2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-4513-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDCS51616.2021.00089

FastUp: Fast TCAM Update for SDN Switches in Datacenter Networks

Ying Wan*, Haoyu Song†, Hao Che‡, Yang Xu§, Yi Wang††, Chuwen Zhang*,
Zhijun Wang‡, Tian Pan**, Hao Li¶, Hong Jiang‡, Chengchen Hu¶‖, Bin Liu*
* Tsinghua University, China † Futurewei Technologies, USA ‡ University of Texas at Arlington, USA
§ Fudan University, China †† Southern University of Science and Technology, China ‖ Xilinx, Singapore
¶ Xi'an Jiaotong University, China ** Beijing University of Posts and Telecommunications, China

***Abstract*—TCAM is widely used for flow table lookup in Software-Defined Networking (SDN) switches for datacenter and enterprise networks. While its lookup throughput is unparalleled, TCAM updating, particularly for new rule insertions, can impair the overall system performance. A rule insertion entails two steps: 1) Computing the rule moving operations; and 2) Interrupting the TCAM lookups to apply the operations. In previous work, the performance gain on one step is always at the expense of the performance loss on the other. However, update throughput and latency depend on both. In this paper, we present a faster and more balanced TCAM update scheme, which not only achieves the shortest interrupt time so far but also significantly reduces the computation time. By using a novel *sequential stack*, *FastUp* reduces the time and space complexity of the state-of-the-art schemes from $O(m^2)$ and $O(m)$ to $O(m \log h)$ and $O(h)$, respectively, where $h << m$. Evaluations show that *FastUp* shortens the computation time and the interrupt time by 100× and 1.6×, respectively, which is equivalent to 15× update delay reduction and 10× update throughput gain against the state-of-the-art schemes. Moreover, we debunk a common mistake and show the dynamic programming based algorithm cannot be used to solve the reorder problem, and instead we use a bidirectional rule moving method to address the problem. In addition, we propose a practical method to find the theoretical lower bound of interrupt time in relatively large TCAM, which can be used to evaluate the optimality degree of TCAM update schemes. Evaluations show that *FastUp* achieves 90% optimality.**

I. Introduction

Software-Defined Networking (SDN) [1] allows application policies to be enforced, revoked, or modified quickly at runtime and provides a high level of operational flexibility. This is made possible by manipulating (*i.e.*, adding, deleting, or modifying) flow table rules in SDN switches. Today, using Ternary Content Addressable Memory (TCAM) [2] for flow tables becomes the *de facto* industry standard in SDN switches of datacenter and enterprise networks for two reasons: (1) TCAM allows a search key extracted from an incoming packet to compare with all the stored rules in parallel, thus ensuring line-speed forwarding; (2) TCAM rules support a wide range of patterns, including exact matching, Longest Prefix Matching (LPM), and range matching, which are flexible enough for policy representation.

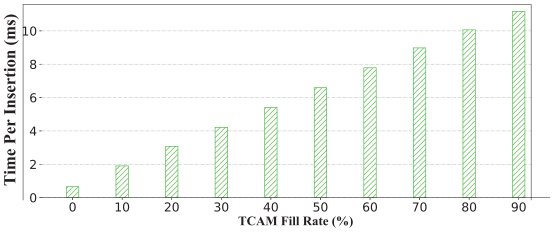


Fig. 1. Measurement results on a Tofino Switch with a 4K-entry TCAM.

However, the application of TCAM is plagued by its limited capacity and high update cost [3]–[6]. Although many solutions are proposed to solve the problem of limit capacity by using TCAM as a cache [5], [7]–[15], they mainly focus on how to improve the TCAM hit-rate, but ignore the accompanying TCAM refreshing problem. The frequent cache refreshing further aggravates the severity of the TCAM update problem.

