Fast Lookup Is Not Enough: Towards Efficient and Scalable Flow Entry Updates for TCAM-based OpenFlow Switches

Kun Qiu[*†], Jing Yuan[*†], Jin Zhao[*†], Xin Wang[*†], Stefano Secci[‡], Xiaoming Fu[§]
[*]School of Computer Science, Fudan University, Shanghai, China
[†]Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China
[‡]Sorbonne Université, CNRS, LIP6, France
[§]Georg-August-Universität Göttingen, Germany
{qkun,yuanj16,jzhao,xinw}@fudan.edu.cn, stefano.secci@sorbonne-universite.fr, fu@cs.uni-goettingen.de

***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 creating 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 $1k$ line size.**

I. Introduction

Software-defined networks (SDN) and OpenFlow [1] are increasingly being adopted by enterprise networks and even carrier networks. The advantage brought by SDN is dynamic network reconfiguration thanks to global view on network states. An increased spectrum of functionalities is being explored in SDN, especially to enhance SDN response time to networking events such as failures or topology changes, since it determines the agility of the control loop [2]. In the case of the respond time of failure recovery in carrier networks, re-routing of rules in switches has to be finished within $25ms$ [3], to avoid congestion or packet loss. Meanwhile, traffic engineering applications, e.g., B4 [4], also require fast switch reconfiguration to improve network efficiency.

Although many solutions are proposed to increase the controller processing power in order to shorten control-plane processing latency [5]–[9], they cannot avert the considerable latency in the data plane, which is mainly caused by rule update of switch [10]. According to the recent measurement results, a commercial OpenFlow switch, can only process 42 rule updates in $1s$ [11]. 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 [12]. The primary reason why OpenFlow switches can perform inefficiently in flow entry update is that they use ternary content addressable memory (TCAM) [13] – an memory architecture that can be seen as an ordered array with parallel look-up ability [14] – 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 movements of existing flow entries [15], [16].

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 [17]–[20]. For example, a modular composition approach [21]–[24] can minimize the number of updates by reducing redundant updates; and Dionysus [15] reduces multi-switch policy update latency caused by suboptimal scheduling. Another way is to design a new firmware with some efficient algorithms [25]–[30] 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 [31] mainly focuses on designing an efficient policy compiler, but the poor performance of its TCAM update scheduler leads the firmware time, which is the time used by TCAM update scheduler, up to $100ms$ for one update in a flow table with a size of $4k$ 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 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 a programmable hardware OpenFlow switch ONetSwitch [32]. Through hardware evaluation, our solution reveals to be 100x faster than the solution of RuleTris. The evaluation also demonstrates our solution has a good scalability in a large-scale hardware emulation flow table. We also modify FastRule to satisfy the particular TCAM layout [29], [30] in order to prove that FastRule can also be utilized in 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 paper is organized as follows: we first give a background of TCAM, flow dependency and DAG in section II. In section III, we describe the FastRule solution from the state of the art. In section IV, we introduce our greedy algorithm and BIT in details, and BIT is an efficient data structure that performs minimum range querying. In section V, we present our modification in different TCAM layouts. In section VI, we evaluate FastRule and analyze the evaluation results. We conclude in section VII.

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* [10], [31]. 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 [1]. 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. 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.

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* [18]. In the TCAM, each flow entry has its physical address [14], 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 whose physical address 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 whose physical address 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 flow dependency relationship with the newly inserted flow entry also meets the flow dependency requirement.

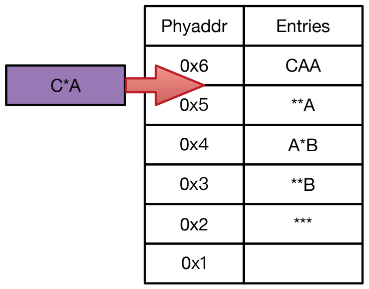
For example, in Fig. 1(b), we need to move 4 flow entries to create a free space for the newly inserted entry if we utilize a priority-based solution, but in Fig. 1(c), only 2 movements are necessary. Thus, directly utilizing flow dependency rather than assigning a priority can significantly decrease the number of movements. The minimum dependency graph, which is a kind of Directed Acyclic Graph (DAG) [19], [31], [33] 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 Fig. 1(c), we can see that it is easy to reduce movements in DAG. Generally speaking, the largest number of movements is small than or equal to c_{max}, which is the largest diameter in all sub-graphs in the DAG. Intuitively speaking, c_{max} indicates how complex the flow dependency is in a flow table. In most cases, such as routing table and access control list, $c_{max} \ll n$.

