A Dynamic Programming Based Tuple Combination for High Speed Packet Classification

Anonymous Author(s)

ABSTRACT

With the development of the network, the OpenFlow packet classification algorithm is facing higher comprehensive performance challenges. It needs to simultaneously satisfy two requirements: high lookup speed and high update speed. Although the packet classification problem has been studied for many years, no published algorithms satisfy these two requirements. Some algorithms like TSS, TupleMerge and PartitionSort are tradeoffs between limited lookup and update speeds. Some decision tree algorithms like ByteCuts have high lookup speeds, but are hard to support updates, which makes them unable to implement in OpenFlow. In this paper, we propose a Dynamic programming based Tuple Combination (DTC) that achieves high lookup and update speeds. First, DTC allows the rules with similar mask lengths to be combined into one tuple. Second, DTC estimates the lookup time for each tuple. Third, DTC uses dynamic programming to find an appropriate tuple combination scheme to reduce the total lookup time. Finally, DTC uses two optimization methods to improve the lookup speed. The experimental results show that the lookup speed of DTC is 31.5 times of TSS, 5.2 times of TupleMerge, 4.8 times of PartitionSort and the update speed is 2.8 times of TSS, 4.8 times of TupleMerge, 5.3 times of PartitionSort.

1 INTRODUCTION

Traditionally, packet classification for Internet is integrated with hardware devices. Although fast, it is difficult to perform flexible and scalable packet forwarding, which is critical with the quick development of new technologies on the Internet. Software Defined Network (SDN) [11] has been proposed to replace hardware, which not only reduces the hardware costs but also makes the entire network more flexible and easier to manage. The standard protocol in SDN is OpenFlow [17]. Although packet classification is one of the core components, it remains the bottleneck for packet forwarding.

The packet classification mainly involves rule matching, where a rule is generally described with d fields $f_1, f_2, ..., f_d$. Each field f_i in a rule corresponds to a range l_i, r_i. In the classic packet classification problem, the rule has five fields, containing the source IP address, the destination IP address, the source port, the destination port and the protocol. The packet header contains d numbers $(p_1, p_2, ..., p_d)$, which correspond to d fields. A packet matches a rule in a field f_i if $p_i \in l_i, r_i$, and matches a rule if it matches all fields. If a packet can match multiple rules, the rule with the highest priority will be selected.

The main challenges faced by packet classification are the lookup speed and rule update speed. Even though many solutions have been proposed, none has simultaneously achieved high speed for both lookup and update. Algorithms such as Tuple Space Search (TSS) [20], TupleMerge [5] and PartitionSort [24] support dynamic updates and attempt to trade off between lookup speed and update speed, but neither speed is very high. Some decision tree methods such as ByteCuts [6] and SmartSplit [10] only focus on high lookup speed, but can not support dynamic updates to implement in OpenFlow. In order to simultaneously achieve high lookup speed and high update speed, we propose a Dynamic programming based Tuple Combination (DTC) for packet classification.

Existing algorithms that support dynamic updates divide rules into multiple groups, with each group named a tuple to contain rules of the same feature. Different algorithms form the tuple with different features and data structures. For example, the tuple in TSS stores the rules with the same source IP mask length and destination IP mask length, and uses cuckoo hash table to support updates. The tuple in TupleMerge can contain the rules with different source IP mask length and destination IP mask length, and omits the bits of the rules to store them into a cuckoo hash table. The tuple in PartitionSort stores the sortable rules and applies binary search tree to achieve logarithmic lookup and update complexity.

To achieve high speed for both lookup and update, we construct tuples with new feature and new data structure in DTC. The lookup operation requires traversing the tuples to find the matching rule with the highest priority, thus its speed depends on the number of tuples and the time to search through the set of rules inside a tuple. Accordingly, we propose *a Mask Length Reduction (MLR) method* to reduce the number of tuples, and a priority-based data structure to further reduce the

lookup time inside each tuple with four layers: tuples, buckets, slots and rules.

The MLR method allows a tuple to store rules with different source and destination IP mask lengths, and restricts a rule to be inserted into one tuple only. The reduction of the number of tuples with shorter mask length, however, increases the length of the rule chains thus lookup time inside a tuple. There is a need to trade off between the number of tuples and the rule lengths inside tuples, and also take into account the distribution of rules inside different tuples. To properly organize rules into tuples, we propose a dynamic programming method to reduce the average lookup time estimated based on the average number of tuples to access and the number of slots and rules to lookup inside each tuple.

Besides source and destination IP addresses considered conventionally for lookup, to further enhance the lookup speed, we exploit the use of the additional three fields on the IP headers in DTC data structure: the source port, the destination port and the protocol number. We propose the search with *the port hash table* and *the protocol division*.

The update operation for TSS only occurs in a specific tuple with the time increasing with the length of the rule chain, while the update operations for TupleMerge and PartitionSort need to traverse the tuples. For DTC, the MLR method reduces the number of tuples available and can map an update operation to a specific tuple. Inside a tuple, it further exploits the priority data structure and the port hash table to improve the update speed.

Our experimental results demonstrate that DTC can simultaneously achieve high lookup speed and high update speed. The lookup speed of DTC is 31.5 times that of TSS, 5.2 times that of TupleMerge and 4.8 times that of PartitionSort. The look up speed of DTC can compete with the state-of-art decision tree method, ByteCuts, while the later can not support the dynamic updates. The update speed of DTC is 2.8 times that of TSS, 4.8 times that of TupleMerge and 5.3 times that of PartitionSort.

The remaining of the paper is organized as follows. We first review related work in Section II. In Section III we introduce the Mask Length Reduction method and the DTC priority data structure. We then estimate the lookup time for each tuple and use dynamic programming to select a set of appropriate tuples in Section IV. In Section V, we propose two optimization methods: the port hash table and the protocol division. The experimental results are shown in Section VI. Finally, we conclude the work in Section VII.

2 RELATED WORK

We classify the existing representative packet classification algorithms into three categories: The first category is the tuple-based methods that support dynamic updates, containing the classic tuple-based method, the merged tuple method and the binary search tree method. The second category is the decision tree methods that only focus on high lookup speed, but are hard to updates. The third category is the the hardware methods that use additional hardware to support lookup. Our proposed DTC falls into the first category because of the high update speed, and has the same high lookup speed as the state-of-art decision tree method in the second category.

The classic tuple-based method: The classic tuple-based method is Tuple Space Search (TSS) [20]. It divides rules into multiple tuples according to the prefix lengths of two fields, the source IP address and the destination IP address. A 32 bit address can form up to 33*33 tuples. Each tuple contains a cuckoo hash table and uses the chain structure to store the rules with all the five fields. ***For the same prefix, it can have a few rules. Why not including IP address?*** The update operation in TSS is mapped to a specific tuple, ***what do you mean "occurs in the corresponding tuple"? How to find a tuple for the update?*** which leads to fast update speed. However, TSS has a slow lookup speed, because it revrequires traversing each tuple and to look for the rule with the highest priority.

