FastRule: Efficient Flow Entry Updates for TCAM-based OpenFlow Switches

Kun Qiu, Jing Yuan, Jin Zhao, Xin Wang, Stefano Secci, Xiaoming Fu

FastRule: Efficient Flow Entry Updates for TCAM-based OpenFlow Switches

Kun Qiu, *Student Member, IEEE*, Jing Yuan, Jin Zhao, *Member, IEEE*, Xin Wang, *Member, IEEE*, Stefano Secci, *Senior Member, IEEE*, Xiaoming Fu, *Senior Member, IEEE*

***Abstract*—With an increasing demand for flexible management in software-defined networks (SDNs), it becomes critical to minimize the network policy update time. Although major SDN controllers are now optimized for rapid network update at the control plane, there is still room for data plane optimization in terms of update time, when using TCAM-based physical SDN commodity-off-the-shelf switches. A slow update directly affects network performance and creates bottlenecks. To minimize flow entry update time, a dependency graph, a kind of DAG (directed acyclic graph), can be used for the access management of flow entries at the switch. Thanks to the DAG, unnecessary entry movements, which are the main factor slowing down flow entry updates, can be avoided. However, existing algorithms show limitations when updates become very frequent. We propose a new flow entry update algorithm, called FastRule, that exploits a greedy strategy with an efficient data structure to accelerate flow entry update with a DAG approach. Moreover, we also adjust our algorithm for other flow table layouts to make it scalable. We elaborate on the correctness of FastRule and test our algorithm using a hardware switch. Compared with existing algorithms, the evaluation shows that our algorithm is about 100x faster than state-of-the-art solutions with a flow table of $1k$ size.**

***Index Terms*—Software-Defined Networks, TCAM, OpenFlow, greedy algorithm, flow update**

I. Introduction

SOFTWARE-DEFINED NETWORKS (SDN) and OpenFlow [2] are increasingly being adopted by enterprise networks and even carrier networks. The advantage brought by SDN is dynamic network reconfiguration thanks to the global view on network states. An increased spectrum of functionalities is being explored in SDN. How to enhance the response time to network update events such as failures or topology changes is critical, since it determines the agility of the SDN control loop [3]. In the case of failure recovery in carrier networks, re-routing rules in switches has to be finished within $25ms$ [4], to avoid congestion or packet loss. Meanwhile, traffic engineering applications, e.g., B4 [5], also require fast switch reconfiguration to improve network efficiency.

Manuscript received May 3, 2018; revised January 7, 2019; accepted January 11, 2019. The work was supported in part by Natural Science Foundation of China under Grant 61571136, and by 863 program under Grant 2015AA016106. Part of this work was reported in IEEE ICDCS 2018 [1].(Corresponding author: Jin Zhao.)

K. Qiu, J. Yuan, J. Zhao, and X. Wang are with the School of Computer Science, Fudan University, Shanghai 201203, China, and also with Shanghai Key Laboratory of Intelligent Information Processing, Shanghai 200433, China (e-mail:qkun@fudan.edu.cn; yuanj16@fudan.edu.cn; jzhao@fudan.edu.cn; xinw@fudan.edu.cn).

S. Secci is with Cnam, Cedric Lab, Paris, France (e-mail:seccis@cnam.fr).

X. Fu is with Georg-August-Universitaẗ Goẗttingen, Germany (e-mail:fu@cs.uni-goettingen.de).

Although many solutions are proposed to increase the controller processing power in order to shorten control-plane processing latency [6]–[11], they cannot avert the considerable latency in the data plane, which is mainly caused by rule update of the switch [12]. According to the recent measurement results, a commercial OpenFlow switch can only process 42 rule updates in $1s$ [13]. Thus, reducing the rule update latency of switch is a critical task.

Usually, the switching rule (flow entry) update latency is the time to add, delete, modify flow entries in the flow table of SDN switches [14]. The primary reason why OpenFlow switches can perform inefficiently in flow entry update is that they use ternary content addressable memory (TCAM) [15] – a memory architecture that can be seen as an ordered array with parallel look-up ability [16] – whose function is mainly designed for fast entry lookup, not for fast updating. Flow entries in TCAM are usually stored from top to bottom, ordered by decreasing physical addresses. If the header of an incoming packet matches with multiple flow entries, only the entry with the highest physical address is chosen. Thus, during the flow entry update, the switch cannot prevent maintaining the order of entries in the TCAM, which may cause a significant number of movements of existing flow entries [17], [18]. In fact, not only OpenFlow-based switch, but also 5G mobile networks are highly dependent on TCAM update efficiency. It is reported that some new 5G firewall designs use TCAM to increase their detection, differentiation and selective blocking efficiency [19].

The problem can be approached from two dimensions. One is to minimize the number of flow entry updates sent to switches from the control plane [20]–[23]. For example, a modular composition approach [24]–[27] can minimize the number of updates by reducing redundant updates; and Dionysus [17] reduces multi-switch policy update latency caused by suboptimal scheduling. Another way is to design a new firmware with efficient algorithms [28]–[33] in switches that can decrease flow entry movements in TCAM. The minimum dependency graph, a kind of Directed Acyclic Graph (DAG), can avoid unnecessary flow entry movements in the procedure of flow entry update. Utilizing DAG in the firmware needs a policy compiler, whose function is to convert entry update requirement into DAG, and a TCAM update scheduler, whose function is to convert an update in DAG back into a sequence of TCAM entry movements. The state-of-the-art solution called RuleTris [34] mainly focuses on designing an efficient policy compiler, but the poor performance of its TCAM update scheduler leads to large firmware time, say, up to $50ms$ for

one update in a flow table with a size of $1k$ entries.

In order to overcome these limitations, we propose FastRule, an efficient and scalable flow entry update framework that can achieve $0.04ms$ firmware time per-update in a $1k$ size flow table by providing a high-performance TCAM update scheduler. Our scheduler reduces the time complexity of calculating update sequence to $O(c_{avg}(\log n)^2)$ with only $O(n)$ space complexity by a greedy algorithm and an efficient data structure based on Binary Indexed Tree (BIT), where n is the size of TCAM and c_{avg} is the average diameter of subgraphs in the DAG. According to our measurement, a common value of c_{avg} for a $n = 40k$ flow table is less than 15, which is far less than n. We implement our scheduler in the firmware of ONetSwitch [35], a programmable hardware OpenFlow switch. Through hardware evaluation, our solution reveals to be 100x faster than the solution of RuleTris. The evaluation also demonstrates that our solution scales well with the flow table size increases, as shown in our large-scale hardware emulations. We also modify FastRule to satisfy the particular TCAM layouts [32], [33] in order to prove that FastRule can also be utilized in the different type of OpenFlow switches. We elaborate on the correctness of FastRule and prove that we can always find a solution with our algorithm.

The rest of this paper is organized as follows: we first describe the background of TCAM, flow dependency and DAG in section II. In section III, we describe the framework of FastRule. In section IV, we introduce a greedy algorithm for scheduling flow entry movement, and an efficient data structure, BIT, for querying minimum range. In section V, we discuss several flavors of FastRule in different TCAM layouts. In section VI, we evaluate FastRule and analyze the evaluation results. In section VII, we give the related work. We conclude in section VIII.

II. Background

As above mentioned, the TCAM is designed for high-speed packet matching rather than for efficient entry updating in the flow table. The reason for the slow update is that the TCAM must keep the order of flow entries to satisfy a restriction called *flow dependency* [12], [34]. Besides the priority (defined in OpenFlow specification), the *minimum dependency graph*, a kind of DAG, is a widely utilized way to handle *flow dependency*. In this section, we give a brief description of the *flow dependency* restriction, DAG, and how they decrease the TCAM update latency.

A. Flow dependency

Similarly to the route entry that includes a prefix and a forwarding port, the flow entry includes a match field and an action [2]. If an incoming packet matches the match field of a flow entry, the corresponding action is executed. If the match field of two flow entries overlaps, i.e., two flow entries match a same incoming packet, a specific order must be provided to solve the matching ambiguity. The *flow dependency* is such a relationship between two flow entries. Since we can define a flow entry A is *dependent* on a flow entry B if B should be matched first, or A is *dependent* on B if A should be matched first, without loss of generality, we define a flow entry A *is dependent on* a flow entry B if B should be matched first. We also use A $\rightarrow$ B to indicate A *is dependent on* B directly. Moreover, if there is an entry C, and A $\rightarrow$ B $\rightarrow$ C, we can say A *is dependent on* entry C indirectly.

TABLE I
Terms of Definition

Notation	Description
$G = (V, E)$	A flow dependency graph G with node set V and edge set E
n	The number of flow entries or nodes in G
$f_u \in V$, $u \in [0, n]$	A node in DAG, also indicates a flow entry in flow table
$e_{f_u,f_v} \in E$	An edge in DAG, indicates $f_u \rightarrow f_v$
m	The number of flow dependency requirements or edges in G
c_{max}	The largest diameter of the sub-graph in G
c_{avg}	The average diameter of the sub-graph in G
$phyaddr(f_u)$	The physical address that stores f_u in TCAM
$val(A)$	The flow entry or node in physical address A

B. Flow entry update in existing hardware switches

Previous research shows that the main reason for TCAM slow update is the flow dependency maintenance based on an integer index or *priority* [21]. In the TCAM, each flow entry has its physical address [16], and the TCAM always returns the entry with the highest physical address if it matches multiple entries. When adding a new flow entry, the switch firmware finds a correct place: the physical address which must be higher than flow entries with lower priority, and moves all flow entries that physical addresses are lower than the newly arrived one to create a space (unused TCAM entry) in TCAM. Thus, updating a TCAM flow entry is similar to insert sorting algorithm, i.e., if we have n flow entries, we need $n/2$ movements on average to insert a new flow entry into the TCAM.

C. Dependency graph

Moving all flow entries that physical addresses is lower than the newly arrived flow entry will lead to a large number of entry movements. However, it is apparent that only moving flow entries that have a flow dependency relationship with the newly inserted flow entry also meets the flow dependency requirement.

For example, in Fig. 1(b), we need to move 4 flow entries to create a free space for the newly inserted entry if we utilize a priority-based solution, but in Fig. 1(c), only 2 movements are necessary. Thus, directly utilizing flow dependency rather than assigning a priority can significantly decrease the number of movements. The minimum dependency graph, which is a kind of Directed Acyclic Graph (DAG) [22], [34], [36] commonly used to describe the flow dependency in a flow table. We describe our notations in TABLE I. Specifically, we use a node to indicate a flow entry in the flow table, and we use a directed edge from node f_b to f_a to express node entry f_a *is dependent on* entry f_b. Also, if f_a *is dependent on* entry f_b, the physical address $phyaddr(f_a)$ must be higher than $phyaddr(f_b)$. In

(a) A flow entry with match field "C*A" needs to be inserted into the flow table. We use the uppercase letter to indicate match items of the match field. Usually, an entry with wildcard match field must be dependent on a precise one. Such as "**A" → "C*A", "**B" → "A*B ".