To better understand the TCAM update problem, we measure the update performance of a high-end switch EdgeCore Wedge 100BF-65X [16]. As shown in Fig. 1, for an idle 4K-entry TCAM-based rule table, the time to insert a rule becomes longer as the TCAM fill-rate increases, reaching up to 11ms when the TCAM is 90% full. The result is consistent with previous reports on other SDN switches (*e.g.*, NoviSwitch 1132, Pica8 P-3290, Dell 8132F, and HP 5406zl), supporting less than 50 rule insertions per second [17]. The poor update performance is attributed to the need for maintaining the order of the rules based on their priority. It fails to meet the requirement of update throughput in networks where policies churn fast [18]–[27], as well as the requirement of update delay imposed by applications [12], [17], [28]–[36]. For example, fast failure recovery leaves no more than 10ms for a routing table update [37]; traffic engineering in datacenters grants only a 20ms budget to activate a new rule [38].

No wonder the TCAM update problem was and remains a hot research topic. Most schemes [4], [39]–[45] are designed to optimize the TCAM update process by reducing the number of rule moves per update. Although the state-of-the-art schemes (Γ_{bh} [42] and *RuleTris* [4]) achieve excellent performance in reducing the rule moves per insertion by using Dynamic Programming Algorithm (DPA), their practicability is impaired

The work of Ying Wan, Yi Wang, and Bin Liu was supported in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2019B1515120031, in part by the National Natural Science Foundation of China under Grant 62032013, Grant 61872213, and Grant 61432009, and in part by "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" under Grant LZC0019. Corresponding Author: Bin Liu (lmyujie@gmail.com).

2575-8411/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDCS51616.2021.00089

by the excessive computation time. Evaluations show that their average computation time is on the order of hundreds of milliseconds for a 4K-entry TCAM, falling short of the above-mentioned update requirements.

Some other schemes (Γ_{down} [42] and *FastRule* [43], [44]) focus on shortening the computation time of Γ_{bh} and *RuleTris* at the cost of increased rule moves. Such an approach is undesirable for two reasons. First, since TCAM lookups and updates share the same interface, the excessive rule moves interfere with the packet forwarding. A study shows that the lookup interrupt caused by moving 16 rules in TCAM leads to 18 packet drops on an OC-192 interface [40]. Second, the throughput of high-end switches has already reached the level of 12.8Tbps per chip [46] and the increasing trend does not seem to recede, whereas TCAM bandwidth has not been scaled up as fast, which means the bandwidth share allotted to updates shrinks [47]. Therefore, reducing the rule moves is still the paramount task.

Two types of time determine the overall TCAM rule-insertion performance: computation time (T_c) and interrupt time (T_i). T_c is the time for computing a rule moving sequence for inserting a rule and T_i is proportional to the number of rule moves, measuring the lookup interrupt time per update. It is easy to see the rule-insertion delay is at best $T_c + T_i$, and the throughput is at most $1/max\{T_c, T_i\}$, overlooking the communication delay and overhead.

It is clear that a better scheme should make both T_c and T_i small, without being biased towards one and neglecting the other. In this paper, we put forward a new scheme, *FastUp*, which presents the following advantages over the state-of-the-art schemes:

- *FastUp* reduces the time and space complexity from $O(m^2)$ and $O(m)$ to $O(m \log h)$ and $O(h)$, respectively, by using a Sequential Stack-based Algorithm (SSA) in place of the predominant *DPA* in the state-of-the-art schemes, where h, the diameter of the rule graph (Section II-B), is much smaller than m, the number of TCAM entries. Experiments show that *FastUp* shortens T_c by 100×, resulting in a 15× reduction in update delay and a 10× increase in update throughput.
- Differing from the existing schemes with the requisite on the location of empty entries (*e.g.*, bottom), *FastUp* allows rules to move in both directions, so the empty entries at any location can be utilized, which not only makes *FastUp* achieve 100% TCAM utilization, but also helps to shorten the interrupt time. Experiments show that *FastUp* shortens the average and maximal T_i by 1.3× and 1.6×, respectively.
- Some earlier schemes ignore the reorder problem. Some other schemes realize it but mistakenly assume it can be resolved by the same *DPA*-based algorithm. We show that *DPA*-based algorithm can induce infinite-loop if used for reorder resolution (Section IV-B2), so *FastUp* uses a dedicated Rule Chain-based Algorithm (RCA) for it instead. RCA exhibits better performance than the other solutions by allowing bi-directional moves.