The merged tuple method: TupleMerge [5] improves upon TSS by omitting bits of the rule. The mask length of a tuple is decided by the first inserted rule, and it can store the rules with a longer mask length until the rule chain is longer than a threshold k. If a rule can not be inserted into any current tuple, it will build a new tuple. In this way, TupleMerge reduces the number of tuples to improve its lookup speed. However, its lookup speed is slower than SmartSplit, and its update operation revrequires traversing the tuples rather than being mapped directly to a specific tuple, so its update speed is slower than TSS.

The binary search tree method: PartitionSort [24] classifies a packet based on binary search tree. It splits the rules into multiple tuples, each containing sortable rules. The orders of the five fields in these tuples can be different. The data structure in each tuple is optimized by a binary search tree, and the complexity of the lookup and update operation is the logarithm of the rules. ***what is d and n? Explain*** Most rules can be stored in the first few tuples. As a result, PartitionSort achieves fast lookup speed and fast update speed. However, its

lookup speed is slower than SmartSplit, and its update speed is slower than TSS.

The decision tree methods: Some decision tree methods such as HiCuts [9], HyperCuts [8], HyperSplit [19], Efficuts [22], SmartSplit [10], BitCuts [15], CutSplit [13] and ByteCuts [6] use one or more decision trees to build data structure. Each node in the decision tree represents a range of five fields and stores the rules that overlap with the node ranges. The node splits the range into multiple intervals to form children nodes and allocates rules to the children nodes. The rule that overlaps with multiple nodes will be copied multiple times. This could lead to the explosion of memory consumption in decision tree methods. The state-of-art decision tree method, ByteCuts, uses byte extraction to cut the rules, which achieves the highest lookup speed and small memory cost. However, the splitting scheme of these methods depends on the rule set at the building time. If the rules are constantly updated, the previous splitting scheme could be inappropriate, which can further affect the lookup speed and the memory cost. Therefore, these decision tree methods are not suitable for high speed updates. ***(ZCY) I simplify this part and add ByteCuts.***

The hardware methods: Ternary content addressable memory (TCAM) is a hardware device. It is mainly used to quickly find ACLs, routes and other entries. TCAM has been used for packet classification [1, 3, 12, 14, 16, 18] to improve the lookup speed, but it has small memory size and can not store a large number of rules. In addition, TCAM cannot well support the dynamic updates. Some FPGA methods [2, 7] and GPU methods [4, 23] have the same problems. Finally, hardware methods add the costs of hardware and energy. Therefore, these algorithms are difficult to apply to the SDN environment where the rules may need to be flexibly updated.

3 REDUCED TUPLE SEARCH

In a packet classification problem, rules can be classified into multiple groups and each group is called as a tuple. The lookup process needs to traverse tuples to find the matching rule with the highest priority. Therefore, we need to reduce the number of tuples and the lookup time in each tuple to achieve high lookup speed. In this section, we propose a Mask Length Reduction (MLR) method to reduce the number of tuples, and design a DTC data structure with three priority-based enhancements to reduce the lookup time in each tuple.

3.1 Mask Length Reduction

The rules can be divided into multiple tuples by different mask lengths of the source and destination IP addresses. As shown in Figure 1, the mask lengths of Rule1 and Rule2 are (24,24), the mask length of Rule3 is (16,24), and the mask length of Rule4 is (16,16). Therefore, the rules are divided into three tuples. The lookup operation requires traversing the tuples to find the rule with the highest priority. Each tuple contains a hash table that uses two fields to match the rules, and then applies the chained search to check the five fields. ***Why directly to the remaining three fields? You could also actually search the match of the actual source and destination of rules, right?*** For example, the Packet2 requires traversing three tuples with three ***why 3 not 2?*** hash searches and a rule check to match Rule4 in Tuple3.

To reduce the number of tuples for higher lookup speed, the Mask Length Reduction (MLR) takes two strategies: reducing the mask length of the rule to a specific value, and merging rules with similar mask lengths into one tuple. In Figure 1, MLR reduces the rules with the mask lengths in (16∼24, 16∼24) to (16,16). So Rule1-Rule4 become Rule1'-Rule4', with the mask lengths trimmed to (16,16), and all four rules are placed in a tuple. The Packet2 only performs a hash search and a rule check to match Rule4 in Tuple1'.

Different from TSS and TupleMerge, a tuple in DTC contains the rules with a range of mask lengths and a rule is inserted into only one tuple. In contrast, containing only rules with the same mask length, TSS generates a large number of tuples. As a rule in TupleMerge can be inserted into multiple tuples, it is difficult to manage the tuples, which increases the update speed.

Reducing the number of tuples on the one hand increases the lookup speed, but on the other hand also leads more rules to fall into the same tuple and increases the lookup speed inside the tuple. ***How to find a tuple? If it is through hashing, not search, will higher number of tuples leads to more searching time? What is slot? You haven't introduced slot yet, right? You kept saying slot here.(ZCY) I add the tuple traversing in the first paragraph*** For example, Rule1-Rule3 are changed to Rule1'-Rule3' and have the same source IP and destination IP. Therefore, they are indexed to be stored in the same tuple in the hash table and form a chain.

If a tuple containing too much rules, the lengths of the rule chain will be long, which will affects the lookup speed. Therefore, it is important to consider of both the number of tuples and the lengths of the rule chains. ***Do you need to look for step size in this work? You are forming tuples, but didn't do it through the finding of "step size", from what you presented. So this***

paragraph does not mactch your scheme, although you are looking for the best tuple formulation.

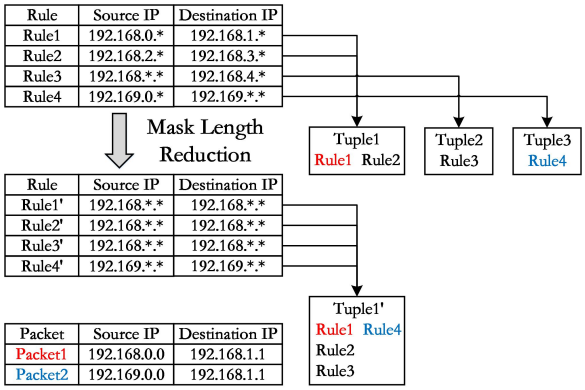


Figure 1: Mask Length Reduction.

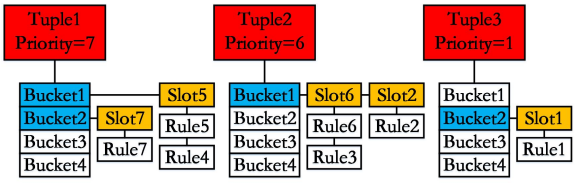


Figure 2: Basic DTC data structure.

3.2 basic data structure of DTC

The basic data structure of DTC consists of four parts: tuples, buckets, slots and rules, as shown in Figure 2.