(b) If the flow dependency is implied by priority, the firmware introduces a priority 20 to the new flow entry, and schedule the insertion according to 20 into space with physical address 0x5. It needs 4 movements in order to provide the space. We call it a priority-based solution, which is a naïve solution widely utilized in OpenFlow switches.

(c) If the flow dependency is implied by a DAG, the firmware finds that the newly inserted entry has no dependency on entry "A*B" and "**B" in DAG. It needs only 2 movements to provide the space. We call it DAG-based solution.

Fig. 1. An example of flow entry insertion in a TCAM based flow table. We firstly simply introduce how TCAM match works. There are 5 flow entries in the TCAM, and we use an uppercase letter to indicate an entry field in the flow entry. There are 3 match items in the match field: 'A,B,C' indicate a fixed item, and '*' indicate 'ANY' (omitted). 'ANY' means it will match any possible value in the packet header. If there is an incoming packet with packet header "CAA", the flow entry "CAA", "C*A" and "***" are matched, but only "CAA" is the match result. This is because "CAA" has the highest physical address. In (a), we need to insert a new entry with match field "C*A". (b) shows the movements if the flow dependency implied by priority, and (c) shows the movements if the flow dependency implied by DAG. Usually, utilizing DAG can significantly decrease the number of movements.

Fig. 1(c), we can see that it is easy to reduce movements in DAG.

The diameter of a graph is equivalent to the length of the "longest shortest path" between any two nodes in the graph. Intuitively, c_{max} indicates how complex the flow dependency is in a flow table. In most cases, such as in routing tables and access control lists, $c_{max} \ll n$.

Usually, a DAG, converted from a routing table or access control list, may have sub-graphs since the DAG is composed of several disconnected parts. The existence of disconnected parts will not affect the dependency between flow entries, It can decrease the expected number of movements. Thus, $c_{avg} < c_{max} \ll n$.

D. TCAM update scheduler

For inserting a flow entry into the TCAM, after the correct place for the newly inserted entry is chosen, a sequence of flow entry movements is applied in order to make the chosen space free in the TCAM. Such a sequence is called *update sequence*, which is created by the TCAM update scheduler, part of the switch firmware.

We use (I, f, A) to indicate the insert operation, and use (D, A) to indicate the delete operation. For example, we can use the sequence $(I, C*A,$ 0x5), $(I, **A,$ 0x4), $(I, A*B,$ 0x3), $(I, **B,$ 0x2) and $(I, ***,$ 0x1) to indicate the update sequence in Fig. 1(b). Also, we can use the sequence $(I, C*A,$ 0x5), $(I, **A,$ 0x2), and $(I, ***,$ 0x1) to indicate the update sequence in Fig. 1(c).

Due to the large time cost of the priority-based solution, a more efficient algorithm is needed to calculate an update sequence from graph elements (such as nodes and edges).

RuleTris utilizes a dynamic programming algorithm with the time complexity $O(n^2)$ in the TCAM update scheduler to calculate the update sequence. However, it lacks efficiency when n is large. Motivated by our observation that the length of most update sequences is not longer than c_{max}, and the average length is about c_{avg}, we design an optimized algorithm whose time complexity is related to c_{max} or c_{avg}. Moreover, as deleting a flow entry from TCAM is simpler than inserting one in most cases [34], we first discuss the flow entry insertion in Section III and IV.

III. The Working Flow of FastRule

In this section, we give an overview of FastRule. We use Fig. 2 as an example to present the workflow of flow entry insertion in FastRule. The first stage is the compiler, which converts a request of flow entry insertion into a request of node insertion in DAG. There are many approaches to contribute a compiler, and we can apply existing approaches, e.g., the one in RuleTris [34], to our framework. Usually, the output contains a node: a flow entry f, and all flow dependency requirements that f must satisfy. The third stage is the TCAM; we apply the update sequence into TCAM by TCAM API. In our evaluation, we use the API provided by ONetSwitch [35].

The second stage searches for a sequence of TCAM entry movements, i.e., an update sequence, which starts with the newly inserted flow entry and ends with a free space in TCAM. We design an algorithm using a greedy strategy, which is an approach that always takes the locally optimal choice. To put it simply, the algorithm constantly finds the address with the least number of movements for creating a free space for the newly inserted entry in a candidate address set. Each candidate address is associated with an integer metric. The smaller the metric is, the more optimal the address (using less number of movements to create a free space in this address) is considered to be.

Fig. 2 gives a brief workflow of the second stage. Firstly, we must find candidate addresses for f before the greedy algorithm, and these candidate addresses must satisfy the flow

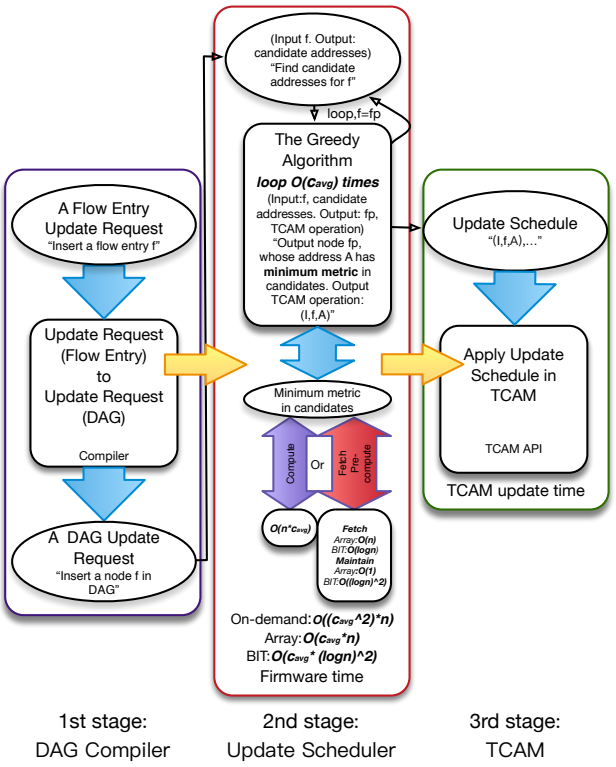


Fig. 2. The working flow of FastRule. We give the input and output of algorithms in the greedy algorithm, and we explain how they work with quotation marks.

dependency requirements of f. Next, an address A with the minimum metric is chosen from the candidates set and its $f_p = val(A)$ is obtained. Then, f is inserted in A to displace f_p, and a TCAM operation (I, f, A) is added to the update sequence. Next, the f_p becomes the new f, and we must find candidate addresses which satisfy the flow dependency requirements of the new f in a new loop. The loop is over if there exists at least one free space in candidate addresses, which means the new f can be put into a free address. Usually, the loop performs $O(c_{avg})$ times.

Choosing the minimum metric from candidates contributes to the most dominant portion of time in FastRule. In the bottom of the second stage, we give three methods, which have different time complexity to get the minimum metric:

1) **On-demand:** Computing the metric of all candidates from scratch every time. Choosing the minimum metric from candidates set has an $O(c_{avg}n)$ time complexity.
2) **Pre-compute with array:** Utilizing an array to save metrics of all candidates, and updating metrics after the loop is over. Choosing the minimum metric from candidates has an $O(n)$ time complexity, while updating metrics has an $O(c_{avg})$ time complexity.
3) **Pre-compute with BIT:** Utilizing a modified Binary Indexed Tree (BIT) to save metrics for all candidates, and updating metrics after the loop is over. Choosing the minimum metric from candidates has an $O(\log n)$ time complexity, updating metrics has an $O(c_{avg}(\log n)^2)$ time complexity.

The BIT is a data structure that is modified to get the minimum metric in all candidates within log time. In Section IV, we briefly introduce our greedy algorithm and BIT, and we also discuss how we optimize FastRule by transforming the **on-demand version** to the **pre-compute with BIT version**. The final time complexity of the **on-demand version** is $O(c_{avg}^2 n)$. The **pre-compute with array version** is $O(c_{avg}n)$, and the **pre-compute with BIT version** is $O(c_{avg}(\log n)^2)$.

As we have mentioned above, the time for the second stage is called *firmware time*, and the time for the third stage is called *TCAM update time*. Compared to previous solutions, FastRule can significantly decrease the firmware time and does not increase TCAM update time in most cases. We evaluate FastRule in terms of firmware time and TCAM update time in Section VI.

IV. Greedy Algorithm

In this section, we first describe how to find candidate addresses for f, and then we give a brief description of the greedy algorithm. We firstly introduce the on-demand version. Then, we introduce the pre-compute with array version, and describe the algorithm of updating metrics. Last, we introduce BIT, and describe how to update metrics in BIT. In this section, without loss of generality, we assume that the highest TCAM physical address and free space are in the top.

A. On-demand: finding candidate addresses

When receiving an incoming request from the DAG compiler, such as inserting a node f with flow dependency requirements $f_a \rightarrow f \rightarrow f_b$, it is precisely that candidate addresses for f range from $phyaddr(f_a)+1$ to $phyaddr(f_b)$. In another case, if node f is an output of the greedy algorithm in the last loop, the flow dependency is $f \rightarrow f_b$, where $phyaddr(f_b) - phyaddr(f)$ is the minimum. In other words, f_b is the nearest node which f depends on. Thus, candidate addresses for f range from $phyaddr(f)+1$ to $phyaddr(f_b)$ in this case.

B. Address metric computation

As we have mentioned, the greedy algorithm needs to choose a candidate address A whose metric is the minimum. We use $M(A)$ to indicate the metric, whose formal definition is given below.

Definition 1. $M(A)$ is the number of nodes in a specific path $P(A)$ in DAG that starts from $val(A)$ and ends with $val(A_l)$. The out-degree of $val(A_l)$ must be 0. For all pair of addresses A_i, A_{i+1} in path $P = \{val(A), val(A_1), val(A_2), ...val(A_l)\}$, they satisfy $A_{i+1} \leq A_t$ for any $A_t \in \{A_t | e_{val(A_i),val(A_t)} \in G\}$.

Intuitively speaking, path P starts from the node in address A, and ends with a node that is not dependent on any nodes. For any pair of addresses A_i, A_{i+1} in path P, the node in A_i is dependent on the node in A_{i+1}, and A_{i+1} must be the nearest address from A_i.

We can use a depth-first search (DFS) algorithm in Algorithm 1 to find P and its length $M(A)$ for any address A:

Algorithm 1: FIND:Finding the path P

Input: Address A, DAG G
Output: P

```
1  s is the neighbor of h with minimum physical address
2  h ← val(A), s ← ∞
3  for w that is the neighbor of h in G do
4      if phyaddr(w) ≤ s then
5          s ← phyaddr(w)
6  P ← P∪ FIND (s)
7  return P
```

finding the nearest node that is depended by the node in the current address (the first address is A); using the searched nodes and its physical address as the input in next search turn. If there is no new node found, the search finishes. The time complexity of the DFS algorithm is $O(c_{avg})$. As G is a DAG that does not have any loop, the algorithm can always get a result.