TABLE I
Terms of Definition

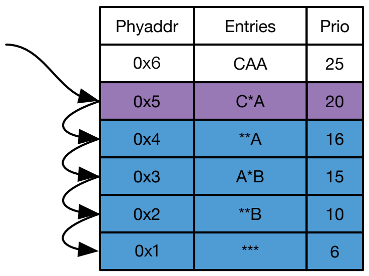
Notation	Description
$G = (V, E)$	A flow dependency graph G with node set V and edge set E
n	The number of flow entries or nodes in G
$f_u \in V$, $u \in [0, n]$	A node in DAG, also indicates a flow entry in flow table
$e_{f_u,f_v} \in E$	An edge in DAG, indicates $f_u \rightarrow f_v$
m	The number of 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

D. TCAM update scheduler

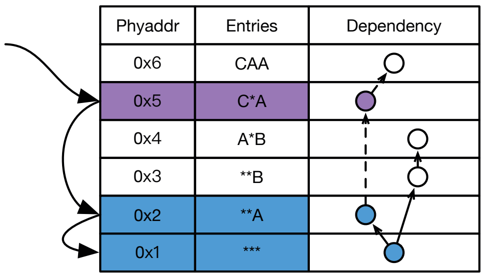
For inserting a flow entry into the TCAM, after the correct place for the newly inserted entry is chosen, a sequence of flow entry movements is applied in order to make the chosen space



(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" $\rightarrow$ "C*A", "**B" $\rightarrow$ "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 the space with physical address 0x5. It needs 4 movements in order to provide the space. We call it priority-based solution, which is a naïve solution widely utilized in OpenFlow switches.



(c) If the flow dependency is implied by 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 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.

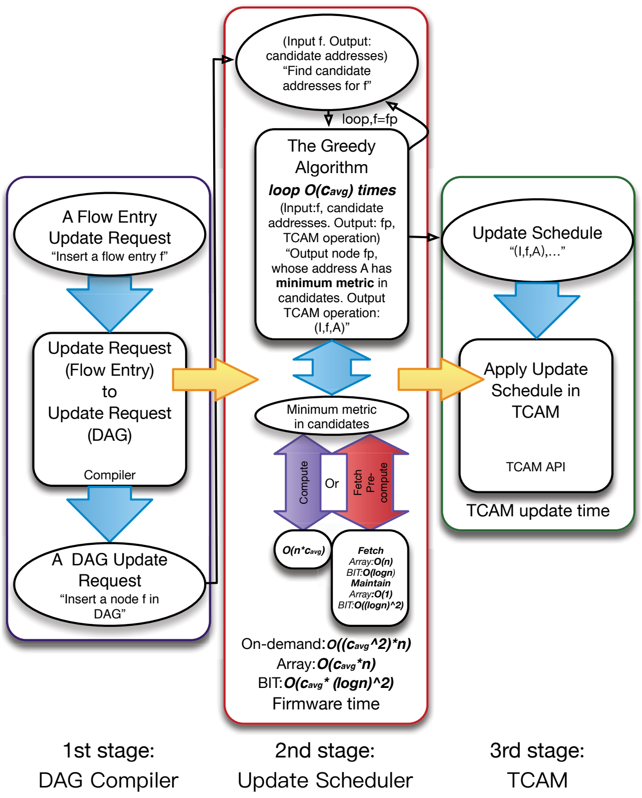
free in the TCAM. Such a sequence is called *update sequence*, which is created by the TCAM update scheduler, part of the switch firmware. We use (I, f, A) to indicate the insert operation, and use (D, A) to indicate the delete operation. For example, we can use the sequence $(I, C*A,$ 0x5), $(I, **A,$ 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 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 [31], we firstly 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 FastRule includes 3 stages, where the second stage is the main one, i.e., converting an update in DAG back into a sequence of TCAM entry movements.