Tuples: Each tuple contains a hash table which consists of multiple buckets. A tuple only stores the rules with the same mask length of source IP and destination IP address after MLR.

Slots: Each slot applies a chain structure to sotre the rules with the same source and destination IP after MLR.

Buckets: Each bucket applies a chain structure to store slots. A slot uses its source and destination IP to form a *key*. Then, a *key* is hashed to be a *hash_key*, which is 32-bits. Suppose the number of buckets in a tuple is n, and the index of the bucket that stores this slot is $hash\_key\%n$. Different slot has different key, however, they can be indexed to the same bucket.

All tuples, rules and slots in DTC data structure are sorted by their priorities. As shown in Figure 2, suppose the rule set contains seven rules Rule1-Rule7 and the priority of Rule$i$ is i(*e.g.*,Rule7 has the highest priority 7). The priority of each tuple is determined by the highest priority rule in this tuple, and all tuples are sorted according to their priorities from high to low. The priority of each slot is determined by the highest priority rule in this slot, and the slots in each bucket are also sorted according to their priorities from high to low. Each rule has a priority itself, and the rules in each slot are sorted according to their priorities from high to low.

3.3 DTC lookup and update

The complete lookup process for a packet in the DTC data structure consists of three steps. First, the packet searches in all tuples. Second, in each tuple, we use the hash function to quickly find the matching slot with the matching source IP and destination IP after MLR. Third, in each matching slot, we check all the rules to find the matching rule with the highest priority. The last matching rule of this packet is the rule with the highest priority among the matching rules in all tuples.

Due to the DTC priority data structure, the lookup process contains three priority optimization methods. The first is the tuple priority optimization. If the priority of the current matching rule is higher than or equal to the current tuple, we can skip the remaining tuples and return this matching rule. The second is the slot priority optimization. If the priority of the current matching rule is higher than or equal to the current slot, we can skip the remaining slots and then continue the lookup process in the next tuple. The third is the rule priority optimization. If the priority of the current matching rule is higher than or equal to the current rule, we can skip the remaining rules and slots, and then continue the lookup process in the next tuple. For example, suppose Packet5 can match both Rule5 and Rule2. In Tuple1, when Packet5 matches Rule5, the priority of the current matching rule is 5, then we skip Rule4 and continue the lookup process in the next tuple by the rule priority optimization. In Tuple2, we check Slot6 and skip Slot2 by the slot priority optimization. In Tuple3, the priority of the matching rule is higher than Tuple3, so we stop the lookup process and return Rule5 by the tuple priority optimization.

The pseudo code for the lookup process is shown in Algorithm 1. 1Line1-2 initialize the matching rule r to $NULL$ and the priority of the matching pri to -1. 2Line3-34 are the first step that traverses all tuples. Line5-7 are the tuple priority optimization. Line8-9 calculate the source IP sip and the destination IP dip after MLR. Line10-12 use the hash function of sip and dip to find the indexed bucket in the hash table. 3Line13-33 are the second step that traverses the slot chain to find the matching slot. Line14-16 are the slot priority optimization. 4Line19-29 are the third step that traverses the rule chain to find the matching rule. Line26-28 are the

rule priority optimization. 5Finally, the lookup process returns the matching rule r in line35.

Algorithm 1: The lookup process

```
Input: the array of tuples tuples, the number of
       tuples tuples_num, the packet packet;
Output: the matching rule r;
1  r = NULL;
2  pri = −1;
3  for i = 0; i < tuples_num; i++ do
4      tuple = tuplesi;
5      if pri ≥ tuple.pri then
6          break;
7      end
8      sip = packet.sip&tuple.sip_mask;
9      dip = packet.dip&tuple.dip_mask;
10     hash = HashFunctionsip, dip;
11     hashtable = tuple.hashtable;
12     slot =
         hashtable.bucketshash&hashtable.mask;
13     while slot ≠ NULL do
14         if pri ≥ slot.pri then
15             break;
16         end
17         if sip == slot.sip and dip == slot.dip then
18             rule = slot.rule;
19             while True do
20                 if Matchrule, packet == True then
21                     pri = rule.pri;
22                     r = rule;
23                     break;
24                 end
25                 rule = rule.next;
26                 if rule == NULL or pri ≥ rule.pri
                     then
27                     break;
28                 end
29             end
30             break;
31         end
32         slot = slot.next;
33     end
34 end
35 return r;
```

The operation of inserting or deleting a rule is very simple. First, we use the mask lengths of the source IP and the destination IP of this rule to find the corresponding tuple. Then, we perform the insert or delete operation in the hash table of this tuple. If the insert or delete operation changes the priority of the slot or tuple, we adjust the order of the slots or tuples to ensure the DTC priority structure.

4 TUPLE COMBINATION

To achieve high lookup speed, the DTC should simultaneously reduces the number of tuple and the lookup time in each tuple. First, we propose a calculation and estimation method to estimate the lookup time in each tuple. Second, we propose a dynamic programming method to find a set of tuples covering all kinds of mask length to reduce the total lookup time, which can be considered as a tuple combination problem.

4.1 Lookup time Estimation

The lookup time can be divided into four parts: the hash search time of the tuple, the check time of the slot, the verification time of the rule and the constant time of the lookup. Each part is approximated as the access number N multiplied by the average time cost T for each access. Therefore, we define the total lookup time T_{lookup} as:

$T_{lookup} = N_{tuple} * T_{tuple} + N_{slot} * T_{slot} + N_{rule} * T_{rule} + N_{lookup} * T_{constant}$

The average time cost T_{tuple}, T_{slot}, T_{rule} and $T_{constant}$ can be calculated by liner programming. And N_{lookup} is obviously equal to the size of the packet set. If we can calculate the access numbers N_{tuple}, N_{slot} and N_{rule}, then we can use this formula to estimate the total lookup time T_{lookup}. Unfortunately, the access numbers in each packet lookup process are difficult to be calculated, because the step sizes of MLR are unknown and the priority optimization methods are complex. Therefore, we will calculate the access numbers in each tuple, and then estimate their lookup time. Finally, we combine the appropriate tuples into a complete data structure to reduce the total lookup time T_{lookup}.

As shown in Figure 2, suppose the rule set contains seven rules, Rule1-Rule7. The priority of Rule$i$ is i, thus Rule7 has the highest priority 7. And the packet set contains seven packet Packet1-Packet7 which respectively match Rule1-Rule7. Next we will calculate the access numbers N_{tuple}, N_{slot} and N_{rule} for each tuple.

4.1.1 N_{tuple}. The access number of a tuple is related to the priority of this tuple. Because Packet$i$ matches Rule$i$, Packet$i$ must access the tuple with the priority higher than or equal to i. In other words, Tuple$i$ is accessed by the packet whose matching rule has a priority less than or equal to the priority of Tuple$i$. The access numbers of each tuple are shown in Table 1. For example, because each rule is matched by a packet, the access

number of Tuple2 is equal to the number of rules whose priority is less than or equal to 6.