C. Greedy algorithm

After f, the node to be inserted, and its candidate addresses are available, we can start the greedy algorithm to get the update sequence. We describe the greedy algorithm in Algorithm 2.

We use a recursion form to describe the algorithm. The algorithm finds the address with minimum metric in candidates from line 5 to 9, insert f in A, and output a TCAM operation (I, f, A) to update sequence in 12. We invoke the algorithm with new f_p and new candidate addresses in line 11 to 12. If there exists a free space, the recursion will stop at line 14.

We give an example to describe how it works in Fig. 3.

Algorithm 2: SCHEDULE: Output TCAM update sequence

Input: Candidate addresses $phyaddr(f_a)$ to $phyaddr(f_b)$, node f
Output: Update Schedule $U(f)$

```
1  f_p is the node whose address has the minimum metric
2  A is the physical address of f_p
3  succ(A) is the address of the nearest (in address)
   successor of f_p
4  h is the current minimum metric
5  for k ∈ (phyaddr(f_a), phyaddr(f_b)] do
6      Compute M(k)
7      if M(k) ≤ h then
8          h ← M(k), A ← k
9          f_p ← val(k)
10 if exists succ(A) then
11     f_a ← A + 1, f_b ← succ(A)
12     U(f) ← (I, f, A)∪ SCHEDULE(f_a, f_b, f_p)
13 else
14     U(f) ← (I, f, A)
15 return U(f)
```

Before we prove the correctness of our algorithm, we first give Proposition 1.

Proposition 1. Suppose there is a node f which satisfies $f_a \rightarrow f$ and the out degree of node f is 0. If there exists at least one free space that physical addresses is higher than $phyaddr(f_a)$ in the flow table, it can always find an address A to insert f.

Proposition 1 tells us that if f is not dependent on any node, and the physical address of free space is higher than $phyaddr(f_a)$, it can always be inserted successfully.

Proof 1. In line 5, $phyaddr(f_b)$ is the entry with the highest TCAM physical address in the flow table. After running line 5 to 9, we can always find A with $M(A) = 0$, and f is inserted at line 14.

Then, we can state the following proposition.

Proposition 2. The greedy algorithm can always find a solution if there exists at least one free space in the flow table.

Proof 2. For current node f, and candidate addresses $phyaddr(f_a)$ to $phyaddr(f_b)$, we have the following cases.

1) if the out-degree of f is 0 (f is not dependent on any nodes), then choose an address that is a free space in the candidate addresses set, and insert f into the free space. A solution hence exists.
2) if the out-degree of f is not 0, and there is an address that is a free space in the candidate addresses set, insert f into the free space. A solution hence exists.
3) if the out-degree of f is not 0, and there is no free address in the candidate addresses set, then choose node f_p whose addressing metric is the minimum in the candidate addresses set, and call the algorithm with f_p as new f. From the definition of candidate addresses set, if there is only one address in the set, f_p satisfies the existence condition, $f \rightarrow f_p$.

As G is a DAG (no loop exists), the out-degree of f_p selected in case 3) will reach 0 after a finite number of iterations. Thus, case 3) will turn into case 1) eventually. In case 1), the free space must be higher than $phyaddr(f_a)$. If the free space is lower than $phyaddr(f_a)$, it must be occupied by the previous call in case 2). Thus, according to Proposition 1, the algorithm can find a solution.

The time complexity of the greedy algorithm can be outlined as follows: the time complexity for line 5 to line 9 is $O(c_{avg}n)$, and the greedy algorithm needs c_{avg} times to stop. Thus the total time complexity is $O(c_{avg}^2 n)$.

D. Using an array data structure to store metrics

In Algorithm 2, the function from line 5 to line 9 is choosing the minimum metrics in candidate addresses, which costs most of the time in the greedy algorithm. As we have mentioned above, in order to prevent computing metrics from scratch, we can use an $O(n)$ array to store pre-computed metrics. The tradeoff is the maintenance time of updating the array after each loop. We also use $M[]$ to indicate the array. We describe our maintaining algorithm in Algorithm 3 for $M[]$ after inserting a new node into the flow table.

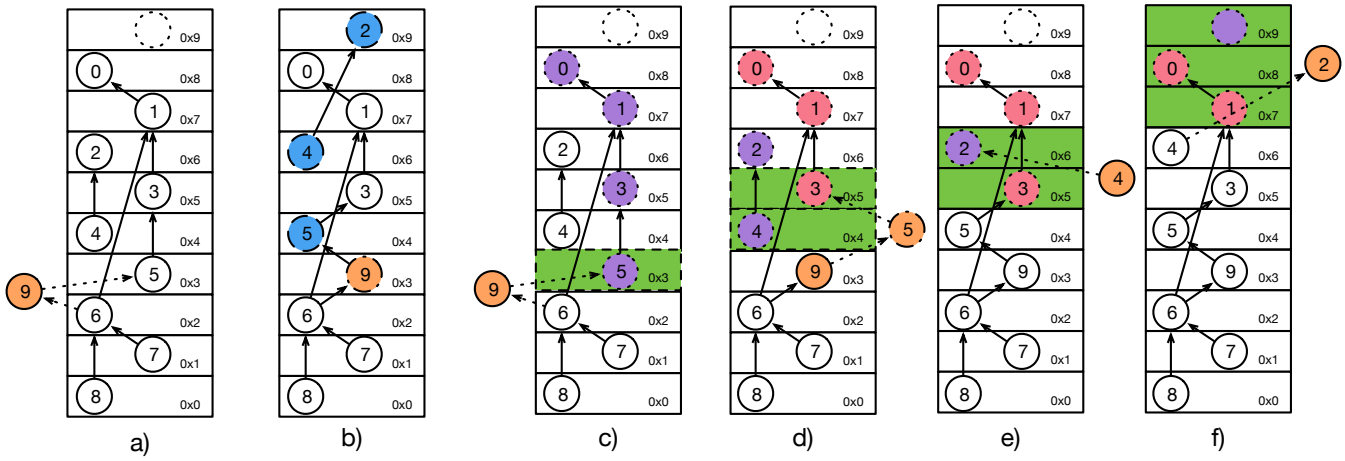


Fig. 3. An example of creating an update sequence for inserting f into DAG. From a) we can see there are 9 nodes (entries) in the DAG (table), and we need to insert a new node 9 that is dependent on node 5, and node 6 is dependent on the new node 9. b) shows the flow table after node 9 is inserted. The length of the update sequence is 4. Only nodes with blue color need to be moved, and the update sequence U(0x3) is (I,9,0x3),(I,5,0x4),(I,4,0x6),(I,2,0x9). We give the detail of the first two callings of algorithm SCHEDULE in remain figures to show how our algorithm works. We use green color to indicate candidate addresses. In c), we call SCHEDULE(0x3,0x3,9). The only selection in candidate address is 0x3, M(0x3)= 4, and P(0x3) is 0x3, 0x5, 0x7, 0x8. We insert node f = 9 at 0x3. The f_p = 5, and we call SCHEDULE(0x4,0x5,5). We have two available selections for A in candidate addresses: 0x4 and 0x5. M(0x4)= 2 and P(0x4) is 0x4, 0x6. M(0x5)= 3 and P(0x5) is 0x5, 0x7, 0x8. We choose 0x4 as A since M(0x4)< M(0x5). The f_p = 4, and we insert node f = 5 at 0x4. In e) and f), we insert node 4 and 2. Eventually, the flow table will become b).

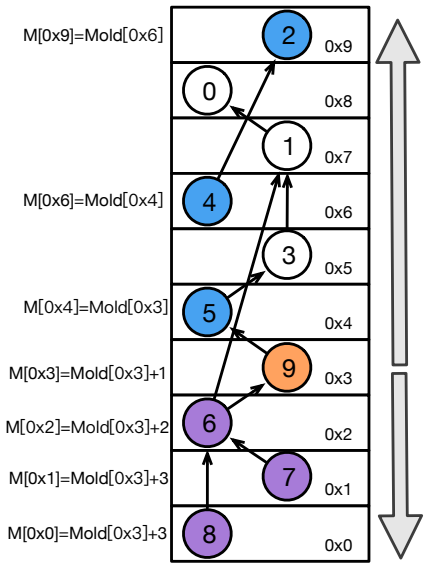


Fig. 4. An example for updating M[] after the greedy algorithm. We update metrics ($M[phyaddr(9)]$, $M[phyaddr(5)]$, $M[phyaddr(4)]$, $M[phyaddr(2)]$) whose addresses in $U(9)$ and we also update metrics ($M[phyaddr(6)]$, $M[phyaddr(7)]$, $M[phyaddr(8)]$). These nodes (6, 7, 8) are directly or indirectly dependent on node 9. We use $M[A] = Mold[B]$ to indicate the update process. $Mold[B]$ is the metric of physical address B before update.

Algorithm 3: UPDATE: Update M[] after inserting node f at A

Input: A, $U(f)$,M[] DAG G
Output: An updated M[]

1 $U(f)$ is the update schedule created by SCHEDULE
2 **for** *let h be the last address to the first address in* $U(f)$ **do**
3 **if** *h is not the last address in* $U(f)$ **then**
4 $M[w] \leftarrow M[h]$
5 $w \leftarrow h$
6 $M[A] \leftarrow M[succ[A]] + 1$
7 Q is a queue, Insert A to Q
8 **while** Q *is not empty* **do**
9 h is the head of Q, dequeue h from Q
10 **if** h *is not* A **then**
11 $M[h] \leftarrow M[h] + 1$
12 **for** w *is the neighbor of* h *that satisfies* $succ[w] = j$ **do**
13 Insert w to Q

From Fig. 4 we can see that the algorithm updates M[] by two steps:

1) Update the metrics for addresses that are in the update sequence $U(f)$.
2) Update the metrics for addresses whose nodes are directly or indirectly dependent on node f.

The time complexity can be depicted as follows. Modifying an element in array M[] costs $O(1)$. From step 1) we have to update metrics in the update sequence and the c_{avg} is the length of the update sequence, the time complexity of step 1) is $O(c_{avg})$. From step 2) we have to update metrics of the node that are directly or indirectly dependent on f. The time complexity of step 2) is $O(c_{avg}(1+d_{in}))$ while d_{in} is the average in-degree of G. We find $d_{in} < 1$ in all data sets, which means that most flow entries are not depended on by other entries. Thus, the total time complexity of updating metrics is $O(c_{avg})$. The time complexity of the greedy algorithm decreases to $O(c_{avg}n)$ since the time complexity of line 5 to line 9 decrease to $O(n)$.

E. Using a modified Binary Indexed Tree to store metrics

As we have mentioned in the previous section, utilizing Binary Indexed Tree (BIT) (also called Fenwick Tree) can decrease the time complexity of line 5 to line 9 in Algorithm 2 to $O(\log n)$. We discuss the design of BIT in detail in this section.

