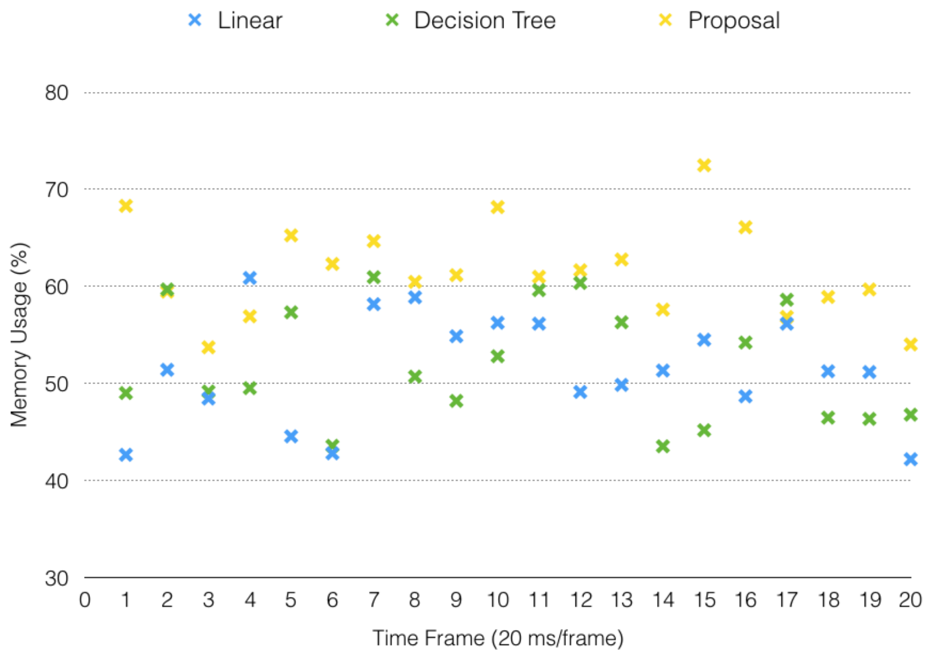

Figure 16: Memory Usage Comparison (threshold = 50 packets)

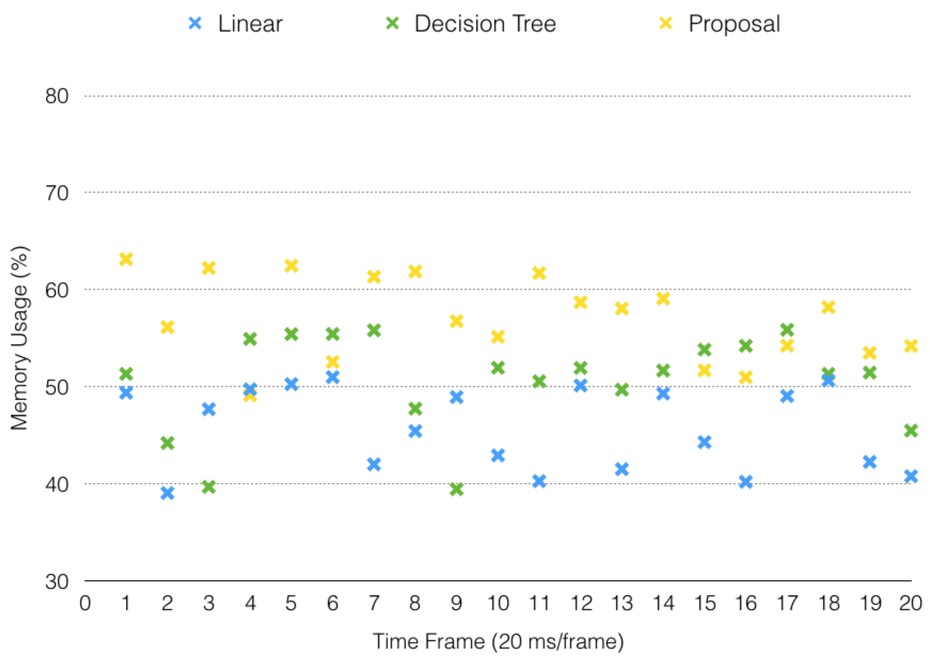

Figure 17: Memory Usage Comparison (threshold = 25 packets)

6.3.4 Evaluation on Latency

Figure 18 depicts the routing latency under these three algorithms. It shows not much difference between the linear approach and the decision tree approach, which indicates that the decision tree approach has almost no gains on improving routing efficiency. And under flow-level approach, it shows higher latency in the initial stage, but the performance gains advantage for the long run.