The first stage is the compiler, which coverts 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 [31],



A Flow Entry Update Request
"Insert a flow entry f"
Update Request (Flow Entry) to Update Request (DAG)
Compiler
A DAG Update Request
"Insert a node f in DAG"
(Input f. Output: candidate addresses)
"Find candidate addresses for f"
loop,f=fp
The Greedy Algorithm
loop O(cavg) times
(Input:f, candidate addresses. Output: fp, TCAM operation)
"Output node fp, whose address A has minimum metric in candidates. Output TCAM operation: (I,f,A)"
Minimum metric in candidates
Compute
Or
Fetch Pre-compute
O(n*cavg)
Fetch Array:O(n) BIT:O(logn)
Maintain Array:O(1) BIT:O((logn)^2)
On-demand:O((cavg^2)*n)
Array:O(cavg*n)
BIT:O(cavg* (logn)^2)
Firmware time
Update Schedule
"(I,f,A),..."
Apply Update Schedule in TCAM
TCAM API
TCAM update time
1st stage: DAG Compiler
2nd stage: Update Scheduler
3rd stage: TCAM

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.

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 [32].

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 based on the greedy algorithm, which is an approach that always takes the locally optimal choice. To put it simply, the algorithm constantly finds the most optimal address in a candidate addresses set, for the newly inserted entry. An integer called metric determines whether a candidate address is optimal, accordingly with the principle that the smaller the metric is, the more optimal the 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 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 contribute to the most 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 a time complexity with $O(c_{avg}n)$.
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 a time complexity with $O(n)$, while updating metrics has a time complexity with $O(c_{avg})$.
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 a time complexity with $O(\log n)$, updating metrics has a time complexity with $O(c_{avg}(\log n)^2)$.

The BIT is a data structure that is modified to get the minimum metric in all candidates in a log time. In Section IV, we briefly introduce our greedy algorithm and BIT, and we also propose that how we optimize FastRule from 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 the *firmware time*, and the time for the third stage is called the *TCAM update time*. When compared to previous solutions, FastRule can significantly decrease the firmware time and does not increase TCAM update time in most cases. We evaluate two metrics 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. Without loss of generality, the highest physical address and free space are in the top of TCAM.

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 $f_a \to f \to f_b$, it is precisely that candidate addresses for f is ranged from $phyaddr(f_a) + 1$ to $phyaddr(f_b)$. In another case, if node f is an output of the greedy algorithm in last loop, the flow dependency is $f \to f_b$, where $phyaddr(f_b) - phyaddr(f)$ is the minimum. In other words, f_b is the nearest node f dependent on. Thus, candidate addresses for f is ranged 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. We now give the definition of $M(A)$.

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 to find P and its length $M(A)$ for any address A: 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 is finished. The time complexity of the DFS algorithm is $O(c_{avg})$. As G is DAG that does not have any loop, the algorithm can always get a result.

C. Greedy algorithm

After node f that need 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 1.

We use a recursion form to describe the algorithm. The algorithm finds the address with minimum metric in candidates from line 5 to 9, and 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 is a free space exists, the recursion will stop at line 14.

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

Algorithm 1: 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 adddress 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 $k \in (phyaddr(f_a), phyaddr(f_b)]$ do
6      Compute $M(k)$
7      if $M(k) \le h$ then
8          $h \leftarrow M(k)$, $A \leftarrow k$
9          $f_p \leftarrow val(k)$
10 if exists $succ(A)$ then
11     $f_a \leftarrow A + 1$, $f_b \leftarrow succ(A)$
12     $U(f) \leftarrow (I, f, A) \cup$ SCHEDULE$(f_a, f_b, f_p)$
13 else
14     $U(f) \leftarrow (I, f, A)$
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 satisfies $f_a \rightarrow f$ and the out degree of node f is 0. If there exists at least one free space whose physical address 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 4, $phyaddr(f_b)$ is the maximum physical address TCAM have. After running line 5 to 9, we can always find A that $M(A) = 0$, 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 exits.
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 (the graph is directed and no loop exists), the out-degree of f_p selected in case 3) gets 0, eventually. Thus, the case of the algorithm will be 1) finally. 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 previous call in case 2). Thus, from Proposition 1, the algorithm can find a solution and exits.

About the time complexity of the greedy algorithm: 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 1, 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 the scratch, we can use an $O(n)$ array to save metrics. The tradeoff is maintenance time of updating the array after each loop. We also use $M[]$ to indicate the array. We describe our maintaining algorithm for $M[]$ after inserting a new node into the flow table.

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

1) Update metrics with addresses that in the update sequence $U(f)$.
2) Update metrics with addresses that nodes in these addresses are directly and indirectly dependent on node f.