Table 1: The access numbers of tuples

	Rule1	Rule2	Rule3	Rule4	Rule5	Rule6	Rule7	Access number
Tuple1	√	√	√	√	√	√	√	7
Tuple2	√	√	√	√	√	√		6
Tuple3	√							1

4.1.2 N_{slot}. As shown in Figure 3. The access number of a slot is divided into two parts: the matching access and the collision access. In the lookup process, a packet needs to traverse the slot chain to find the matching slot in a bucket. If the packet matches a slot, it is a matching access. And if the packet does not match a slot, it is a collision access because the hash table has hash collision. In our experiment, the matching access occupies 20% and the collision access occupies 80%. This probability distribution is related to the size of the hash table, In our implementation, the number of buckets is a power of 2 and the number of slots is up to 85% of the number of buckets. Suppose the priority of Packet$i$ is equal to the priority of the matching rule(*e.g.*,the priority of Packet7 is 7), then we will respectively calculate the number of matching access and the number of collision access.

First, for the matching access part, we divide it into the low priority matching access and the high priority matching access. For example, if Slot6 is matched by Packet5 and the priority of Packet5 (pri = 5) is less than or equal to the priority of Slot6 (pri = 6), we define this matching access as a low priority matching access for Slot6. And if Slot5 is matched by Packet6 and the priority of Packet6 (pri = 6) is higher than the priority of Slot5 (pri = 5), we define this matching access as a high priority matching access for Slot5. Our experimental results show that the number of the high priority matching access only occupies 0.03% of the number of slot access. Therefore, we can ignore the high priority matching access and only calculate the low priority matching access. For example, the priority of Slot6 is 6 and the low priority matching access only occurs in Packet1-Packet6 whose priority is less than or equal to 6. If Packet1, Packet3, Packet5 and Packet6 can match Slot6, the matching access number of Slot6 is 4. Since the priorities of Packet1-Packet6 are less than or equal to the priority of Slot6 and the priority of Slot6 is less than or equal to the priority of Tuple2, Packet1-6 will always look up in Tuple2 regardless of the order of the tuples. This feature makes the number of the low priority matching access independent of the order of the tuples.

Second, for the collision access part, we divide it into the low priority collision access and the high priority collision access. For example, if Slot6 is accessed but not matched by Packet5, and the priority of Packet5 (pri = 5) is less than or equal to the priority of Slot6 (pri = 6), we define this collision access as a low priority collision access for Slot6. And if Slot5 is accessed but not matched by Packet6, and the priority of Packet6 (pri = 6) is higher than the priority of Slot5 (pri = 5), we call this matching access match as a high priority collision access for Slot5. Our experimental results show that the number of the high priority collision access only occupies 6.6% in the number of slot access. To keep the number of the collision matching access is always constant regardless of the order of the tuples, we ignore the high priority collision access. For the number of the low priority collision access, we use the probability theory to estimate it. For example, Slot7 can be accessed by Packet1-Packet6 and there are four buckets in Tuple1, so the collision access of Slot7 is $14 * 6 = 1.5$. In our experimental results, the error of the probability estimation is only 3.6%.

4.1.3 N_{rule}. The access number of rule is divided into two parts: the low priority rule access and the high priority rule access. For example, if Rule6 is accessed by Packet5 and the priority of Packet5 (pri = 5) is less than or equal to the priority of Rule6 (pri = 6), we define this rule access as a low priority rule access for Rule6. And if Rule5 is accessed by Packet6 and the priority of Packet6 (pri = 6) is higher than the priority of Rule5 (pri = 5), we define this rule access as a high priority rule access for Rule5. Our experimental results show that the number of the high priority rule access only occupies 0.9% in the number of the rule access. Therefore, we can ignore the high priority rule access and only calculate the number of the low priority rule access. For example, if Packet1, Packet3, Packet5 and Packet6 can match Slot6, Rule3 will be accessed by Packet1 and Packet3 whose priority is less than or equal to the priority of Rule3 (pri = 3). Therefore, the number of the rule access of Rule3 is 2. Furthermore, the number of the low priority rule access is also independent of the order of the tuples.

The distribution for the access numbers of the tuples, slots and rules are shown in Figure 3. Some access calculations are accurate, some are estimated by probability and some are ignored. In our experimental results, the average estimate error of the lookup time for a tuple is only 1.3%. And the average calculate time for a 10K rule set is just 0.29 seconds, which is shown in Section VI. Because the lookup time for a tuple is independent of the order of the tuples, we can combine the appropriate tuples into a complete data structure to reduce the total lookup time.

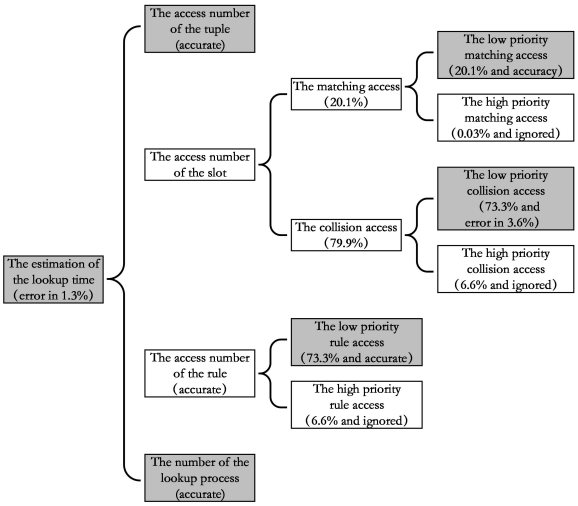


Figure 3: Calculate the number of lookup access.

4.2 Dynamic Programming

we define $T_{x_1y_1x_2y_2}$ is the lookup time of a tuple that the mask length range of source IP is $[x_1, x_2]$ and the mask length range of destination IP is $[y_1, y_2]$. Our goal is to find a set of tuples and combine them into a complete data structure. These tuples should contain all combination mask lengths of source IP and destination IP. And the sum of the lookup time of these tuples should be as small as possible.

It is obvious that we can use a integer linear programming (ILP) method to describe the tuple combination problem. In Figure 4, $X_{x_1y_1x_2y_2}$ indicates whether the Tuple$_{x_1y_1x_2y_2}$ is in the set of selected tuples. The objective function is to minimize the sum of the lookup time of selected tuples. The first constraint is each combination mask length of source IP and destination IP$[i, j]$ should be included in a selected tuple. The second constraint is $X_{x_1y_1x_2y_2}$ should be 0 or 1. 0 means Tuple$_{x_1y_1x_2y_2}$ is not selected, and 1 means Tuple$_{x_1y_1x_2y_2}$ is selected. This ILP problem looks simple, however, if we expand this ILP,it contains about 33^4 variables and constraints. Therefore, this ILP problem is too complicated to solve.

$$\textbf{Min S} = \sum_{x_1=0}^{32}\sum_{y_1=0}^{32}\sum_{x_2=x_1}^{32}\sum_{y_2=y_1}^{32} T_{x_1y_1x_2y_2} X_{x_1y_1x_2y_2}$$

$$\textbf{s.t.}$$

$$\sum_{x_1=0}^{i}\sum_{y_1=0}^{j}\sum_{x_2=i}^{32}\sum_{y_2=j}^{32} X_{x_1y_1x_2y_2} = 1, i \in [0, 32], j \in [0, 32]$$

$$X_{x_1y_1x_2y_2} \leq 1, x_1 \in [0, 32], y_1 \in [0, 32], x_2 \in [x_1, 32], y_2 \in [y_1, 32]$$

Figure 4: The linear programming of combining tuples.