We assume such high latency from our proposal is a result of packet counting operations before a flow is recognized. Thus, by decreasing the recognition threshold, we believe the period of high latency can be shorten to a more acceptable range. Our extended experimental data shown in Figure 19 and Figure 20 support our expectation.

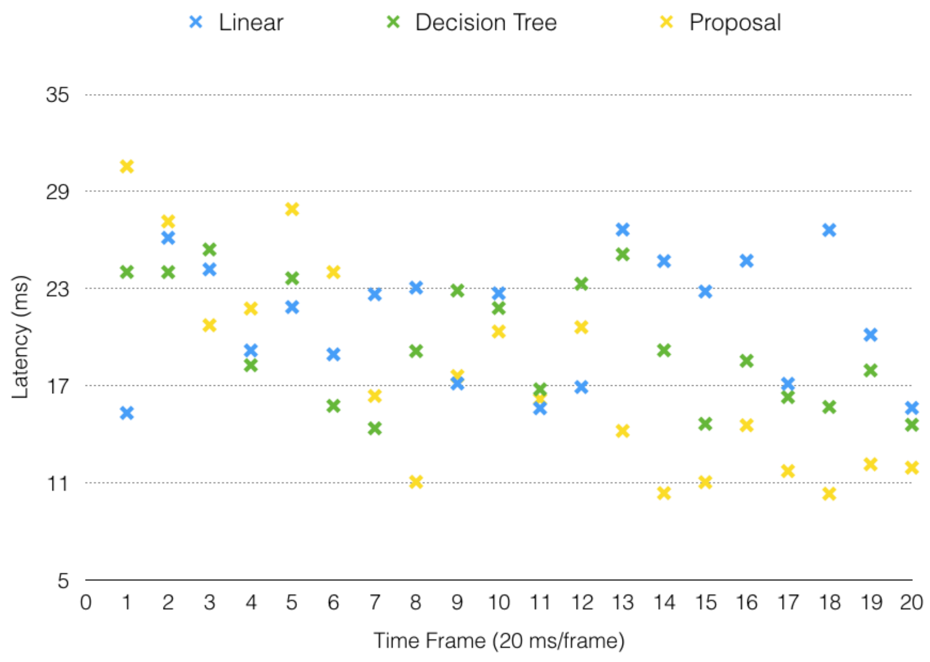

Figure 18: Latency Comparison (threshold = 100 packets)