About the time complexity: 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 most flow entries are not depended by other entries. Thus, the total time complexity of updating metrics is $O(c_{avg})$. The time complexity of 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 1

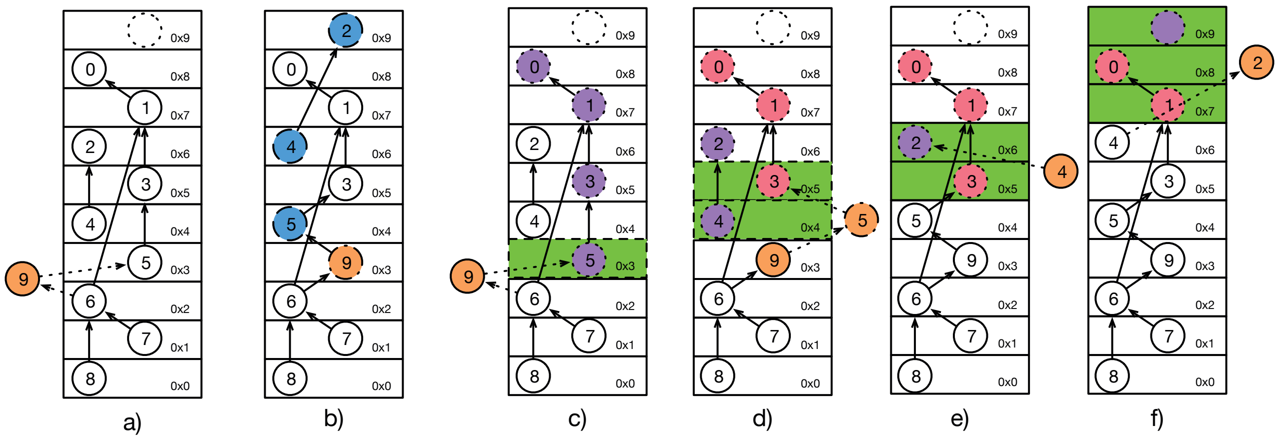


Fig. 3. An example of creating an update sequence for inserting f into DAG. From a) we can see there are 9 nodes (entries) in the DAG (table), and we need to insert a new node 9 that is dependent on node 5, and node 6 is dependent on the new node 9. b) shows the flow table after node 9 is inserted. The length of update sequence is 4. Only nodes with blue color need 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 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 selection 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).

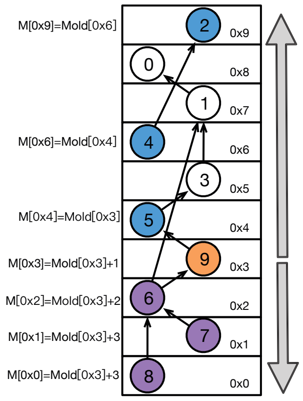


Fig. 4. An example for updating $M[]$ after the greedy algorithm. We update metrics ($M[phyaddr(9)]$, $M[phyaddr(5)]$, $M[phyaddr(4)]$, $M[phyaddr(2)]$) whose addresses in $U(9)$ and we also update metrics ($M[phyaddr(6)]$, $M[phyaddr(7)]$, $M[phyaddr(8)]$). These nodes $(6, 7, 8)$ are directly or indirectly 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.

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

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

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 use an another array $B[]$ to store the sum

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

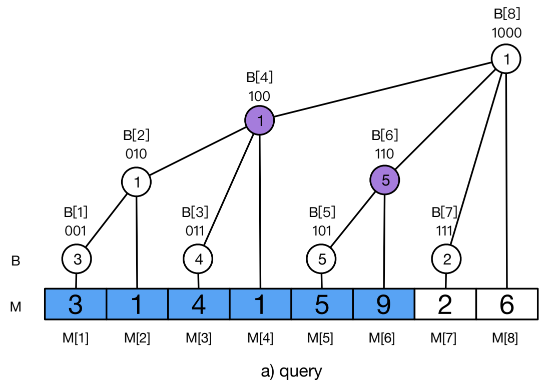
like $B[11] = R[11], B[10] = R[9 \dots 10], B[8] = B[1 \dots 8]$. If we use binary to represent an integer, such as $11 = 1011_2$, we can find 11 can be decomposed by minus the rightmost 1 in its binary presentation. For example, $1011_2 = 1010_2 + 0001_2$, $1010_2 = 1000_2 + 0010_2$. Thus, $R[1 \dots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