To solve the tuple combination problem, we propose a dynamic programming method to reduce the computational complexity. As shown in Figure 5, the x coordinate is the mask length of the source IP and the y coordinate is the mask length of the destination IP. The number k in the coordinate system $[x, y]$ indicates that there are k rules that the mask length of the source IP is x and the mask length of the destination IP is y. Then, we define $S_{x_1y_1x_2y_2}$ is the sum of the lookup time of tuples that the mask length range of source IP is $[x_1, x_2]$ and the mask length range of destination IP is $[y_1, y_2]$. A S is a same colour matrix in a large matrix. To make $S_{x_1y_1x_2y_2}$ as small as possible, we use three ways to calculate $S_{x_1y_1x_2y_2}$. First, $S_{x_1y_1x_2y_2}$ is $T_{x_1y_1x_2y_2}$, which means $S_{x_1y_1x_2y_2}$ only contains a tuple itself like Matrix0. Second, $S_{x_1y_1x_2y_2}$ can be vertically cut into two parts such as Matrix1, Matrix2 and Matrix3. Third, $S_{x_1y_1x_2y_2}$ can be horizontally cut into two parts such as Matrix4, Matrix5 and Matrix6. We cut $S_{x_1y_1x_2y_2}$ into two parts S_1 and S_2, which means the selected tuples of $S_{x_1y_1x_2y_2}$ is the sum of the selected tuples of S_1 and S_2. For example, if we want to calculate S_{0033} and the Matrix3 has the smallest sum of the lookup time, S_{0033} can be combined by S_{0023} and S_{3033}. Through the backtracking algorithm, we can know that S_{0023} contains $Tuple_{0022}$ and $Tuple_{0332}$, S_{3033} contains $Tuple_{3031}$ and $Tuple_{3233}$. Therefore, S_{0033} contains four tuples: $Tuple_{0022}$,$Tuple_{0332}$,$Tuple_{3031}$ and $Tuple_{3233}$.

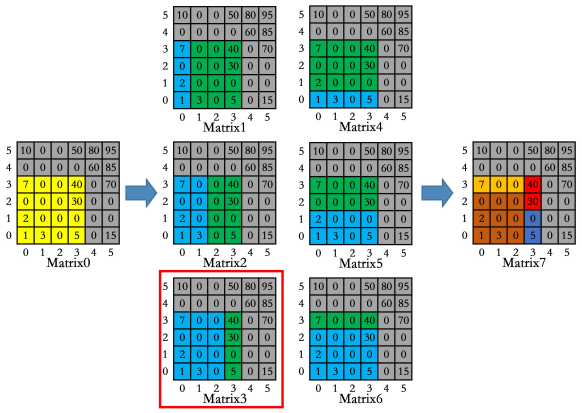


Figure 5: The method of combining tuples.

The pseudo code is shown in Algorithm 2. Line1-2 enumerate the width and the height of the matrix and traverse matrix size from small to large. Line 3-4 enumerate the lower left corner of the matrix $[x_1, y_1]$. Line5-6 calculate the upper right corner of the matrix $[x_2, y_2]$, which is equal to $[x_1 + w, y_1 + h]$. Line7 indicates that $S_{x_1y_1x_2y_2}$ can be $T_{x_1y_1x_2y_2}$ itself. Line8-10 vertically cut

$S_{x_1y_1x_2y_2}$ into two parts. And Line11-13 horizontally cut $S_{x_1y_1x_2y_2}$ into two parts. At last, line18 returns the lookup time of each matrix S. Through the backtracking algorithm, we get a set of tuples to form a complete data structure to reduce the total lookup time.

Algorithm 2: The dynamic programming for tuple combination problem

Input: the lookup time of each tuple T
Output: the sum of the lookup time for each matrix S;

```
1  for w = 0; w ≤ 32; w++ do
2      for h = 0; h ≤ 32; h++ do
3          for x1 = 0; x1 ≤ 32 − w; x1++ do
4              for y1 = 0; y1 ≤ 32 − h; y1++ do
5                  x2 = x1 + w;
6                  y2 = y1 + h;
7                  Sx1y1x2y2 = Tx1y1x2y2;
8                  for xk = x1; xk < x2; xk++ do
9                      Sx1y1x2y2 =
                         minSx1y1x2y2, Sx1y1xky2 +
                         Sxk + 1y1x2y2;
10                 end
11                 for yk = y1; yk < y2; yk++ do
12                     Sx1y1x2y2 =
                         minSx1y1x2y2, Sx1y1x2yk +
                         Sx1yk + 1x2y2;
13                 end
14             end
15         end
16     end
17 end
18 return S;
```

5 OPTIMIZATION METHOD

In the DTC data structure, we use two fields (source IP and destination IP) to determine the corresponding tuple, which can reduce the number of tuples. And the remaining three fields (source port, destination port and protocol) are only used for verification. It is wasteful if we can not make full use of all five fields. In this section, we propose two optimization methods to improve our data structure: the port hash table and the protocol division.

5.1 The Port Hash Table

Although the lookup speed of DTC relative to the TSS algorithm has been greatly improved, it is not as fast as HyperCuts and PartitionSort in some rule sets. By observing these rule sets, we find that there are a lot of rules with the same source IP and destination IP. In the DTC or TSS data structure, these rules will be placed in a slot using a chain structure. This leads to a situation that the rule chain is too long. If there are n rules in this slot and each rule is matched once, the access number of the rule is On^2.

Even though the source port and destination port of the rules are expressed by ranges, we divide it into two types: the accurate port and the range port. For example, if the source port of a rule is 20∼20, we define it as an accurate source port. If the source port of a rule is 100∼200, we define it as a range source port. In our experimental results, most of the rules in a same slot have different source port and destination port. And about 80% of these rules have the accurate source port or destination port rather than the range port. This feature gives us the opportunity to quickly match the rules with the accurate source port or destination port.

Therefore, we propose the port hash table to optimize our DTC data structure. As shown in Figure 6. The port hash table consists of three parts: source port hash table, the destination port hash table and the rule chain. The rules with the accurate source port are stored in the source port hash table. Other rules with the accurate destination port are stored in the destination port hash table. And the remaining rules with the range source port and the range destination port are stored in the rule chain. If the length of the rule chain in a slot is greater or equal to a threshold number l and the rule chain contains at least a rule with an accurate source port or destination port, it will be replaced by the port hash table. In our experimental results, the lookup speed is the fastest when l is 7. And from another perspective, the small threshold $l = 7$ can reflect the effect of the port hash table.