1) Original BIT data structure: The original BIT data structure is used for quickly calculating the cumulative frequency. Suppose there is an array $R[]$ that has n elements, BIT can get the sum

$$R[1 \dots a] = \sum_{i=1}^{a} R[i], a \in [1, n]$$

in $O(\log n)$ with space complexity $O(n)$ by maintaining an array $B[]$. As each integer p can be represented as $2^{k_1} + 2^{k_2} + \cdots + 2^{k_q}$, For example. $11 = 1 + 2 + 8$, the sum $R[1 \dots 11]$ can also be represented by $R[1 \dots 11] = R[11(1)] + R[9 \dots 10(2)] + R[1 \dots 8(8)]$. It is easy to decompose an integer p into $2^{k_1} + 2^{k_2} + \cdots + 2^{k_q}$ by utilizing a function $\text{LOWBIT}(x) = x\&(-x)$ (& is bit and). LOWBIT(x) can get the integer whose value equals to the rightmost 1 in the binary presentation of x, such as $\text{LOWBIT}(1011_2) = 0001_2$, $\text{LOWBIT}(1010_2) = 0010_2$. BIT uses an another array $B[]$ to store the sum

$$B[x] = \sum_{i=x-\text{LOWBIT}(x)+1}^{x} R[i], x \in [1, N]$$

like $B[11] = R[11], B[10] = R[9 \dots 10], B[8] = B[1 \dots 8]$. If we use binary to represent an integer, such as $11 = 1011_2$, we can find 11 can be decomposed by minus the rightmost 1 in its binary presentation. For example, $1011_2 = 1010_2 + 0001_2$, $1010_2 = 1000_2 + 0010_2$. Thus, $R[1 \dots 11(1011_2)] = B[11(1011_2)] + B[10(1010_2)] + B[8(1000_2)]$. For computing any $R[a \dots b], a, b \in [1, n]$, we can compute $R[1 \dots b] - R[1 \dots a]$ directly.

2) Minimum Range Query: In array based Algorithm 2, the function of line 5 to line 9 is to find the minimum metrics in array $M[]$, which takes $O(n)$ times. This is a minimum range query problem, which can be solved by a modified BIT in $O(\log n)$ times.

We can perform a minimum range querying by changing the definition of $B[]$ from sum ($\sum$) to minimum value:

$$B[x] = min(M[i]), i \in [x - \text{LOWBIT}(x) + 1, x], x \in [1, n]$$

From Fig. 5(a) we can see that querying the $min(M[a \dots b])$ by computing $min(M[1 \dots b], M[1 \dots a])$ is not possible. We can only query $M[a \dots b]$ by decomposing the range $[a, b]$ into several ranges in $B[x] = min(M[(x - \text{LOWBIT}(x) + 1) \dots x])$ and find the minimum value in these ranges. We give our algorithm in Algorithm 4. The time complexity is $O(\log n)$.

3) Update: In the original BIT, if we update $M[i]$ by increasing or reducing a value Δ, we only need to increase or decrease Δ in all $B[j]$ whose range includes i. However, in the scenario of minimum range query, we need to recompute all $B[j]$ whose range includes i. However, if we directly compute each $B[j]$ by definition, the time complexity will be $O(n \log n)$. From Fig. 5(b), we can use computed $B[j - 2^k], 2^k < \text{LOWBIT}(j) \leq 2^{k+1}$ to update $B[j]$. Thus, we

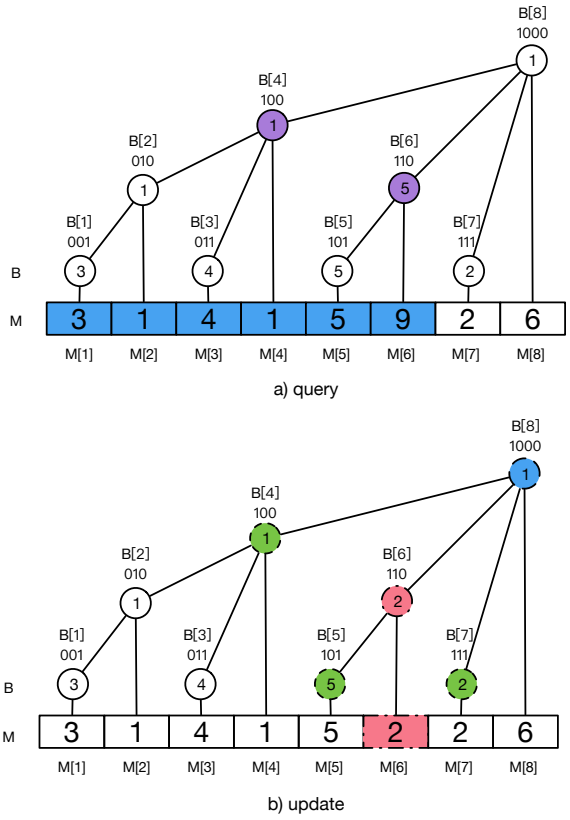


Fig. 5. An example for querying and updating BIT. In a), we query the minimum value in $M[1 \dots 6]$, which can be decomposed as $B[4] = M[1 \dots 4]$ and $B[6] = M[4 \dots 6]$. Thus, we can only compare $B[4]$ and $B[6]$ to get the minimum value is 1. In b). we update the value of $M[6]$ from 9 to 2. Thus, we have to check all ranges that include $M[6]$. The first is $B[6]$, which is the minimum value of $M[5]$ and $M[6]$, and we update $B[6] = 2$. The next is $B[8]$, which is the minimum value of $B[4]$, $B[6]$, $R[7]$, $R[8]$. Due to $B[4] = 1$, we do not change the value of $B[8]$.

Algorithm 4: QUERYBIT: Query minimum value $M[a \dots b]$ in BIT

Input: $M[]$, BIT B, range a, b
Output: The minimum value in $M[a \dots b]$

```
r is the minimum value in M[a ... b]
while a ≤ b do
    r ← min(r, M[b])
    b ← b − 1
    while b − a ≥ LOWBIT(b) do
        r ← min(r, B[b])
        b ← b−LOWBIT(b)
return r
```

give the updating algorithm in Algorithm 5. The time complexity is $O((\log n)^2)$. Thus, the time complexity of the greedy algorithm is decreased from $O(c_{avg}n)$ to $O(c_{avg}\log n)$, and the time complexity of updating metrics is increased from $O(c_{avg})$ to $O(c_{avg}(\log n)^2)$. Overall, the time complexity of the BIT version is $O(c_{avg}(\log n)^2)$.

Algorithm 5: UPDATEBIT: Update $M[i]$ in BIT

Input: BIT B, new value $M[i]$, n is size of M and B
Output: The updated BIT B

```
1 j ← i
2 while j ≤ n do
3     B[j] ← R[j]
4     k ← 1
5     while k ≤LOWBIT(j) do
6         B[j] ← min(B[j], B[j − k])
7         k ← k ∗ 2
8     j ← j+LOWBIT(j)
```

V. FASTRULE IN DIFFERENT TCAM LAYOUTS

A. Free spaces interleaved between non-free spaces

In most cases, all flow entries are arranged in the bottom (or top, depending on the arrangement of physical address) of the TCAM (such as in the leftmost figure in Fig. 6), and the free space is in the top (or bottom) of the TCAM. We call this layout as the original layout. The main disadvantage of this design is that it may occur a large number of entry movements if the new entry is inserted near the bottom (top) of the flow table. In some other interleaved layouts, the TCAM keeps j free spaces (unused TCAM entries) in every i non-free spaces [18], as shown in Fig. 6, in anticipation of inserting and deleting of flow entries. The average loop times of the greedy algorithm improves to i.

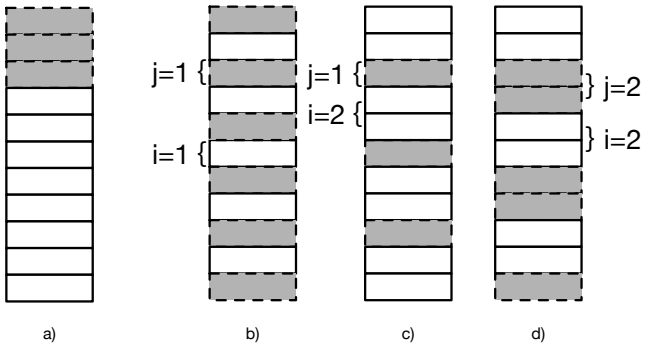


Fig. 6. The TCAM keeps j free spaces (unused TCAM entries) in every i non-free spaces. a) is the original layout, with the free space in the top of the TCAM. We assign $i = 1$ and $j = 1$ in b), which means there exists 1 free space in every 1 non-free space. In c), we assign $i = 1$ and $j = 2$, and in d), we assign $i = 2$ and $j = 2$. Obviously, it is convenient to find a free space for newly inserted entries.

However, it can decrease to c_{max} if all intermediate spaces are filled up. In order to keep this layout, the firmware needs to re-arrange existing entries at intervals. This may involve extra entry movements during the re-arrangement, which is considered as overhead. We evaluate FastRule in this layout by varying i and j in Section VI.

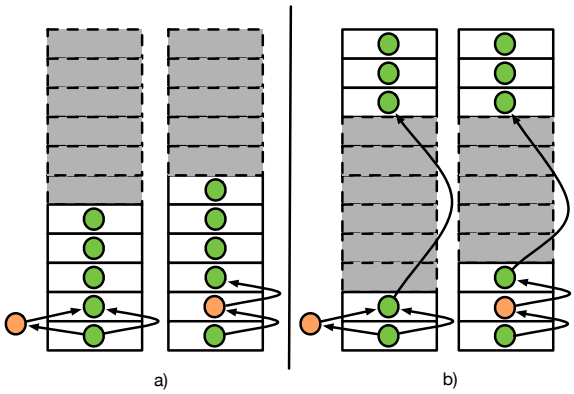


Fig. 7. An example of inserting an entry into the original design and separated design. There are five entries in the flow table. In a), if the new entry is inserted in the bottom of the table, we need 4 moves to create a space for the new entry. In b), if we separate existing entries into two parts located in the bottom and top, only 1 move is necessary to create a space for the new entry.

B. Free spaces in the middle

Another TCAM layout is to separate all flow entries into two parts, one part in the bottom and the other part in the top, and the free spaces are in the middle [33]. Moreover, the insert and delete behaviors in this layout are quite different from the original layout. In this section, we discuss insert/delete behaviors in this particular layout. In the original layout, the time complexity of the greedy algorithm is related to c_{avg}. In this particular layout, we can reduce the time complexity by designing update policies. The largest movements can be upper bounded by $\frac{c_{avg}}{2}$.

As the example shown in Fig. 7 (b), we separate flow entries in the flow table into two parts: some entries in the bottom, and others in the top. Moreover, if we use m_t and m_b to indicate the largest number of movements for the bottom parts and top parts respectively, the largest m_t and m_b are smaller than $\frac{c_{avg}}{2}$ definitely. As the example shows in Fig. 7 (b), this separation moves free spaces into the middle, and it can lower the upper bound of the number of movements significantly.