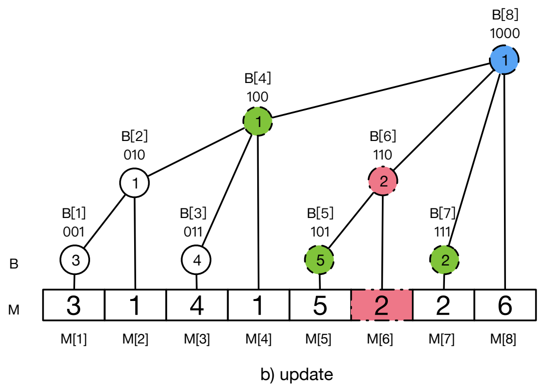


Fig. 5. An example for querying and updating BIT. In a), we query the minimum value in $M[1\dots6]$, which can be decomposed as $B[4] = M[1\dots4]$ and $B[6] = M[4\dots6]$. Thus, we can only compare $B[4]$ and $B[6]$ to get the minimum value is 1. In b). we update the value of $M[6]$ from 9 to 2. Thus, we have to check all range 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 $B[8]$.

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. 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 include i. However, in the scenario of minimum range query, we need to re-compute all $B[j]$ whose range including 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 $B[j]$. 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 update metrics is increased from $O(c_{avg})$ to $O(c_{avg}(\log n)^2)$. Overall, the time complexity of the BIT version is $O(c_{avg}(\log n)^2)$.

V. FastRule in Different TCAM Layouts

In most cases, all flow entries are arranged in the bottom (or top) of the TCAM (such as in Fig. 6(a)), and the free space is

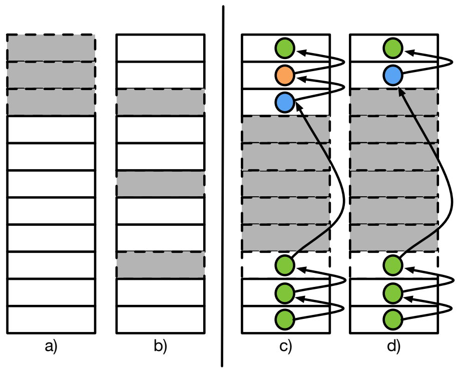


Fig. 6. a) is the original layout, b) is a layout that keeping a free space in every 2 non-free spaces. c) and d) demonstrate an example for balance deleting an entry. From c) and d), we can see that after the entry (orange node) was deleted, the free space was occupied by other flow (blue node) entry immediately.

in the top (or bottom) of the TCAM. We call this layout as the original layout. However, in some other layouts, the TCAM keeps a few free spaces (unused TCAM entries) in every K non-free spaces [16], as shown in Fig. 6(b), in anticipation of inserting and deleting of flow entries. The average loop times of the greedy algorithm improves to K, but it can decrease to c_{max} if all intermediate spaces are filled up.

A further approach is that all flow entries are separated into two parts, one part in the bottom and the other part in the top, and the free space is in the middle [30]. Moreover, the insert and delete behaviors are different from the original layout. In this section, we discuss insert/delete behaviors in this particular layout. As an example shows in Fig. 6(c)(d), we separate flow entries in the flow table into two parts: some entries in the bottom, and others in the top.

1) Insert: If the newly inserted node f satisfies $f_a \to f \to f_b$, 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 insert f at the free space in the middle part. Before we insert f into the middle part, we need to judge whether bottom or top is feasible to insert. If 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. Insert f in the middle part does not cost any flow entry movements.

2) Delete: Deleting a flow entry is more complex then 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. 6 a) and b), we have to move other entries to the free space, which is considered as overhead.