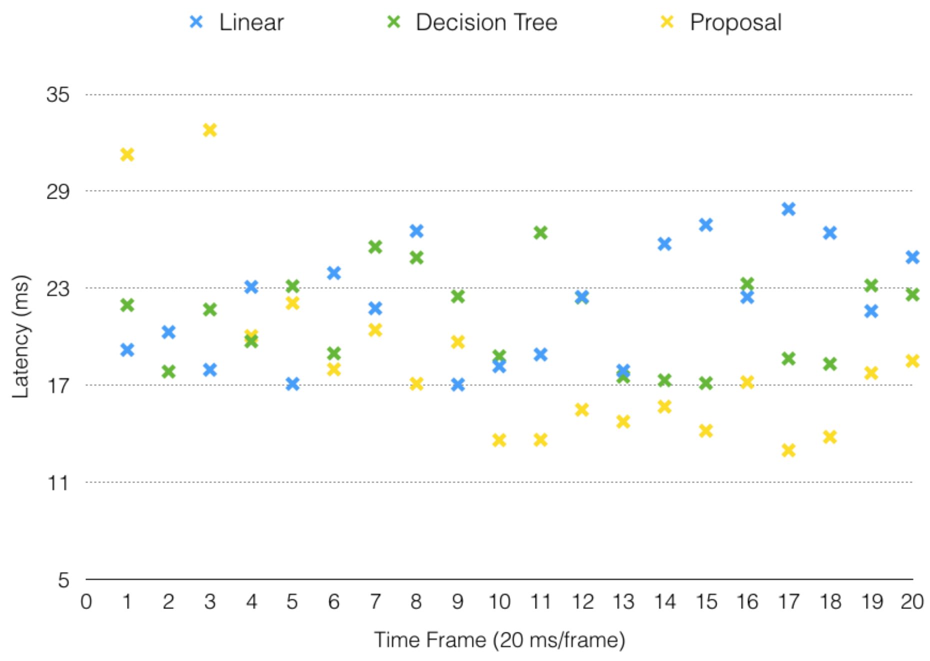

Figure 19: Latency Comparison (threshold = 50 packets)

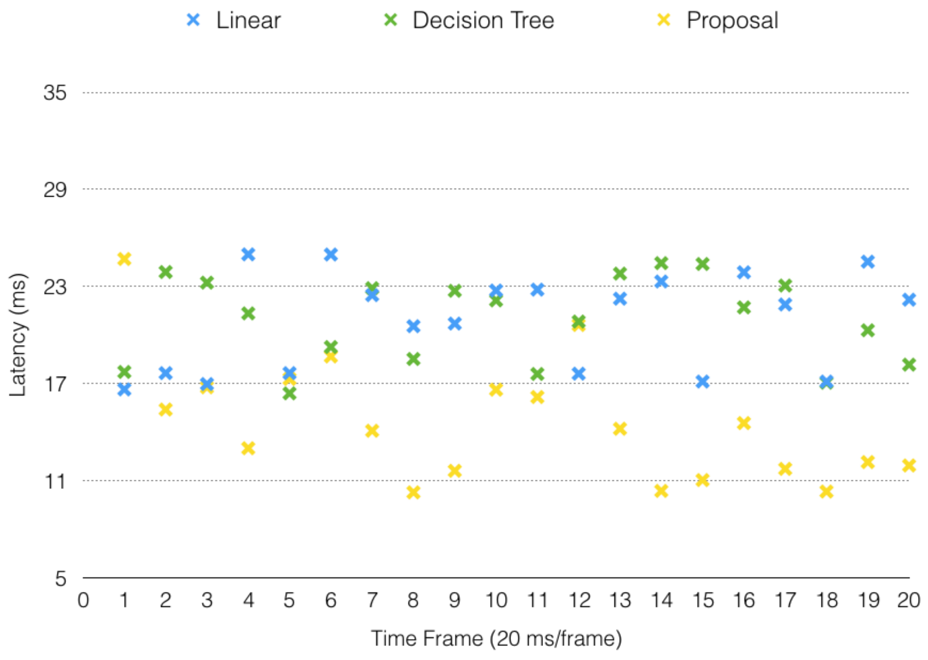

Figure 20: Latency Comparison (threshold = 25 packets)

6.4 Comparison on Implementations of Proposal

The empirical evaluations in section 6.3 are based on the choices that the counting mechanism is implemented as a bloom-filter, using *csum1* and *crc16* hash functions at the same time. In this section, we will evaluate different counting implementation and how the router performance is influenced by hash functions.

6.4.1 Evaluation on Different Counting Implementations

As discussed in section 5.3, the counting structure can be implemented in two ways, i.e. using *direct counter* from the stateful memories or using a bloom filter data structure.

Our evaluation (Figure 21) shows that the difference in performance is negligibly small. Though using *direct counter* in P4 does yield a little advantages in terms of memory and computation cost, considering that *direct counter* will not work for new flows that are not specified in the table, we believe the bloom-filter is more promising for this trade-off.