Furthermore, some works [4], [42], [48] claim they realize the Optimal TCAM Update (OTU), *i.e.*, achieving the theoretically shortest T_i. We refute the assertion by a simple counterexample and then explain the essential reason. For the first time, we propose a practical Branch-and-Bound Algorithm (BBA) to acquire OTU for a relatively large TCAM, which can be used to evaluate the degree of optimality for any fast TCAM update scheme. Our evaluations show that *FastUp* achieves 90% optimality.

The rest of the paper is organized as follows. Section II provides the background. Section III summarizes the related work. Section IV discusses two key algorithms *SSA* and *RCA* of *FastUp*. Section V gives the theoretical analysis of OTU problem and proposes *BBA*. Section VI reports the performance evaluation results. Section VII concludes the paper.

II. Background

A. Flow Table and TCAM

A rule $r = (pri, match, action)$ in a flow table R comprises three parts. $r.pri$ is an integer representing r's priority where a larger value means a higher priority. $r.match$ defines a set of packet header prefixes or ranges for matching. $r.action$ indicates the action on packets that match $r.match$. Rules in a flow table may overlap (*i.e.*, $r_i.match \cap r_j.match \neq \varnothing$), so a packet may match multiple rules. Such cases are resolved by taking the matched rule with the highest priority.

Fig. 2(a) shows a flow table with 6 rules $\{r_0 \sim r_5\}$, and r_6 is a new rule to be inserted. Their overlapping relationship is shown in Fig. 2(b). The packet p with the field values {0111, 1011} matches r_2, r_4, and r_5 but r_2 is taken as the best.

As shown in Fig. 2(c), $\{r_0 \sim r_5\}$ are "programmed" in a match table in TCAM and an associated action table in SRAM. When multiple rules are matched simultaneously by a packet, TCAM can only return the action associated with the matched rule at the lowest memory address. Hence, overlapping rules must be placed in TCAM in descending priority order.

B. Rule Graph and Reorder Problem

The naive way for flow table placement in TCAM is to store all rules in strict priority order, as shown in Fig. 2(c). Thus, the Naive Moving Strategy (NMS) for inserting r_6 to TCAM requires moving downward all the rules $\{r_1 \sim r_5\}$ with a lower priority than r_6 by one entry.

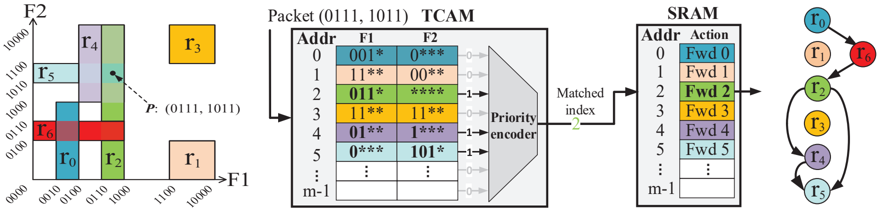
Differently, inserting r_6 to the second entry and moving r_1 downward to the seventh entry also guarantees the correctness of packet matching, but with only one rule move. Because r_1 does not overlap with $\{r_2 \sim r_5\}$, the relative position between r_1 and $\{r_2 \sim r_5\}$ is insignificant. In other words, non-overlapping rules are not required to be kept in strict priority order. It is sufficient to maintain the correct relative position between only overlapping rules [42], which can be formulated by the following *Condition I*.

$$\forall\, r_i, r_j \in R,\ if\ r_i \succ r_j,\ r_i.\text{addr} < r_j.\text{addr} \qquad (1)$$

r.addr indicates the position of r in TCAM, *e.g.*, r_0.addr=0 in Fig. 2(c). $r_i \succ r_j$ ($r_j \prec r_i$) means $r_i.match \cap r_j.match \neq \varnothing$

Rule	Pri	Match		Action
		F1	F2	
r_0	9	001*	0***	Fwd 0
r_1	7	11**	00**	Fwd 1
r_2	6	011*	****	Fwd 2
r_3	5	11**	11**	Fwd 3
r_4	2	01**	1***	Fwd 4
r_5	1	0***	101*	Fwd 5
r_6	8	0***	010*	Fwd 6

(a) Flow table.