1) Insert: If the newly inserted node f satisfies $f_a \rightarrow f \rightarrow f_b$, and $phyaddr(f_a)$ and $phyaddr(f_b)$ are both in the bottom part or top part of the TCAM, we just insert f at bottom part or top part. Otherwise, we need to insert f at the free space in the middle part. Before we insert f into the middle part, we need to judge whether the bottom or top is feasible to insert. We compare the number of flow entries m_t and m_b, and define a threshold t; if $m_t - m_b > t$, which means the number of flow entries in the top part is larger than in the bottom part, we insert the new entry in the bottom part. Otherwise, we insert the new entry in the top part. Inserting f in the middle part does not cost any flow entry movements.

An alternative choice is to keep m_t and m_b smaller than $\frac{c_{avg}}{2}$ all the time. This can decrease the maximum number of movements, but may also increase the number of movements

during the maintenance, which is considered as overhead. Balancing m_t and m_b in a proper interval is a good choice.

There is another consideration when we insert flow entry into the free space. As an example in Fig. 8 b), inserting several flow entries in order into continuous free spaces may cost extra movements. An easy solution is to randomly choose a free space from these continuous free spaces each time when inserting the new flow entry.

2) Delete: Deleting a flow entry is more complicated than in the original layout. We have two options:

1) *Dirty delete:* Delete the flow entry, left the space available for newly inserted flow entries.
2) *Balance delete:* Delete the flow entry, and then use other existing flow entries to fill this space.

Both two options have advantages and disadvantages. If we use the *Dirty delete*, free spaces are not in the middle of the flow table; this will waste much space in TCAM. If we use the *Balance delete*, as an example shows in Fig. 9 a) and b), we have to move other entries to the free space, which is considered as overhead. Moreover, we still have to balance entries between the top and bottom to make sure $|m_t - m_b| \leq t$.

We have evaluated both this layout (insert with dirty delete, insert with balance delete) and the original layout in our evaluation section to show the differences in efficiency.

VI. Evaluation

We evaluate FastRule through experiments on ONetSwitch [35], which is an open-source hardware OpenFlow switch. ONetSwitch is a ZedBoard with an up to 800Mhz Cortex-A9, 512MB DDR3 RAM. We use C++ to implement our framework and use g++ provided by Xilinx to cross-compile without any optimization. We evaluate the average TCAM update time and firmware time by measuring $1,000$ random updates.

1) Large size TCAM emulation: The original TCAM in ONetSwitch is pretty small (256 entries in ONetSwitch45), which is not enough for the experimental evaluation. RuleTris solves this problem by emulating a large size TCAM in a Linux server; in the server, they evaluate their algorithm on the emulated TCAM and output the number of TCAM moves that is needed for TCAM update – as each TCAM move costs a constant amount of time ($0.6ms$), it uses the total number of TCAM moves times the average latency of a TCAM move to estimate the TCAM update time. We use a more accurate way to emulate large size TCAM. Similarly to RuleTris, we also use a Linux server to evaluate our algorithm, but it is only used to ensure the correctness of our algorithm (by checking whether flow entries in emulated TCAM are in the correct physical address). In order to emulate a large size TCAM with a small size (ONS_HW_TABLE_SIZE, defined in ONetSwitch) TCAM in ONetSwitch, we modulo the original address with ONS_HW_TABLE_SIZE (such as $(I, f, A\%256)$, ONS_HW_TABLE_SIZE=256 in ONetSwitch45 [34]) if the original address is larger than or equal to ONS_HW_TABLE_SIZE, and update the TCAM with the modulo address. The update time is not affected by utilizing the modulo address.

TABLE II
Data set

Type	ACL4							
n	250	500	1K	2K	4K	10K	20K	40K
c_{max}	3	3	3	6	3	4	13	5
c_{avg}	1.1	1.0	1.1	1.1	1.1	1.1	1.6	1.1
Type	ACL5							
n	250	500	1K	2K	4K	10K	20K	40K
c_{max}	2	3	3	5	3	3	9	4
c_{avg}	1.0	1.0	1.1	1.1	1.1	1.1	1.2	1.1
Type	FW4							
n	250	500	1K	2K	4K	10K	20K	40K
c_{max}	5	7	3	8	4	4	15	4
c_{avg}	1.5	1.4	1.1	1.6	1.1	1.1	1.6	1.0
Type	FW5							
n	250	500	1K	2K	4K	10K	20K	40K
c_{max}	5	7	5	8	5	5	12	5
c_{avg}	1.4	1.4	1.2	1.3	1.2	1.1	1.2	1.1
Type	ROUTE							
n	250	500	1K	2K	4K	10K	20K	40K
c_{max}	2	3	3	3	3	3	4	4
c_{avg}	1.6	1.6	1.7	1.7	1.7	1.7	1.7	1.8

2) Data set: To confirm that our methods are robust and scalable enough, we evaluate FastRule on various type of flow tables: two from Access Control List (ACL4, ACL5), two from Firewall (FW4, FW5) and one from Routing Table (ROUTE). For ACL4, ACL5 and FW4, FW5, we firstly use the well-known policy generator ClassBench, with configuration names ACL4, ACL5, FW4, FW5 provided in ClassBench [37], to generate policies, and use ClassBench-ng [38] to covert these generated policies into OpenFlow entries. For ROUTE, we download an L3 routing table(routeviews-rv2-20170606) from CAIDA [39], and use ClassBench-ng [38] to convert a subset of prefixes into OpenFlow entries. We summarize characteristics of these flow tables in TABLE II. The number of flow dependency m ranges from 37 to 38225 in ACL4, 3 to 4557 in ACL5, 365 to 24130 in FW4, 168 to 40303 in FW5 and 169 to 31381 in ROUTE. From m we can see d_{in} is small since 10% flow entries have a flow dependency. It is obvious that the number of flow dependency in FW4, FW5 are larger than the number in ACL4, ACL5. Moreover, ROUTE has a larger c_{avg} than others. We also prepare a data set of flow updates to flow tables with each size. We generate 250 updates for the flow table with 250 entries, 500 updates for the flow table with 500 entries, and 1000 updates for the

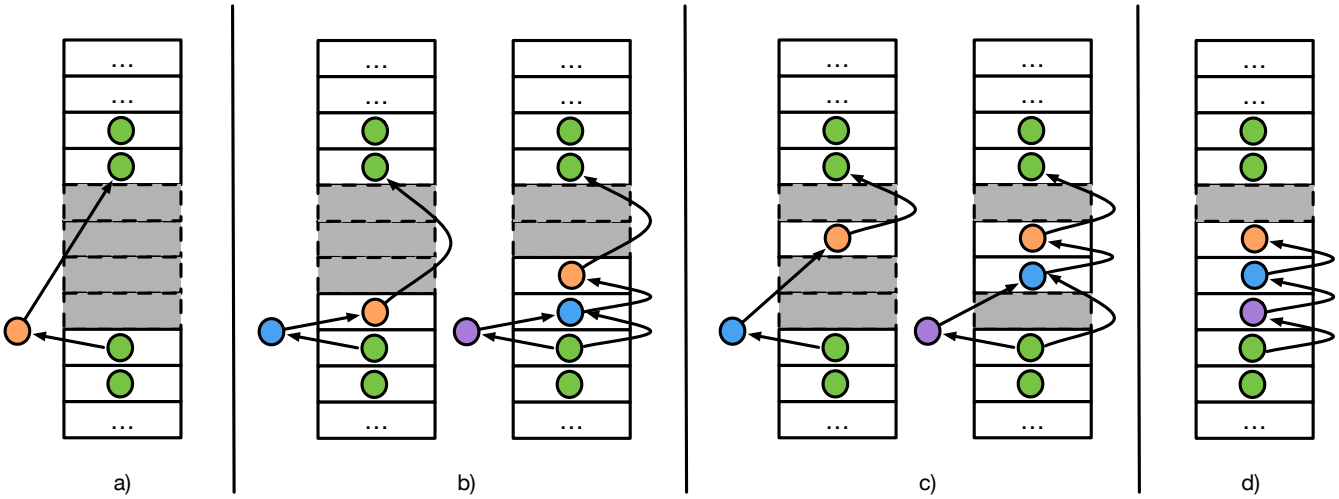


Fig. 8. Inserting flow entries in order continuous free spaces may cost extra movements. a) is the initial state of the flow table, and d) is the final state of the flow table. In b), we insert flow entries in order, we can find we need to move the orange node 3 times, and to move the blue node 2 times. In c), we do not insert flow entries in order, we can find we do need any extra movements for the orange node and the blue node in the best case.

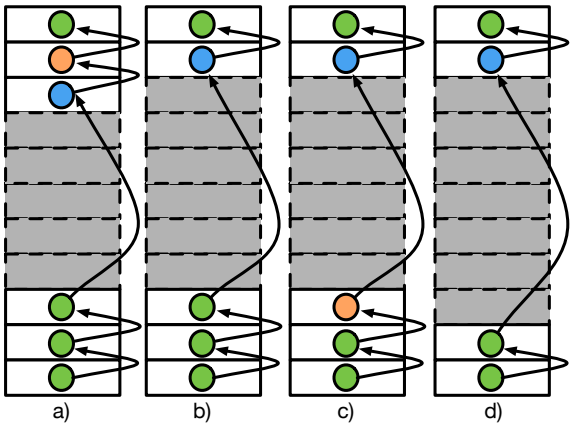


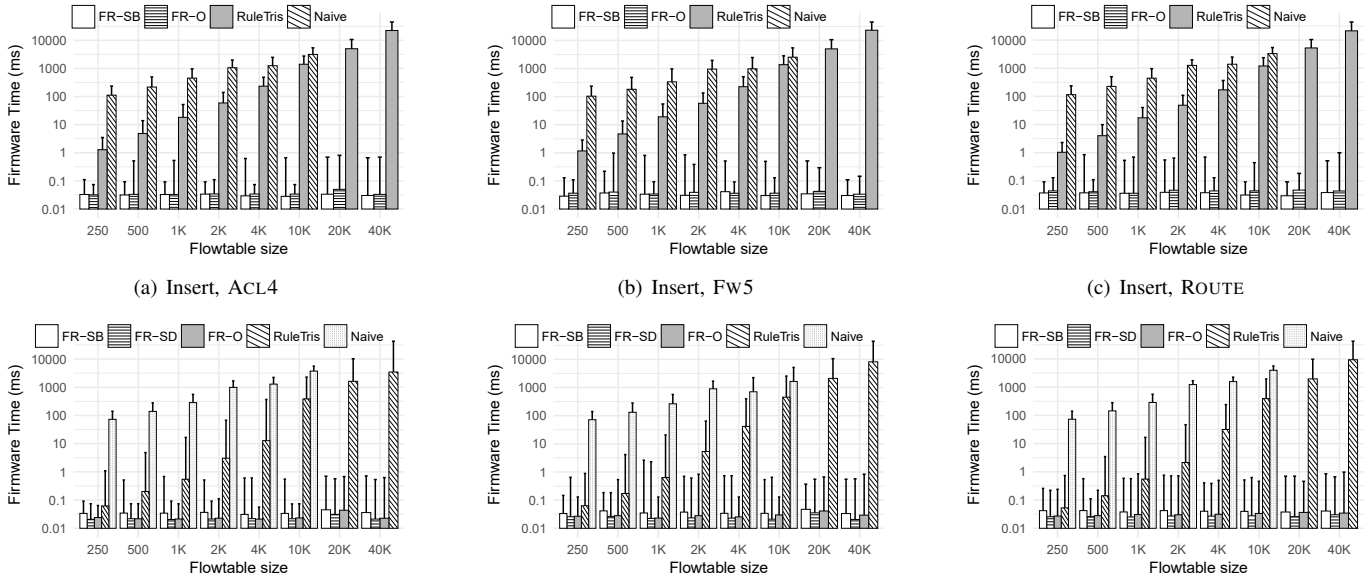
Fig. 9. An example of deleting flow entry. From a) and b), we can see that after the entry (orange node) was deleted, the free space was occupied by other flow (blue node) entry immediately. The situation is the same in c) and d). We just delete the entry (orange node), and we do not need any extra movement.