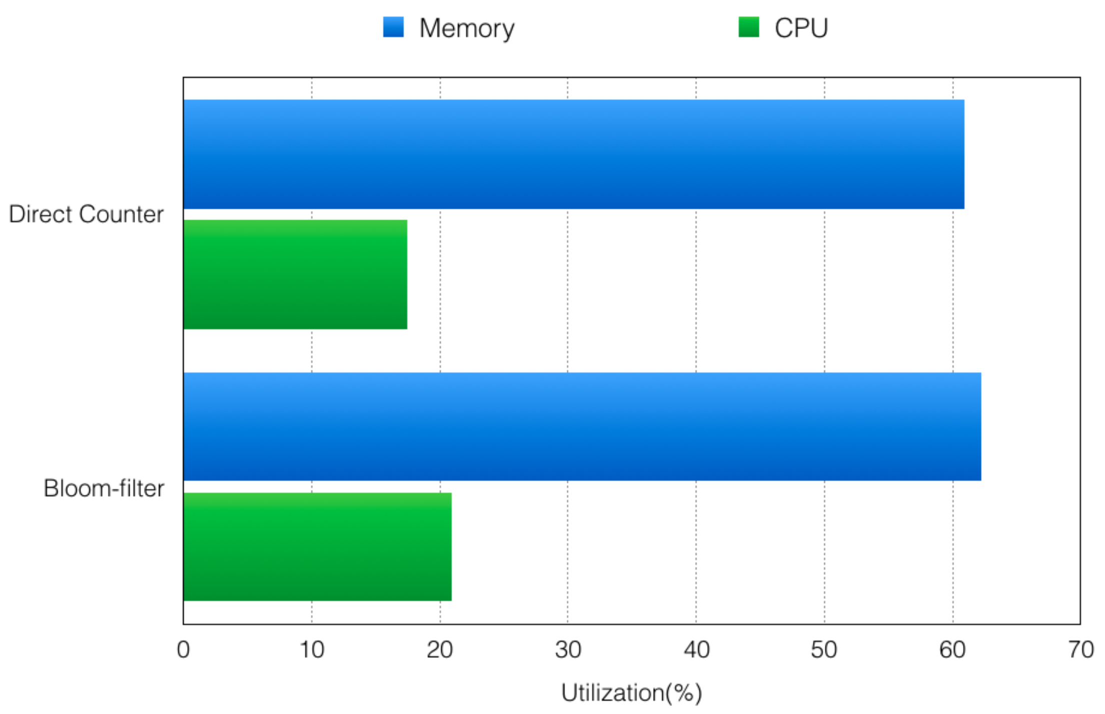

Figure 21: Direct Counter and Bloom-filter Implementation Comparison

6.4.2 Evaluation on Hash Functions

A number of hash functions are originally available in P4 [27]. Especially, *csum16* is normally used for calculating IPv4 header checksum [35], while *crc16* and *crc32* are two cyclic redundancy check (CRC) for error-detecting with different polynomial lengths.

Figure 22 illustrates the CPU utilization on the virtual machine using different hash functions for implementation. It is obvious that *crc32* function costs relatively more computation power. While the combination *csum1+crc16* does need more CPU resource as expected, the increase is not regarded as a considerable amount. Since we need to avoid hash collision, such trade-off is believed to be acceptable.