(b) Rule overlapping.

(c) Implementation of the flow table in TCAM and SRAM.

(d) DAG.

Fig. 2. An example of flow table with 6 rules $\{r_0 \sim r_5\}$ and a new rule r_6 to be inserted.

and $r_i.pri > r_j.pri$. r_i is called an ascendant of r_j, and r_j a descendant of r_i. Condition I dictates that each rule must be placed above all its descendants and below all its ascendants.

Based on this, rule graph [41] is widely used to guide the TCAM flow table updates, where each node represents a rule and each directed edge represents the overlapping relationship of two rules. An edge is directed from an ascendant node to a descendant node, which makes the rule graph a Directed Acyclic Graph (DAG). The rule graph for Fig. 2(a) is shown in Fig. 2(d). Condition I can be expressed equivalently as that any rule placement scheme according to the topological order of the rule graph is feasible and vice versa.

In Fig. 2(d), before r_6 is inserted, r_0 and r_2 are non-overlapping. It is fine to place r_0 below r_2 in TCAM without violating Condition I. However, if r_6 is to be inserted, which happens to be a descendant of r_0 and an ascendant of r_2, r_0 must be moved above r_2 to maintain the proper topological order. Such a *reorder problem*, which is specific to the rule graph-based schemes, must be detected and resolved first.

III. Related Works

Due to its high hardware cost and power consumption [49], TCAM's capacity is limited and it may fail to accommodate a large-scale rule table. Some researchers tackle this issue by compacting the rule tables [50], [51]. Others use TCAM as a cache instead [5], [7]–[14]. However, these works [5], [7], [8], [10], [52] focus on the selection of caching contents but overlook the potential performance impact of TCAM updates due to cache refreshment. Ding *et al*. take the number of rule replacements into consideration when selecting rules to cache, but the measures negatively affect the TCAM hit-rate [11]. Li *et al*. avoid the TCAM update problem when refreshing TCAM. The consequence is that it requires every packet to be processed by software regardless of the matching status in TCAM, nullifying the purpose of the cache [13]. Since using TCAM as a cache requires frequent cache refreshment in order to maintain the high TCAM hit-rate, a fast TCAM update scheme is crucial.

Some works optimize the update process by reducing the redundant updates at the controller level [53]–[56]; most schemes are designed for individual updates [4], [39], [41]–[44]. The existing TCAM update schemes can be classified based on their technical features.

Among these schemes, the earlier works adopt the priority grouping method featuring a small T_c. PLO [41] groups LPM rules based on the prefix length. FFU [39] groups general rules based on their topology order. The worst-case T_i for these schemes subjects to the number of groups, which is usually between tens and hundreds.

To shorten the T_i, later schemes start to use each rule's priority value and its position in the rule graph in lieu of the priority grouping. A clear trade-off on selecting the candidate rules to move distinguishes these schemes. Given the update rule r, *Cao* [41] and Γ_{down} only consider a single candidate (*i.e.*, r's closest ascendant or descendant) in each recursive step. On the other extreme, Γ_{bh} and *RuleTris* evaluate all the feasible candidates (*i.e.*, the rules between r and r's closest ascendant or descendant) at each step, which leads to a near-optimal T_i at a high T_c. *FastRule* gives up some T_i gain in exchange of a smaller T_c. Only for the new rule, are all the candidates considered for *FastRule*; in each recursive step, *FastRule* regresses to the single candidate approach as in Γ_{down} and *Cao*.

Cao, Γ_{down}, and Γ_{bh} assume that all the empty entries are at bottom, so the middle empty entries caused by rule deletions cannot be used and the expensive and limited TCAM resource cannot be utilized fully. *FastRule* solves this problem but is at the cost of rule moves when deleting rules. *Cao*, *RuleTris*, and *FastRule* ignore the reorder problem. Γ_{down} and Γ_{bh} notice the reordering problem, however, their reorder resolution not only requires excessive rule moves, but in some cases does not work (Section IV-B).