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 [32], 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 [31]) 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 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

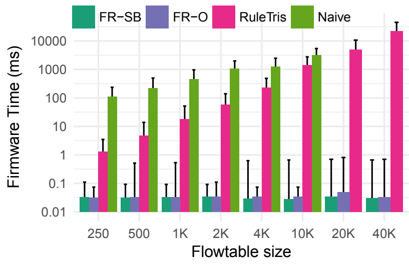
2) Data set: To confirm that our methods are robust and scalable enough, we evaluate FastRule on various type of flow tables: two from Access Control List (Acl4, Acl5), two from Firewall (Fw4, Fw5) and one from Routing Table (Route). For Acl4, Acl5 and Fw4, Fw5, we firstly use the well-known policy generator ClassBench, with configuration names Acl4, Acl5, Fw4, Fw5 provided in ClassBench [34], to generate policies, and use ClassBench-ng [35] to covert these generated policies into OpenFlow entries. For Route, we download a L3 routing table (routeviews-rv2-20170606) from CAIDA [36], and use ClassBench-ng [35] 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 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 flow table with $1k, 2k, 4k, 10k, 20k, 40k$ entries. Moreover, we generate two type 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 chose from existing entries in the flow table. As for the deletion, we randomly delete a flow entry in the flow table.

3) 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 measured separately on the physical OpenFlow switch.

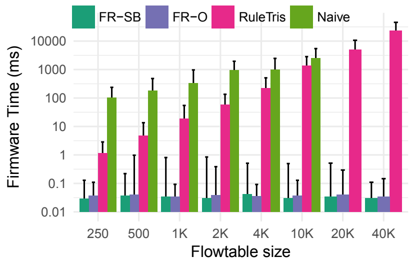
4) Layout and algorithm: In section V, we have introduced the impact of layout and different insert/delete behaviors. In evaluation, we use FR-SB to indicate the separated layout with balance delete, FR-SD to indicate the separated layout with dirty delete, FR-O to indicate the original layout. Moreover, we use RuleTris to indicate the dynamic programming algorithm in RuleTris, and Naïve to indicate the widely used insertion sort algorithm.

A. Firmware time: computing update sequence

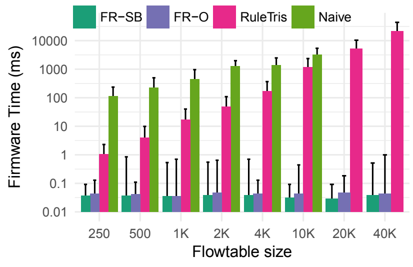
Firstly, we show the average firmware time on these flow tables in Fig. 7. We choose Acl4, Fw5 and Route with



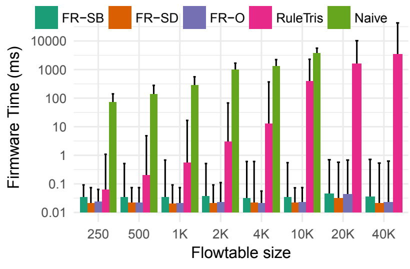
(a) Insert, ACL4



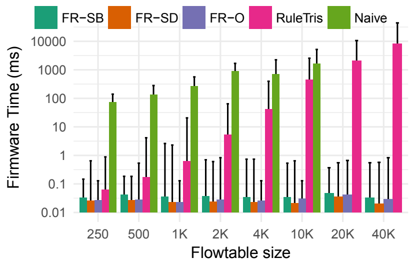
(b) Insert, FW5



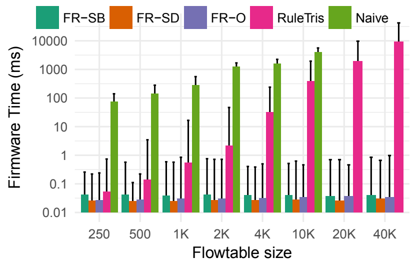
(c) Insert, ROUTE



(d) Insert and delete, ACL4

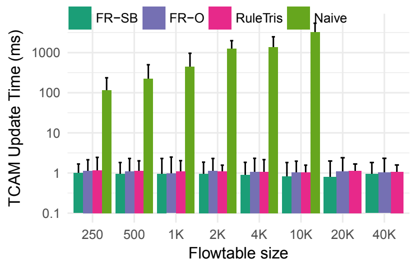


(e) Insert and delete, FW5

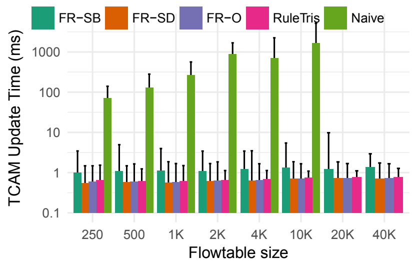


(f) Insert and delete, ROUTE

Fig. 7. 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.



(a) Insert, ROUTE



(b) Insert and delete, FW5

Fig. 8. The TCAM update time in ROUTE and FW5. We do not put Naïve in $20k$ and $40k$ since Naïve can not finish in half an hour.