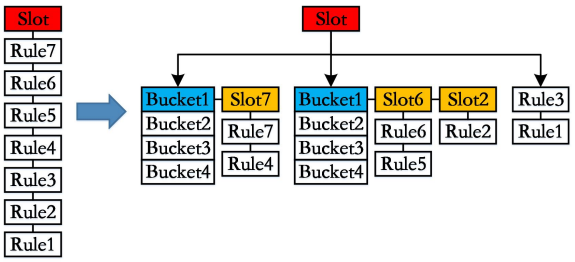


Figure 6: The port hash table.

5.2 The Protocol Division

The feature of the protocol is different from the source port and the destination port. The protocol only has

two types: the accurate protocol and the fully matched protocol. For example, if the protocol is 6∼6, we define it as an accurate protocol. And if the protocol is 0∼255, we define it as a fully matched protocol. Since the types of the protocol number are limited in each rule set, we propose the protocol division method to optimize the lookup process. As shown in Figure 7. The protocol division divide the rules into multiple groups according to the protocol field, for each group, we build a DTC data structure. In a lookup process, the packet only needs to look up in the corresponding DTC data structure according to the protocol of the packet.

The protocol division has two advantages. First, the protocol division reduces the number of rules in each DTC, and the lengths of the rule chains are also reduced. This will improve the lookup speed and the update speed. Second, in the rule verification process, since the protocol matches all the rules in this DTC data structure, we just need to check four fields instead of five fields. This will further improve the lookup speed.

However, the protocol division is a trade-off optimization. There are some rules with fully matched protocol. For example, the protocol field of Rule5 in Figure 7 is $[0, 255]$. These rules will be inserted into all DTC data structures. Therefore, Rule5 is inserted in DTC1, DTC2 and DTC3. This situation will reduce the update speed and increase the memory cost. Considering that DTC has fast update speed and small linear memory cost, the benefits of using the protocol division are far greater than the cost. In addition, We use pointer structure to store Rule5 in these three DTC data structures, which avoids the multiple copies of the five fields information of the rule.

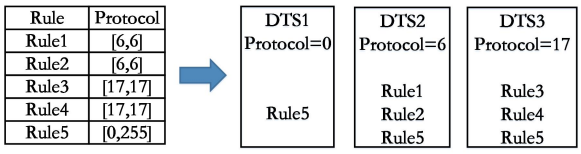


Figure 7: The protocol division.

6 EXPERIMENTAL RESULTS

6.1 Experimental Methods

We compare our DTC to three category methods. The first category is the tuple methods, containing Tuple Space Search (TSS), TupleMerge and PartitionSort. TSS has the current highest update speed. It is implemented in Open vSwitch (OVS). TupleMerge and PartitionSort attempt to trade off between lookup speed and update speed. The second category is the decision Tree method ByteCuts. ByteCuts is a decision tree method with the current highest lookup speed. The third category is DTC_basic and DTC_port. DTC_basic is the algorithm without two optimization methods and DTC_port is the algorithm that only uses the port hash table.

Rule Set: We use ClassBench [21] to generate different types of rule sets. ClassBench is a packet classification benchmark that uses different seeds to imitate rule sets in real environments. It contains twelve parameter seeds: five access control lists (ACL), five firewalls (FW) and two IP chains (IPC). Furthermore, there are other parameters that significantly change the distribution characteristics of the rule set and the packet set. For example, these parameters change the prefix lengths of the source IP and the destination IP, and the scope size of the source port and the destination port. Therefore, we use the sample parameters in ClassBench to generate our data sets and packet sets. For each parameter seed, we generate the rule sets in three sizes: 10K, 50K and 100K.

Tuple Space Search: We use the TSS implementation from GitHub[1]. It uses the cuckoo hash to implement the hash table. And it uses tuple priority order to optimize the lookup process.

TupleMerge: The TupleMerge is implemented by the authors of the paper and published on GitHub[2].

PartitionSort: The PartitionSort is implemented by the authors of the paper and published on GitHub[1].

ByteCuts: The ByteCuts is implemented by the authors of the paper and published on GitHub[3].

DTC: We implement DTC according to this paper. The DTC use both two optimization methods: the port hash table and the protocol division. It will be published on GitHub to see more details.

DTC_port: DTC_port is a DTC algorithm without the protocol division. As mentioned before, the protocol division is a trade-off method. On the one hand, the protocol division improves the lookup speed. However, on the other hand, it affects the update speed. Even though the benefits of using the protocol division are far greater than the cost, we will show the difference between them in our experiments.

DTC_basic: DTC_port is a DTC algorithm without the port hash table and the protocol division. The difference performance of DTC_basic and DTC_port will show the effect of the port hash table.

[1]https://github.com/sorrachai/PartitonSort
[2]https://github.com/drjdaly/tuplemerge
[3]https://github.com/drjdaly/bytecuts

Evaluation metrics: We mainly use four metrics to measure the performance of each algorithm: the lookup speed, the update speed and the memory cost. For each rule set, the packet set contains 100K packets to perform the lookup operation. And it performs 100 rounds to get the average lookup throughput. The update speed contains the insert speed and the delete speed. We use 25% rules in each rule set to perform the insert operation and the update operation. The memory cost is the memory required by each algorithm to build the data structure. Finally, all programs run 10 times to get the average of these metrics.

Experimental environments: The experiments is carried on a computer with Intel(R) Core(TM) i7-8700 CPU@3.20GHz, 6 cores, 64KB L1, 256KB L2, 12MB L3 cache respectively, and 8GB of DRAM. The operation system is Ubuntu 16.04.

6.2 The Tuple-based Methods

We compare the performance in Tuple-based methods, containing TSS, TupleMerge and PartitionSort. First, we calculate the number of tuples, the access numbers of tuples and rules. These are the key figures that effect the performance. Second, we compare the lookup speed, update speed and the memory cost between these methods.

The number of tuples:

We compare the lookup throughput within TSS, PartitionSort, HyperCuts and DTC. The size of each rule set is 10K and we select the first two rule sets from each type to show in Figure 9(a). The lookup speed of TSS is the slowest, because TSS has too many tuples to check in lookup process. The lookup speed of PartitionSort is 7.4 times of TSS, which is consistent with the conclusion of the paper. As a classic decision tree method, HyperCuts performances better than PartitionSort in lookup speed, but it is hard to updates. Because of the protocol division, the lookup speed of DTC is faster than DTC*. In summary, compared to all these three algorithms, our DTC has the fastest lookup speed in each rule set. The lookup speed of DTC is 28.7 times of TSS, 3.7 times of PartitionSort, 2.4 times of HyperCuts and 1.2 times of DTC*.