flow table with $1k, 2k, 4k, 10k, 20k, 40k$ entries. Moreover, we generate two types of update. The first type only contains entry insertion, and the second type contains an entry insertion and an entry deletion after the insertion. As for flow entry insertion, we create a new f that satisfies $f_a \rightarrow f \rightarrow f_b$, where f_a and f_b are randomly chosen from existing entries in the flow table. As for the deletion, we randomly delete a flow entry in the flow table. The reason why we choose random updates is that the time complexity of our algorithm only depends on n and c_{avg}; $1,000$ updates can keep a relatively stable $\log n$, and random insertions will not change c_{avg} much. If c_{avg} increased significantly, newly inserted entries must be inserted like a chain, e.g., the new entry that needs to be inserted is always just dependent on the recently inserted entry.

3) Finite size of TCAM: In each evaluation, the TCAM size equals the sum of existing entries and inserted entries. Thus, for first type, the TCAM sizes are 500 for 250 updates over 250 existing entries, and $1k$ for 500 updates over 500 existing entries respectively. For the second type, TCAM sizes are $1.1k, 2.1k, 4.1k, 10.1k, 20.1k$ and $40.1k$. For the first type of update, the TCAM will be full of flow entries eventually, and for the second type of update, the number of entries in the TCAM will not change. Since the time complexity of our algorithm depends on n and c_{avg}, which is the number of flow entries and the average diameter in the graph, the efficiency of our algorithm does not depend on the finite size of TCAM. If there is a free space in TCAM, our algorithm can find a way to insert a flow entry.

4) Layout and algorithm: In section V, we have introduced the impact of layout and insert/delete behaviors. In the evaluation, we use FR-SB to indicate the separated layout with balance delete, FR-SD to indicate the separated layout with dirty delete, FR-O to indicate the original layout. Moreover, we use RuleTris to indicate the dynamic programming algorithm in RuleTris, and Naïve to indicate the widely used insertion sort algorithm. Also, we evaluate FastRule in TCAM layout with j free spaces in every i non-free spaces. Although the finite size of TCAM does not affect the algorithm efficiency, the separated layout may change c_{avg}, which may result in higher inserting efficiency.

5) Firmware time and TCAM update time: We measure the **firmware time**, which is the time of computing the update sequence from a DAG-based or priority-based flow entry update in the switch firmware: the time is measured from when the computation starts till it is ready to apply the update sequence to the TCAM. We also measure the **TCAM update time**, which includes all update times when applying the update sequence to the TCAM: we fetch the ADDENTRY(), DELETEENTRY() APIs from ONetSwitch SDK, and call these APIs to insert/delete entries in specific physical addresses in TCAM, then the time is measured from the TCAM updating start to the end. The firmware and TCAM update times are



(a) Insert, ACL4 (b) Insert, FW5 (c) Insert, ROUTE

(d) Insert and delete, ACL4 (e) Insert and delete, FW5 (f) Insert and delete, ROUTE

Fig. 10. The firmware time in ACL4, FW5 and ROUTE. We do not put Naïve in $20k$ and $40k$ since Naïve can not finish in half an hour.

measured separately on the physical OpenFlow switch.

A. Firmware time: computing update sequence

Firstly, we show the average firmware time on these flow tables in Fig. 10. We choose ACL4, FW5 and ROUTE with 250 to $40k$ entries in flow tables to show how the overhead increases. In the experiments, we feed 250 updates to the table with 250 entries, 500 updates to the table with 500 entries and $1,000$ updates to tables with other sizes. In Fig. 10(a), 10(b) and 10(c), each update only contains one insert to the ACL4, FW5 and ROUTE tables. In Fig. 10(d), 10(e) and 10(f), every two updates sequentially contain one insert and one delete to the ACL4, FW5 and ROUTE table. We do not add FR-SD in Fig. 10(a), 10(b) and 10(c) since the time used by FR-SB and FR-SD is equal if there is no delete update. The error bar indicates the maximum firmware time in the evaluation.

The firmware times evaluated in ACL4 table are shown in Fig. 10(a) and 10(d). In all cases, the naïve algorithm is the slowest, which takes more than $1,000$ times our algorithms in a $10k$-entry table. The reason is that it needs to locate the suitable place in every update, and assign a new priority for all entries that need to be moved within the TCAM. The RuleTris performs better than the naïve solution (100 times the naïve algorithm in a $2k$ entries flow table), but it is still slower than our algorithms. Moreover, with the increase in flow table size, the time used by the RuleTris and naïve algorithm increase rapidly, but our algorithms can remain stable. Different layout and delete behavior can also affect the efficiency. If updates only contain insert, FR-SB is a little bit faster than the FR-O. However, If the update contains both insert and delete, FR-SB is slower than FR-SD and FR-O. We give a brief analysis in subsection VI-E. Moreover, the maximum firmware times of FastRule algorithms are shorter than the average time of others.

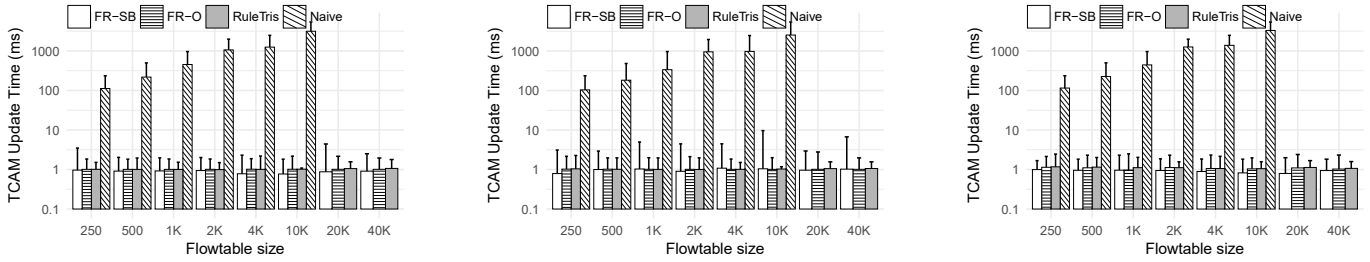
Fig. 10(b), 10(c) and 10(e), 10(f) show the result of firmware time evaluated in FW5 and ROUTE tables. Similarly to the previous experiment, we observe that our algorithms are at least 10 times faster than RuleTris due to the time saved in the firmware time.

B. TCAM update time: time of rule updates on the TCAM

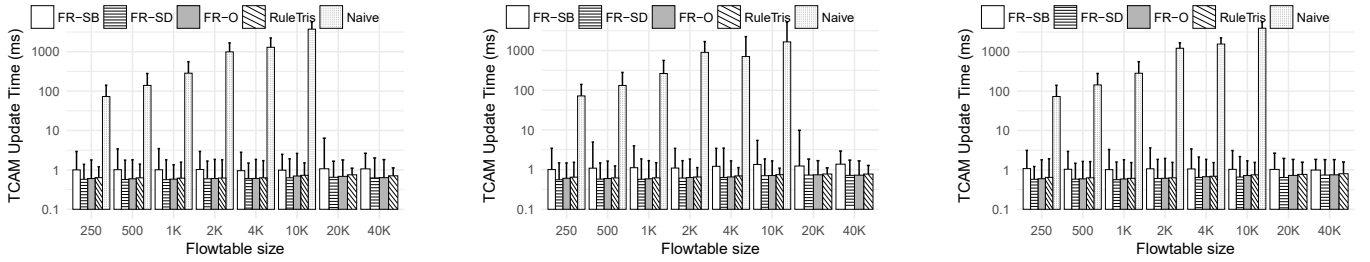
Then, we give the time of rule updates on the TCAM on these flow tables in Fig. 11. In this experiment, we only choose ROUTE in Fig. 11(a) and FW5 in Fig. 11(b) since these two figures are typical and other figures are pretty similar to them. From Fig. 11(a), the TCAM update time of FR-SB and FR-O show no significant differences with RuleTris, which means that our algorithms do not introduce overhead. From Fig. 11(b), we can see FR-SD is the fastest among all algorithms. However, FR-SB is much slower than FR-SD, FR-O and RuleTris. This happens because the FR-SB uses more TCAM movements to perform balance deletion by moving other existing entries to fill the space.

C. The influence of c_{avg}

Because of the time complexity of inserting in FastRule is $O(c_{avg}(\log n)^2)$, which depends on c_{avg}, we give the firmware time among different types of flow tables to show how c_{avg} affects the efficiency of our algorithms. In Fig. 12(a) and 12(c), we can find the time used of FR-O is consistent with c_{avg}. For example, in Fig 12(a), the time used by ROUTE and FW4 in size $2k$ are larger than the time used in another type of tables since the c_{avg} of ROUTE and FW4 are 1.6 and 1.7, which are larger than c_{avg} in other type of tables (ACL4, ACL5 and FW5 are 1.1, 1.1 and 1.3). The situation is the same in Fig. 12(c). Moreover, the average time of FR-SB is a little bit smaller than the FR-O. However, in Fig. 12(b) and Fig. 12(d), the time does not follow c_{avg}. This happens because the time complexity of

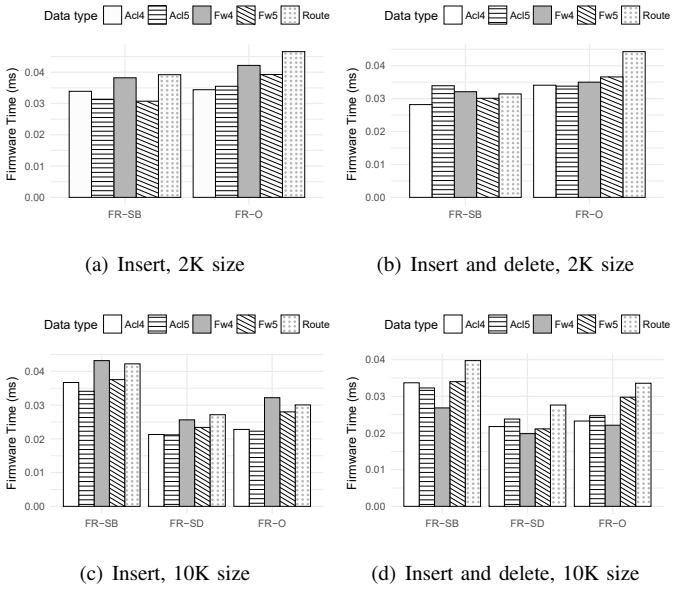


(a) Insert, ACL4 (b) Insert, FW5 (c) Insert, ROUTE



(d) Insert and delete, ACL4 (e) Insert and delete, FW5 (f) Insert and delete, ROUTE

Fig. 11. The TCAM update time in ACL4, FW5 and ROUTE. We do not put Naïve in $20k$ and $40k$ since Naïve can not finish in half an hour.