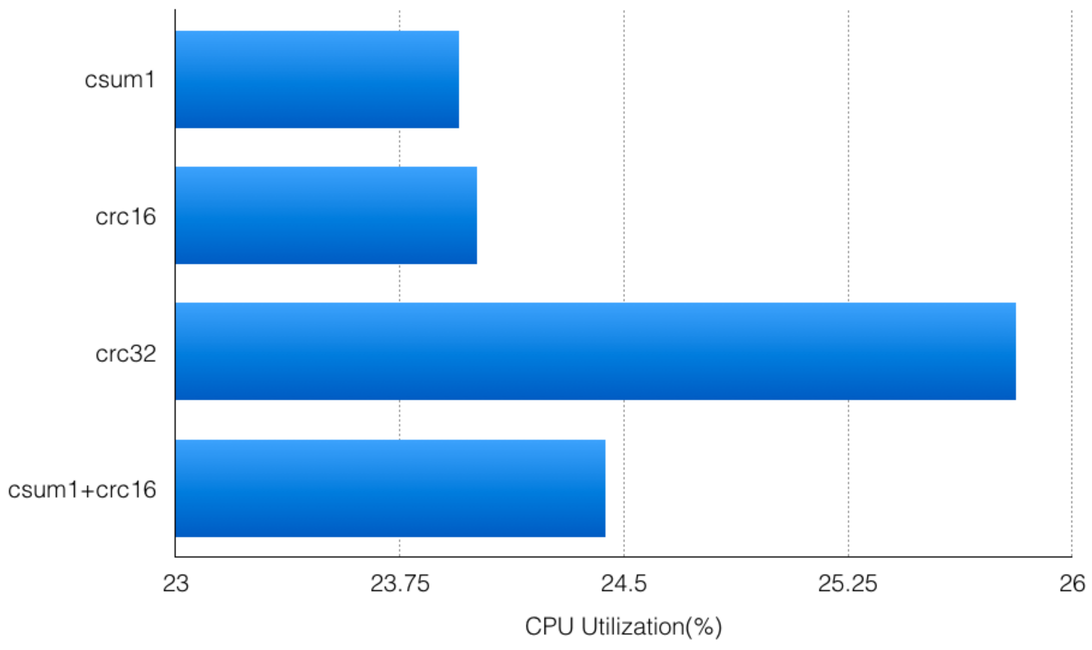

Figure 22: Hash Functions Comparison

6.5 Discussion

Experiments conducted in this thesis show advantages of our proposal in terms of less computation cost and reduced latency. Except for the initial stage, our approach yields roughly 20% better than other approaches, and roughly has 28% less latency on average in our experiment. This improvement is appreciated since modern network traffic is growing exponentially, giving much pressure on the network infrastructure. The comparisons of different counting implementation and hash functions' combinations give some indications for network programmers to evaluate their choices on the basis of their actual needs.

Chapter 7

CONCLUSION

The deployments of software routers compiled in P4 language have the advantages of flexibility and portability. Our research presented in this thesis investigated the various data structure and algorithms for packet classification by the P4 language description.

We conclude from the experimental results that our proposal do have advantages in terms of less computation cost and reduced latency, with acceptable memory cost. We also compared other implementation method for counting mechanism and evaluated other hash functions to access their influence on router performance. Motivated by viewing the packet classification problem from a higher prospective, our approach improves the computation efficiency and forwarding latency of the virtual machine on which the software router is running. While our approach has a the relatively higher memory cost comparing with some previous algorithms, we believe it is an acceptable trade off given that memory resource is getting cheaper over the years.

Since the P4 language is still on its early stage of development, much effort is needed to improve its expressiveness on networking functions. Future studies related to P4's behavior model may shed lights on which component could be further be improved. For example, a customized parser could give more efficiency on parsing particular packets. The utilization of recirculation and clone operation in the egress pipeline could also be used for recursive packet processing.

Bibliography

[1] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, "Requirements for Traffic Engineering Over MPLS," http://www.ietf.org/rfc/rfc2702.txt, accessed on 30 Sep 2016.

[2] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol.37, no. 3, pp.238–275, Sep. 2005.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and others, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol.44, no. 3, pp.87–95, 2014.

[4] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol.17, no. 2, pp.30–32, 2009.

[5] E. E. Haleplidis, E. K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-Defined Networking (SDN): Layers and Architecture Terminology," *Internet Res. Task Force*, pp.1–35, Jan. 2015.

[6] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN," *Queue*, vol.11, no. 12, p.20, 2013.

[7] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan, "No silver bullet: Extending SDN to the data plane," ACM Workshop on Hot Topics in Networks, p.19, 2013.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," ACM SIGCOMM Comput. Commun. Rev., vol.43, no. 4, pp.99–110, 2013.

[9] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: using packets for low latency network programming and visibility," ACM SIGCOMM, pp.3–14, 2014.

[10] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam, "PLUG: Flexible lookup modules for rapid deployment of new protocols in high-speed routers," ACM SIGCOMM Comput. Commun. Rev., vol.39, no. 4, pp.207–218, 2009.