As shown in Figure 9(b), We compare the lookup throughput in different data size. The data sets are ACL1, FW1 and IPC1 in 10K, 50K, 100K. The lookup speed order from slow to fast are still TSS, PartitionSort, HyperCuts, DTC* and DTC. This shows that DTC has the fastest lookup speed on various sizes of data sets.

Because the implementation of SmartSplit can not work well in each data set, we select some data sets to compare its lookup speed in Figure 9(c). The displayed data sets contain 10K_FW1, 10K_FW3, 10K_FW4, 10K_FW5, 100K_ACL2 and 100K_ACL4. The lookup speed of SmartSplit is faster than HyperCuts. However, DTC still has the fastest lookup speed in each rule set, and the lookup speed of DTC is 1.7 times of SmartSplit.

6.3 Update Performance

We compare the update throughput within TSS, PartitionSort and DTC. Since HyperCuts and SmartSplit are hard to update within the decision tree structure, we can ignore their comparative experiments.

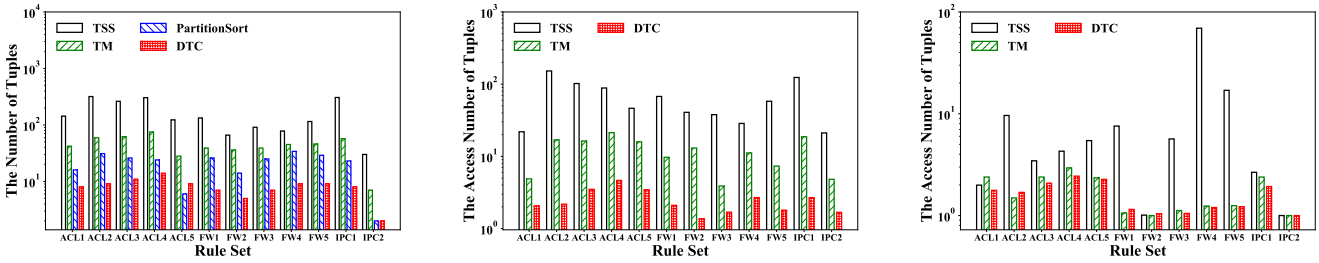
The insert speed is shown in Figure10(a) and the delete speed is shown in Figure10(b). TSS only needs to update in the corresponding tuple and the cuckoo hash updates rapidly. Therefore, TSS has currently the fastest update speed. The update speed of PartitionSort is slower than TSS. Because each tuple in PartitionSort is a decision tree and the updating complexity is logarithmic. The insert speed of DTC is 3.2x faster than TSS and 6.0x faster than PartitionSort. And the delete speed of DTC is 2.0x faster than TSS and 3.8x faster than PartitionSort. The port hash table reduces the lengths of rule chains which makes DTC faster than TSS. Even though the update speed of DTC* is 1.5 times of DTC, the update speed of DTC is still faster than other algorithms.

6.4 Lookup Performance in Update Environment

We use 10K_ACL1 to experiment the lookup performance in update environment. In Figure 10(c), the horizontal ordinate indicates the environmental update speed and the vertical ordinate indicates the lookup throughput with the corresponding environmental update. PartitionSort can not update in 2000Kups and TSS can not update in 5000Kups. Even though the update speed of DTC is slower than DTC*, DTC still has higher lookup performance than DTC* with the environmental update speed less than 2000Kups. Therefore, the protocol division is benefit to the comprehensive performance.

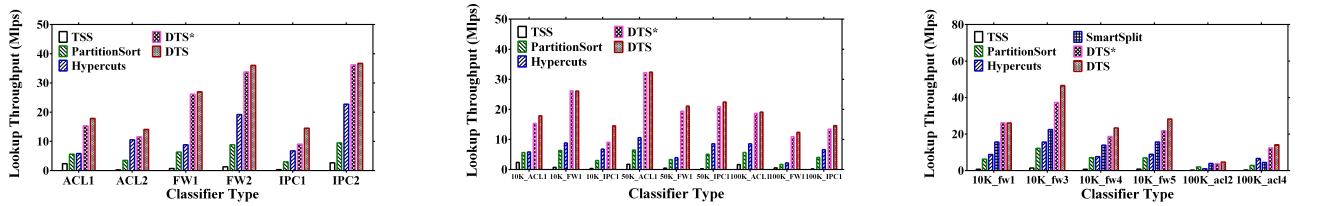
6.5 Memory Cost

The memory cost is shown in Figure 11. we compare the memory cost within TSS, PartitionSort, HyperCuts and DTC. The decision tree of HyperCuts splits rules into multiple parts. Therefore, the memory cost of HyperCuts can explode in some rule sets. The memory cost of TSS is slightly higher than PartitionSort and both of them achieve linear low memory cost. Because of the complex data structure and the protocol division method, the



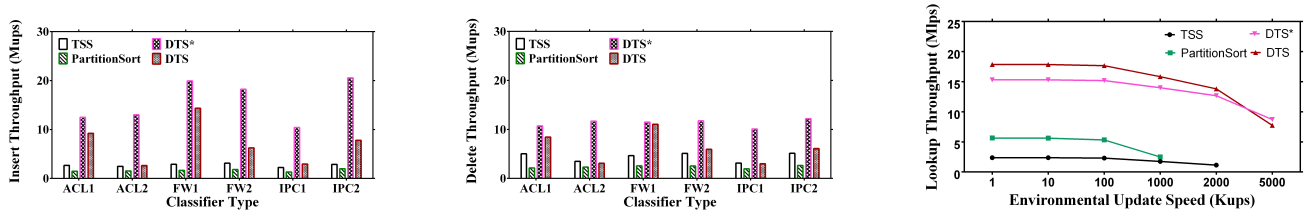
(a) The number of tuples. (b) The average access number of tuples. (c) The average access number of rules.

Figure 8: The figures in tuple-based methods.



(a) In 10K rule sets. (b) In different sizes of rule sets. (c) The lookup throughput of SmartSplit

Figure 9: The lookup throughput in different rule sets.



(a) The insert throughput. (b) The delete throughput. (c) The lookup throughput.

Figure 10: The performance in update environment.

memory cost of DTC is higher than TSS and PartitionSort. However, DTC also achieves linear low memory cost. Considering overall performance, the extra memory cost is worth it.

6.6 DTC* and DTC

As shown in Figure 12, we use twelve types of 10K rule sets to measure the calculate time of DTC. The average calculate time of DTC is 0.58s and the average calculate time of DTC* is 0.29s. Considering that the distribution of rule set may change, we use another thread to calculate the estimation lookup time. If the new tuple combination scheme and the old tuple combination scheme have little difference in the estimation lookup time, we do not have to change the data structure. Otherwise, we use the new tuple combination scheme to rebuild our data structure. Since the distribution of rule set do not

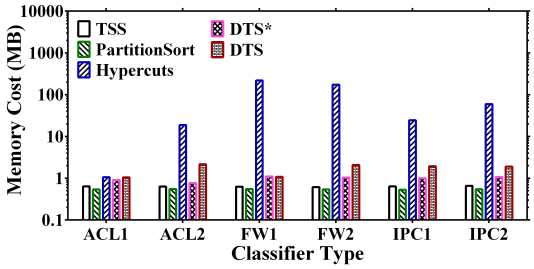


Figure 11: The memory cost.

change dramatically in real environments, our DTC data structure rarely need to be rebuilt.