(a) Insert, 2K size (b) Insert and delete, 2K size

(c) Insert, 10K size (d) Insert and delete, 10K size

Fig. 12. The firmware time among different layouts and delete behaviors.

delete entry operations does not depend on c_{avg}. Moreover, it obviously shows that FR-SB is slower than FR-SD and FR-O in all types of flow tables due to the balance delete overhead, and FR-SD is the fastest method.

D. TCAM layout with j free spaces in every i non-free spaces

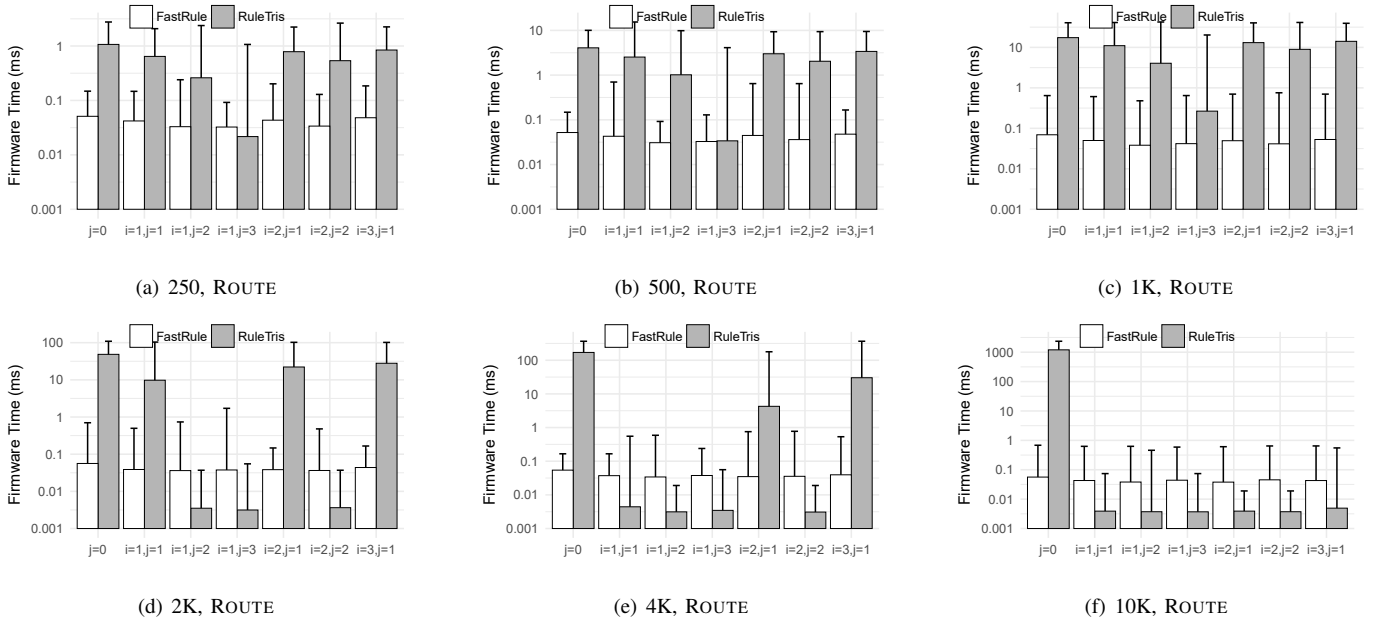
As we have mentioned above, there is another TCAM layout that keeps j free spaces (unused TCAM entries) in every i non-free spaces. In order to confirm that FastRule is scalable in this TCAM layout, we evaluate FastRule in this layout with several combinations that have different i and j. Moreover, we also evaluate RuleTris for comparison.

TABLE III
THE AVERAGE TCAM UPDATE TIME (MS) FOR ONE UPDATE W IN DIFFERENT i AND j (TABLE: ROUTE)
(WITH j FREE SPACES (UNUSED TCAM ENTRIES) IN EVERY i NON-FREE SPACES)

i	j	250	500	1K	2K	4K	10K	20K	40K
1	0	1.15	1.13	1.13	1.13	1.08	1.04	1.13	1.07
1	1	0.84	0.83	0.83	0.64	0.56	0.56	0.52	0.56
1	2	0.60	0.58	0.58	0.56	0.52	0.52	0.58	0.53
1	3	0.59	0.57	0.57	0.52	0.57	0.56	0.54	0.53
2	1	0.91	0.91	0.88	0.69	0.55	0.52	0.52	0.52
2	2	0.69	0.70	0.69	0.59	0.52	0.56	0.56	0.52
3	1	0.99	1.00	0.99	0.81	0.64	0.56	0.58	0.52

We give our evaluation result for the TCAM update time in TABLE III. We give the TCAM update time for ROUTE with flow table size from 250 to 20K. From the result we can see that with the same i, the TCAM update time is smaller when j is larger in most cases. It can be explained that more free spaces between non-free spaces makes fewer movements for the newly inserted entry (the average loop times of the greedy algorithm improves to i). Also, with the same j, the TCAM update time is larger when i is larger in most cases. Moreover, from the result we find that with the increasing size of the flow table, the TCAM update time decreased rapidly except $j = 0$. This conclusion also indicates that arranging free spaces between non-free spaces is better than arranging them in the top (or bottom) of the TCAM.

We give the comparison for the firmware update time between FastRule and RuleTris in Fig. 13. We give the firmware



(a) 250, ROUTE (b) 500, ROUTE (c) 1K, ROUTE

(d) 2K, ROUTE (e) 4K, ROUTE (f) 10K, ROUTE

Fig. 13. The firmware time in ROUTE with different i and j. We do not give figures of 20K and 40K since there are similar to 10K.

update time for ROUTE with flow table size from 250 to 10K. From the result, we can see that with the increasing size of flow table, the firmware time of RuleTris decreased (except $j = 0$). The firmware time of FastRule is about 10 times than RuleTris in 10K size. In other words, if there are large amounts of free spaces between non-free spaces in the TCAM, the firmware time of RuleTris is smaller than FastRule. This is because utilizing BIT costs a constant time in finding the free space. However, the layout with $j = 0$ is the most utilized TCAM layout, which makes FastRule faster in most real-world scenarios.

E. Analysis

In this section, we give some brief analysis to explain the efficiency differences between our algorithms and RuleTris. Moreover, we also analyze the reason why efficiency differences exist among FR-SB, FR-SD and FR-O.

1) Comparison with DP-based solution (such as RuleTris): The efficiency of our algorithm derives from a lower time complexity than previous solutions. The DP-based solution needs a double nested loop: the first loop is iterating all addresses from the address that needs to be inserted to the highest, which needs to take $O(n)$; the second loop is updating the number of movements from current address (the address in the first loop) to the highest address, which takes $O(n)$. Overall, it takes $O(n^2)$ for one insertion. The time complexity of our solution is based on the expected number of movements, which is $O(c_{avg})$.

We only move c_{avg} flow entries for each flow entry update in TCAM. Usually, c_{avg} is very small in real-world data sets. RuleTris utilizes a loop to calculate the potential movements in a range of flow entries that may be moved, and the range may be n. Although c_{max} can also reach n in the worst case, but it seldom occurs in real-world data sets. Moreover, some time-wasting initiation processes (Line 4 to Line 8 in Algorithm 1 of RuleTris [34]) can also be observed in RuleTris, which makes it less efficient than our approach.

2) Efficiency among different layouts and delete behaviors: As we have mentioned above, differences exist between two layouts, and also between delete behaviors. It can be observed that the firmware time with FR-SB is slightly lower than FR-O with pure insert updates, but the firmware time with FR-SB is about 1.5 to 2 times the FR-SD and FR-O times, when the insert updates and deletes update parts count for half each. FR-SB separates all entries into top and bottom, and creates a space in the middle of the flow table. On the one hand, the newly inserted entry falling into the free space can decrease the firmware time of maintaining existing entries, and on the other hand, a separated flow table can decrease c_{max} and c_{avg}, which can also decrease the firmware time since the time complexity of our algorithm is based on n and c_{avg}. However, the situation is different if the proportion of delete updates increased. FR-SB needs to continuously maintain free space in the middle of the flow table, which can increase the delete overhead. If the deleted entry is not near the middle free space, it costs at least one movement to fill the space created by the deleted entry. The FR-SD and FR-O do not cost any movements in delete update, which makes them more efficient than FR-SB.

VII. RELATED WORK

In this section, we introduce some related work aimed at enhancing update efficiency and range minimum querying. There are several existing approaches that can increase flow update efficiency by decreasing flow entry movements. Some approaches utilize new flow table design, e.g. CacheFlow or Mercury [28], [30], others usually utilize DAG to decrease the flow entry movements in incremental TCAM update [18], [29], [31]–[34].

Hierarchical flow table design: As we have mentioned before, although TCAM has high-speed matching efficiency, the power and cost requirements limit its scale for supporting more storage capacity. Though slow in lookup, SRAM has larger storage size, which can be used to increase the capacity of the flow table. CacheFlow [28] maintains a hierarchical flow table design (high-speed TCAM and low-speed SRAM). By utilizing "cover-set", CacheFlow can decrease unnecessary flow entry movements between TCAM and SRAM. Although our work does not utilize hierarchical design, CacheFlow still motivates us towards an efficient algorithm for solving flow dependency.

TCAM update cache: TCAM uses physical address to determine which rule (flow entry) need to be returned when multiple rules are matched. In other words, the TCAM firmware needs to maintain a correct order to keep dependency between flows in TCAM. Usually, the cost of keeping dependency increases with TCAM size. Mercury [30] is a framework that can decrease the number of entry movements by keeping a nominal amount of TCAM space as updating cache. In short, Mercury trades part of the TCAM space for update efficiency.