[11] M. Shahbaz and N. Feamster, "The case for an intermediate representation for programmable data planes," *Proc. 1st ACM SIGCOMM Symp. Softw. Defin. Netw. Res. -*

SOSR '15, pp.1–6, 2015.

[12] H. Song, "Protocol-oblivious forwarding," Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13, p.127, Aug. 2013.

[13] H. Song, J. Gong, H. Chen, and J. Dustzadeh, "Unified POF Programming for Diversified SDN Data Plane," *ICNS 2015, pp.106*, Apr. 2014.

[14] P. Benácek, V. Puš, and H. Kubátová, "Automatic Generation of 100 Gbps Packet Parsers from P4," http://www.beba-project.eu/papers/CESNET_p4.pdf, accessed on 5 Jan 2017.

[15] L. Jose, L. Yan, P. Bosshart, D. Daly, G. Varghese, and N. McKeown, "Mapping Match+ Action Tables to Switches," *Open Netw. Summit Res. Track*, 2014.

[16] P. Li and Y. Luo, "P4GPU: Accelerate Packet Processing of a P4 Program with a CPU-GPU Heterogeneous Architecture," *Proc. 2016 Symp. Archit. Netw. Commun. Syst. - ANCS '16*, pp.125–126, 2016.

[17] C. Kim and others, "In-band Network Telemetry (INT)," The P4 Consortium, http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf, accessed on 16 Feb 2016.

[18] H. Xiao, Y. Xu, and V. P. Kafle, "Study of Real-time Network Monitoring and Troubleshooting by Using In-band Network Telemetry," *IEICE Tech. Rep.*, vol.115, no. 484, pp.141–146, Mar. 2016.

[19] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," Hot Interconnects VII, pp.34–41, 1999.

[20] T. Y. C. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," Proceedings of the 19th IEEE Conference on Computer Communications (INFOCOM '00), vol.0, pp.1213–1222, 2000.

[21] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," *Proc. ACM SIGCOMM Conf.*, pp.213–225, 2003.

[22] R. Montoye, "Apparatus for storing' don't care' in a content addressable memory cell," Hal Computer Systems, 1994.

[23] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *ACM SIGCOMM Comput. Commun. Rev.*, vol.28, no. 4, pp.191–202, 1998.

[24] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router plugins: a software architecture for next generation routers," *ACM SIGCOMM Comput. Commun. Rev.*, vol.28, no. 4, pp.229–240, 1998.

[25] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: is there an alternative to CAMs?," INFOCOM 2003. Twenty-Second Annual Joint Conference of the

IEEE Computer and Communications. IEEE Societies, vol.1, pp.53–63 vol.1, 2003.

[26] A. Feldman and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. 19th IEEE Conf. Comput. Commun. (INFOCOM '00)*, vol.3, pp.1193–1202, 2000.

[27] The P4 Language Consortium, "The P4 Language Specification," *IEEE Trans. Mob. Comput.*, pp.1–90, 2015.

[28] "P4 Language Consortium," http://p4.org/, accessed on 1 Oct 2016.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow," *ACM SIGCOMM Comput. Commun. Rev.*, vol.38, no. 2, p.69, March 2008.

[30] H. Müller-Merbach, "Heuristics: Intelligent search strategies for computer problem solving," European Journal of Operational Research, vol.21, no. 2, Addison-Wesley Pub. Co., Inc.,Reading, MA, pp.278–279, 1985.

[31] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.*, vol.1, no. 4, pp.485–509, Jan. 2004.

[32] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter," *ACM SIGCOMM Comput. Commun. Rev.*, vol.35, no. 4, p.181, Oct. 2005.

[33] "Apache Thrift," http://thrift.apache.org/, accessed on 21 Jan 2017.

[34] "Scapy," http://www.secdev.org/projects/scapy/, accessed on 11 Oct 2016.

[35] S. Deering and R. Hinden, "Internet Protocol," Internet Requests for Comments, RFC18883, 1995.