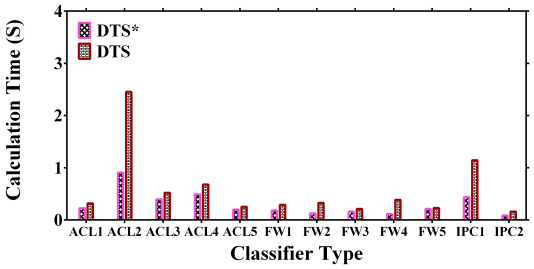


Figure 12: The calculate time.

We count the number of tuples in PartitionSort, DTC* and DTC in Figure 8(a). PartitionSort contains more tuples than DTC* and DTC, and it is not stable as DTC* and DTC. The less tuples improve the lookup speed of DTC* and DTC.Furthermore, the number of more tuples also increases the maximum latency of the lookup.

In Figure 8(b) and Figure 8(c), we calculate the average access number of tuples and the average access number of rules for each lookup operation. The lookup complexity of the decision tree and the binary search tree are *Ologn*, thus the lookup speeds of them are limited. In contrast, both DTC* and DTC has small access number of tuples and rules, which leads to the high lookup speeds of them. In addition, through the protocol division, DTC has less access in tuples and rules than DTC*.

7 CONCLUSIONS

We propose five key contributions. First, we use the Mask Length Reduction method to reduce the number of tuples, which improves the lookup speed, but may causes the long rule chain problem. Second, we build DTC data structure with three priority optimization methods in the lookup process. Third, we calculate the access number of tuples, slots and rules to estimate the lookup time. Fourth, we use dynamic programming method to calculate the tuple combination scheme to reduce total lookup time, which only consume a small amount of time in other thread. Fifth, we propose two optimization methods: the port hash table and the protocol division. The experimental results show that the lookup speed of DTC is 28.5 times of TSS, 3.7 times of PartitionSort, 2.4 times of HyperCuts, 1.7 times of SmartSplit and the update speed is 4.6 times of PartitionSort, 2.4 times of TSS. Furthermore, DTC achieves low linear memory cost like TSS and PartitionSort.

REFERENCES

[1] Anat Bremler-Barr and Danny Hendler. 2010. Space-efficient TCAM-based classification using gray coding. *IEEE Trans. Comput.* 61, 1 (2010), 18–30.

[2] Yeim-Kuan Chang and Han-Chen Chen. 2018. Fast Packet Classification using Recursive Endpoint-Cutting and Bucket Compression on FPGA. *Comput. J.* 62, 2 (2018), 198–214.

[3] Hao Che, Zhijun Wang, Kai Zheng, and Bin Liu. 2008. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Trans. Comput.* 57, 7 (2008), 902–915.

[4] Yu-Kai Chiu, Shanq-Jang Ruan, Chung-An Shen, and Chun-Chi Hung. 2018. The Design and Implementation of a Latency-Aware Packet Classification for OpenFlow Protocol based on FPGA. In Proceedings of the 2018 VII International Conference on Network, Communication and Computing. ACM, 64–69.

[5] James Daly and Eric Torng. 2017. Tuplemerge: Building online packet classifiers by omitting bits. In 2017 26th International Conference on Computer Communication and Networks (ICCCN). IEEE, 1–10.

[6] James Daly and Eric Torng. 2018. ByteCuts: Fast Packet Classification by Interior Bit Extraction. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, 2654–2662.

[7] Jeffrey Fong, Xiang Wang, Yaxuan Qi, Jun Li, and Weirong Jiang. 2012. ParaSplit: A scalable architecture on FPGA for terabit packet classification. In 2012 IEEE 20th Annual Symposium on High-Performance Interconnects. IEEE, 1–8.

[8] Varghese George and J WANG. 2003. Packet Classification Using Multidimensional Cuts. In Proceedings of SIGCOMM.

[9] Pankaj Gupta and Nick McKeown. 1999. Packet classification using hierarchical intelligent cuttings. In Hot Interconnects VII, Vol. 40.

[10] Peng He, Gaogang Xie, Kavé Salamatian, and Laurent Mathy. 2014. Meta-algorithms for software-based packet classification. In 2014 IEEE 22nd International Conference on Network Protocols. IEEE, 308–319.

[11] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. 2015. What you need to know about SDN flow tables. In International Conference on Passive and Active Network Measurement. Springer, 347–359.

[12] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. 2005. Algorithms for advanced packet classification with ternary CAMs. In ACM SIGCOMM Computer Communication Review, Vol. 35. ACM, 193–204.

[13] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. 2018. CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, 2645–2653.

[14] Alex X Liu, Chad R Meiners, and Eric Torng. 2010. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. IEEE/ACM Transactions on Networking (TON) 18, 2 (2010), 490–500.

[15] Zhi Liu, Xiang Wang, Baohua Yang, and Jun Li. 2015. Bitcuts: Towards fast packet classification for order-independent rules. In ACM SIGCOMM Computer Communication Review, Vol. 45. ACM, 339–340.

[16] Yadi Ma and Suman Banerjee. 2012. A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification. In Proceedings of the ACM SIGCOMM

2012 conference on Applications, technologies, architectures, and protocols for computer communication. ACM, 335–346.

[17] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review 38, 2 (2008), 69–74.

[18] Derek Pao, Yiu Keung Li, and Peng Zhou. 2006. Efficient packet classification using TCAMs. Computer Networks 50, 18 (2006), 3523–3535.

[19] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. 2009. Packet classification algorithms: From theory to practice. In IEEE INFOCOM 2009. IEEE, 648–656.

[20] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet classification using tuple space search. In ACM SIGCOMM Computer Communication Review, Vol. 29. ACM, 135–146.

[21] David E Taylor and Jonathan S Turner. 2007. Classbench: A packet classification benchmark. IEEE/ACM transactions on networking 15, 3 (2007), 499–511.

[22] Balajee Vamanan, Gwendolyn Voskuilen, and TN Vijaykumar. 2011. EffiCuts: optimizing packet classification for memory and throughput. ACM SIGCOMM Computer Communication Review 41, 4 (2011), 207–218.

[23] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and TV Lakshman. 2016. Multilayer packet classification with graphics processing units. IEEE/ACM Transactions on Networking (TON) 24, 5 (2016), 2728–2741.

[24] Sorrachai Yingchareonthawornchai, James Daly, Alex X Liu, and Eric Torng. 2016. A sorted partitioning approach to high-speed and fast-update openflow classification. In 2016 IEEE 24th International Conference on Network Protocols (ICNP). IEEE, 1–10.