Incremental TCAM update: A partial order updating theory [29] has been proposed to explore the design space for incremental TCAM updating algorithm. It gives the lower bounds on the TCAM updating performance. Although DUOS [32] and P^2C [31] have considered the flow dependency problem and reduced the updating cost, these solutions still show limitations in computing the updated minimum dependency graph. In contrast, both RuleTris [34] and our work can achieve the minimal updating cost by using the policy compiler, which can generate the minimum dependency information. Chain Ancestor Ordering (CAO) and its optimized version CAO_OPT [18], [33] propose a separated layout design to decrease the number of TCAM updating movements. We have evaluated CAO_OPT (separated layout) in our evaluation section.

Range Minimum Query: Besides the binary indexed tree (BIT), there are also other schemes that can perform minimum range querying in $\log n$ time complexity. The Range Minimum Query (RMQ) problem is defined as finding the minimum element in an array $A[0 \ldots n]$ from indexed L (query start) to indexed R (query end), where $0 \leq L \leq R \leq n$. The simplest solution is to perform a loop in the given range, from R to L. This solution needs $O(n)$ time to query the minimum element. If the array A is static, *Sparse Table* (ST) [40] only has $O(1)$ querying time complexity and $O(n \log n)$ preprocessing time complexity. However, the array A is not static in our situation. A data structure called *segment tree* [41], can be used to speed up the query. Preprocessing the segment tree takes $O(n)$ time, and one range minimum query takes $O(\log n)$ time. Also, segment tree needs extra $O(n)$ space to store the segment tree. Compared to BIT, which only needs an array to store the whole tree, segment tree has to maintain a real tree structure, which needs a significant constant time. Also, the querying algorithm for segment tree is a recursion-based algorithm that is much slower than the querying algorithm in BIT, which is only a loop-based algorithm.

VIII. Conclusion

In this paper, we propose a novel memory update algorithm, called FastRule, to efficiently address the issue of performance bottleneck in TCAM memory update for OpenFlow switches. To decrease the TCAM update latency, we design a greedy algorithm with a specific data structure. First, we propose a fast algorithm with time complexity of $O(c_{avg}^2 n)$ for quickly calculating the update sequence in a flow table of size n, where c_{avg} is the average diameter of a directed acyclic graph. Second, we optimize this algorithm with binary indexed tree to further increase its efficiency, leading to the reduced time complexity of $O(c_{avg}(\log n)^2)$. Moreover, we also optimize our algorithm in some special layouts of the flow table. Meanwhile, We prove the correctness of the greedy algorithm and prove that we can always find a solution with our algorithm. The evaluation results show that our algorithm can be about 100x faster than the state-of-the-art approach, in a $1k$-entry flow table. Furthermore, we analyze the impact of TCAM layouts and delete behaviors on update efficiency. The results demonstrate that FastRule can also decrease the TCAM update latency in scenarios with different layouts and delete behaviors.

References

[1] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "Fast Lookup Is Not Enough: Towards Efficient and Scalable Flow Entry Updates for TCAM-Based OpenFlow Switches," in *Proc. IEEE ICDCS*, 2018, pp. 918–928.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX NSDI*, vol. 10, 2010, pp. 19–19.

[4] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, "Requirements of an MPLS transport profile," Tech. Rep., 2009.

[5] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, 2013, pp. 3–14.

[6] J. Hu, C. Lin, X. Li, and J. Huang, "Scalability of control planes for software defined networks: Modeling and evaluation," in *Proc. IEEE IWQoS*, 2014, pp. 147–152.

[7] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple distributed programming in software-defined networks," in *Proc. ACM SIGCOMM HotSDN*, 2016, p. 4.

[8] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: Towards an Open, Distributed SDN OS," in *Proc. ACM SIGCOMM HotSDN*, 2014, pp. 1–6.

[9] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "SCL: Simplifying Distributed SDN Control Planes," in *Proc. USENIX NSDI*, 2017, pp. 329–345.

[10] B. Davie, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Gude, A. Padmanabhan, T. Petty, K. Duda, and A. Chanda, "A database approach to SDN control plane design," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 1, pp. 15–26, 2017.

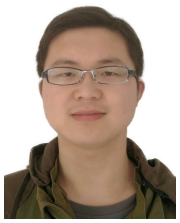
[11] D. L. C. Dutra, M. Bagaa, T. Taleb, and K. Samdanis, "Ensuring end-to-end qos based on multi-paths routing using sdn technology," in *Proc. IEEE GLOBECOM*, 2017, pp. 1–6.

[12] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu, "Compiling minimum incremental update for modular SDN languages," in *Proc. ACM SIGCOMM HotSDN*, 2014, pp. 193–198.

[13] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *Proc. ACM SIGCOMM HotSDN*, 2013, pp. 43–48.

[14] H. Chen and T. Benson, "Hermes: Providing Tight Control over High-Performance SDN Switchesx," in *Proc. ACM CoNEXT*, 2017, pp. 283–295.
[15] F. Long, Z. Sun, Z. Zhang, H. Chen, and L. Liao, "Research on tcam-based openflow switch platform," in *Proc. IEEE ICSAI*, 2012, pp. 1218–1221.
[16] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circu.*, vol. 41, no. 3, pp. 712–727, 2006.
[17] Jin, Xin and Liu, Hongqiang Harry and Gandhi, Rohan and Kandula, Srikanth and Mahajan, Ratul and Zhang, Ming and Rexford, Jennifer and Wattenhofer, Roger, "Dynamic scheduling of network updates," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, 2014, pp. 539–550.
[18] B. Vamanan and T. Vijaykumar, "TreeCAM: decoupling updates and lookups in packet classification," in *Proc. ACM CoNEXT*, 2011, p. 27.
[19] R. Ricart-Sanchez, P. Malagon, J. M. Alcaraz-Calero, and Q. Wang, "Hardware-Accelerated Firewall for 5G Mobile Networks," in *Proc. IEEE ICNP*, 2018, pp. 446–447.
[20] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," in *Proc. USENIX NSDI*, vol. 15, 2015, pp. 87–101.
[21] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu, "Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization," in *Proc. ACM CoNEXT*, 2014, pp. 199–212.
[22] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: simplifying sdn programming using algorithmic policies," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, 2013, pp. 87–98.
[23] S. Vissicchio and L. Cittadini, "Safe, Efficient, and Robust SDN Updates by Combining Rule Replacements and Additions," *IEEE/ACM Trans. Networking*, vol. 25, no. 5, pp. 3102–3115, 2017.
[24] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing software defined networks." in *Proc. USENIX NSDI*, vol. 13, 2013, pp. 1–13.
[25] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," *ACM SIGPLAN Not.*, vol. 49, no. 1, pp. 113–126, 2014.
[26] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Not.*, vol. 47, no. 1, 2012, pp. 217–230.
[27] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, 2011.
[28] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proc. ACM SIGCOMM SOSR*. ACM, 2016, p. 6.
[29] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial Order Theory for Fast TCAM Updates," *IEEE/ACM Trans. Networking*, 2017.
[30] H. Chen and T. Benson, "The case for making tight control plane latency guarantees in SDN switches," in *Proc. ACM SIGCOMM SOSR*. ACM, 2017, pp. 150–156.
[31] J. Van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 4, pp. 560–571, 2003.
[32] T. Mishra and S. Sahni, "DUOS-Simple dual TCAM architecture for routing tables with incremental update," in *IEEE ISCC*, 2010, pp. 503–508.
[33] D. Shah and P. Gupta, "Fast updating algorithms for TCAM," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, 2001.
[34] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE ICDCS*, 2016, pp. 179–188.
[35] ONetSwitch, Open Source Hardware for SDN. Accessed: 2018-02-06. [Online]. Available: https://www.kickstarter.com/projects/onetswitch/onetswitch-open-source-hardware-for-networking
[36] H. Song and J. Turner, "Nxg05-2: Fast filter updates for packet classification using TCAM," in *Proc. IEEE GLOBECOM*, 2006, pp. 1–5.
[37] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Trans. Networking*, vol. 15, no. 3, pp. 499–511, 2007.
[38] J. Matoušek, G. Antichi, A. Lučanskỳ, A. W. Moore, and J. Kořenek, "Classbench-ng: Recasting classbench after a decade of network evolution," in *Proc. IEEE ANCS*, 2017, pp. 204–216.
[39] CAIDA,Center for Applied Internet Data Analysis. Accessed: 2018-02-06. [Online]. Available: https://www.caida.org/home/
[40] M. A. Bender and M. Farach-Colton, "The LCA problem revisited," in *Latin American Symposium on Theoretical Informatics*. Springer, 2000, pp. 88–94.
[41] M. de Berg, M. van Kreveld, M. Overmars, and O. C. Schwarzkopf, "More geometric data structures," in *Computational Geometry*. Springer, 2000, pp. 211–233.

Kun Qiu received his B.Sc. Degree from Fudan University in 2013, and received his Ph.D. Degree from Fudan University in 2019. His research interests include computer network and computer architecture. He stayed at Universite Pierre et Marie Curie and University of Goettingen for half a year as a visiting student. He is a student member of IEEE ACM and CCF.

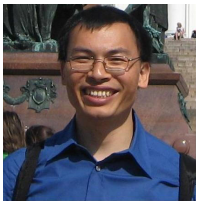


Jing Yuan received his Bachelor Degree from Northwestern Polytechnical University in 2016, and he received his M.Sc. Degree from Fudan University in 2019. His research interest is software-defined networking (SDN).

Jin Zhao received his B. Eng. Degree in computer communications from Nanjing University of Posts and Telecommunications, China, in 2001, and the Ph.D. Degree in computer science from Nanjing University, China, in 2006. He joined Fudan University in 2006. He stayed at University of Massachusetts Amherst for 1 year as a visiting scholar in 2014. His research interests include software defined networking, media streaming and network coding theory. He is a member of IEEE and ACM.

Xin Wang received his Bachelor Degree and the Master Degree from Xidian University, Xi'an, China, in 1994 and 1997 respectively, in Information Theory and Communications. He received the Ph.D. Degree from Shizuoka University, Japan in 2002, in Computer Science. Since 2002, he has been with the School of Computer Science at Fudan University, where he is currently a full professor. He is a member of IEEE and CCF.

Stefano Secci is a Professor of networking at the Cnam, Paris, France. Previously he was associate professor at Sorbonne University-UPMC, Paris, France. He received the M.Sc. Degree in communications engineering from Politecnico di Milano, Milan, Italy, in 2005, and a dual Ph.D. Degree in computer science and networks from Politecnico di Milano and Telecom ParisTech, France, and hold postdoc positions at NTNU, Norway, and GMU, USA. Webpage: http://cedric.cnam.fr/~seccis. He is a senior member of IEEE.



Xiaoming Fu received his Ph.D. in Computer Science from Tsinghua University, China in 2000. He is a professor at the University of Goettingen. He has also held visiting positions at ETSI, University of Cambridge, Columbia University, Tsinghua University, and UCLA. He is a senior member of IEEE, member of ACM.