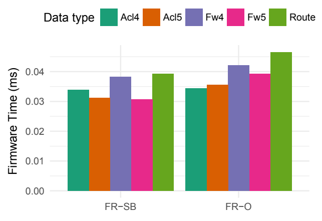
250 to $40k$ entries in flow tables to show how the overhead increases. In the experiments, we feed 250 updates to the table with 250 entries, 500 updates to the table with 500 entries and $1,000$ updates to tables with other sizes. In Fig. 7(a), 7(b) and 7(c), each update only contains one insert to the ACL4, FW5 and ROUTE tables. In Fig. 7(d), 7(e) and 7(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. 7(a), 7(b) and 7(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. 7(a) and 7(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 influence the efficiency. If updates only contain insert, FR-SB is a little bit faster than the FR-O. However, If updates contain both insert and delete, FR-SB is slower than FR-SD and FR-O. We give a brief analysis in subsection VI-D. Moreover, the maximum firmware times of FastRule algorithms are shorter than the average time of others.

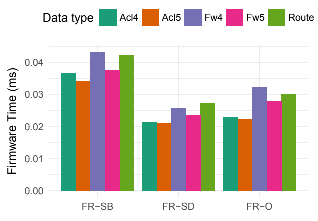
Fig. 7(b), 7(c) and 7(e), 7(f) show the result of firmware time evaluated in FW5 and ROUTE tables. Similarly to the previous experiment, we observe that our algorithms are at least 10 times faster than RuleTris due to the time saved in the firmware time.

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

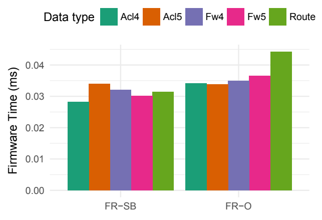
Then, we give the time of rule updates on the TCAM on these flow tables in Fig. 8. In this experiment, we only choose ROUTE in Fig. 8(a) and FW5 in Fig. 8(b) since these two figures are typical and other figures are pretty similar to them. From Fig. 8(a), the TCAM update time of FR-SB and FR-O show no big differences with RuleTris, which means that our algorithms do not introduce overhead. From Fig. 8(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.



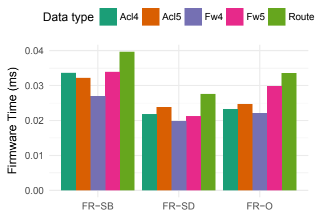
(a) Insert, 2K size



(b) Insert and delete, 2K size



(c) Insert, 10K size



(d) Insert and delete, 10K size

Fig. 9. The firmware time among different layouts and delete behaviors.

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} influence the efficiency of our algorithms. In Fig. 9(a) and 9(c), we can find the time used of FR-O is consistent with c_{avg}. For example, in Fig 9(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. 9(c). Moreover, the average time of FR-SB is a little bit smaller than the FR-O. However, in Fig. 9(b) and Fig. 9(d), the time does not follow c_{avg}. This happens because the time complexity of 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. Analysis

1) Comparison with RuleTris: The efficiency of our algorithms derives from a lower time complexity than previous solutions. We only move c_{avg} flow entries for each flow entry update in TCAM. Usually, c_{avg} is very small in real-world data set. 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 the c_{max} can also be n, 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 [31]) 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 among 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 new 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_{avg}, which can also decrease the firmware time. However, the situation is different if the proportion of delete updates increased. FR-SB needs to continuously maintain a 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. CONCLUSION

In this paper, we propose a scalable rule update algorithm, called FastRule, which is able 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 algorithms with a specific data structure. First, we propose a fast algorithm with a $O(c_{avg}^2 n)$ time complexity for quickly calculating the update sequence in a flow table of size n, where c_{avg} is the diameter of a directed acyclic graph we use. Second, we optimize this algorithm to work at $O(c_{avg}(\log n)^2)$ time complexity, with a data structure to further increase its efficiency. Moreover, we also optimize our algorithm in a special layout of the flow table. Meanwhile, We prove the correctness of the greedy algorithm and prove that we can always find a solution by our algorithm. The evaluation results show that our algorithm can be about 100 times faster than state-of-the-art approach, in a $1k$ size flow table. Furthermore, we analyze the efficiency of different layouts and delete behaviors. The results demonstrate that FastRule can significantly decrease the TCAM update latency in different layouts with different delete behaviors.

ACKNOWLEDGMENTS

The work was partially supported by Natural Science Foundation of China(61571136), 863 program(2015AA016106) and the EU FP7 IRSES MobileCloud Project(612212). Jin Zhao is the corresponding author.

References

[1] 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 Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

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

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

[4] 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 Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[5] J. Hu, C. Lin, X. Li, and J. Huang, "Scalability of control planes for software defined networks: Modeling and evaluation," in *Quality of Service (IWQoS), 2014 IEEE 22nd International Symposium of.* IEEE, 2014, pp. 147–152.

[6] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple distributed programming in software-defined networks," in *Proceedings of the Symposium on SDN Research.* ACM, 2016, p. 4.

[7] 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 *Proceedings of the third workshop on Hot topics in software defined networking.* ACM, 2014, pp. 1–6.

[8] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "Scl: Simplifying distributed sdn control planes." in *NSDI*, 2017, pp. 329–345.

[9] 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 Computer Communication Review*, vol. 47, no. 1, pp. 15–26, 2017.

[10] 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 *Proceedings of the third workshop on Hot topics in software defined networking.* ACM, 2014, pp. 193–198.

[11] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking.* ACM, 2013, pp. 43–48.

[12] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance sdn switches," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies.* ACM, 2017, pp. 283–295.

[13] F. Long, Z. Sun, Z. Zhang, H. Chen, and L. Liao, "Research on tcam-based openflow switch platform," in *Systems and Informatics (ICSAI), 2012 International Conference on.* IEEE, 2012, pp. 1218–1221.

[14] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[15] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 539–550.

[16] B. Vamanan and T. Vijaykumar, "Treecam: decoupling updates and lookups in packet classification," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies.* ACM, 2011, p. 27.

[17] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks." in *NSDI*, vol. 15, 2015, pp. 87–101.

[18] 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 *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies.* ACM, 2014, pp. 199–212.

[19] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: simplifying sdn programming using algorithmic policies," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 87–98.

[20] S. Vissicchio and L. Cittadini, "Safe, efficient, and robust sdn updates by combining rule replacements and additions," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 3102–3115, 2017.

[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing software defined networks." in *NSDI*, vol. 13, 2013, pp. 1–13.

[22] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *SIGPLAN Not.*, vol. 49, no. 1, pp. 113–126, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535862

[23] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 217–230.

[24] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[25] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proceedings of the Symposium on SDN Research.* ACM, 2016, p. 6.

[26] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast tcam updates," *IEEE/ACM Transactions on Networking*, 2017.

[27] H. Chen and T. Benson, "The case for making tight control plane latency guarantees in sdn switches," in *Proceedings of the Symposium on SDN Research.* ACM, 2017, pp. 150–156.

[28] J. Van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560–571, 2003.

[29] T. Mishra and S. Sahni, "Duos-simple dual tcam architecture for routing tables with incremental update," in *Computers and Communications (ISCC), 2010 IEEE Symposium on.* IEEE, 2010, pp. 503–508.

[30] D. Shah and P. Gupta, "Fast updating algorithms for tcam," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, 2001.

[31] 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 *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on.* IEEE, 2016, pp. 179–188.

[32] ONetSwitch, Open Source Hardware for SDN. Accessed: 2018-02-06. [Online]. Available: {https://www.kickstarter.com/projects/onetswitch/onetswitch-open-source-hardware-for-networking}

[33] H. Song and J. Turner, "Nxg05-2: Fast filter updates for packet classification using tcam," in *Global Telecommunications Conference, 2006. GLOBECOM'06. IEEE.* IEEE, 2006, pp. 1–5.

[34] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Transactions on Networking (TON)*, vol. 15, no. 3, pp. 499–511, 2007.

[35] J. Matoušek, G. Antichi, A. Lučanskỳ, A. W. Moore, and J. Kořenek, "Classbench-ng: Recasting classbench after a decade of network evolution," in *Architectures for Networking and Communications Systems (ANCS), 2017 ACM/IEEE Symposium on.* IEEE, 2017, pp. 204–216.

[36] CAIDA,Center for Applied Internet Data Analysis. Accessed: 2018-02-06. [Online]. Available: {https://www.caida.org/home/}