A few works tackle the batch TCAM update problem. CoPTUA [40] focuses on maintaining table consistency and lookup throughput during batch updates. The method actually increases the T_i. The complex table management and batch processing also increase the update latency. Hermes [20] uses a small logical shadow table to process batches of updates and migrates the changes to TCAM periodically. As a system architecture, Hermes lacks an underlying batch update scheme. COLA [45] relies on the individual update schemes mentioned above to calculate the moving sequence for each rule in a batch first and then jointly considers the final TCAM placement. While its T_i is improved, its T_c is still additive and subjects to the poor T_c of the individual updates. COLA adopts Γ_{down} as its reorder solution and solves the inefficiency of Γ_{down} in terms of TCAM utilization at the cost of extra rule moves.

TABLE I
PARAMETER DEFINITION.

Symbol	Description
m, n	The number of TCAM entries and flow table rules
T[i], $0 \leq i < m$	The i-th TCAM entry. T[0] is the TCAM top
$R = \{r_i \| 0 \leq i < n\}$	The flow table composed of n rules r_0, r_1,...,r_{n-1}
$r_i \succ r_j$, $r_j \prec r_i$	r_i is the ascendant of r_j, r_j is the descendant of r_i
$G = (V, E)$	$\forall\, r \in R$, $r \in V$; $\forall\, r_i \succ r_j$, $(r_i, r_j) \in E$
h	the diameter of the rule graph $G = (V, E)$
Des(i), Asc(i)	The descendants and ascendants of the rule in T[i]
r_u	The new rule to be inserted
Des(r_u),Asc(r_u)	The set of descendants and ascendants of r_u
r.addr	The address of the entry in which r is placed
succ(i), succ(r_u)	The successor of the rule in T[i], and r_u. Among all the rules in Des(i), and Des(r_u), the uppermost one is the successor.
pred(i), pred(r_u)	The predecessor of the rule in T[i], and r_u. Among all the rules in Asc(i), and Asc(r_u), the lowermost one is the predecessor.
$\mathbb{S}$	A stack implemented with continuous memory and containing entry addresses. $\mathbb{S}[0]$ is the bottom item.

Therefore, an efficient individual TCAM update scheme is still the prerequisite for those batch TCAM update schemes.

IV. DESIGN AND ANALYSIS OF *FastUp*

The high-level workflow of *FastUp* is shown in Fig. 3. When a new rule needs to be inserted, if it causes the reorder problem, *FastUp* first applies *RCA* to resolve it. Then, *FastUp* applies *SSA* to compute a rule moving and insertion sequence. At last, it interrupts the TCAM lookup and applies the sequence.

A. Algorithm SSA

*1) **Formulation of Moving Cost:*** Table I lists some notations we use for the algorithm description. For ease of description, we assume that rules are populated at the top of TCAM and empty entries are at the bottom, as shown in Fig. 4(a). So the direction of rule moving is downward only. Note that *FastUp* also works when empty bubbles exist between occupied entries, which in fact lead to faster solutions. Without loss of generality, this paper considers the more difficult case where no bubble exists between occupied entries.

To insert a new rule r_u to TCAM, Condition I implies that r_u can be placed in any entry that is below r_u's predecessor (*i.e.*, the ascendant with the highest address) and above r_u's successor (*i.e.*, the descendant with the lowest address). Entries in this range (including r_u's successor) are considered as *candidate locations* for placing r_u. For example, r_6 in Fig. 4(a) has two candidate locations: T[1] and T[2]. Placing a rule in an occupied location causes a chain effect. The original rule in the entry has to be kicked out and relocated in another entry with a higher address, which may in turn kick out another rule. According to [4], inserting r_6 to a non-candidate location will cause more rule moves. Therefore, we only need to evaluate the minimum moving cost C[i] (*i.e.*, the number of rule moves) for each candidate location of T[i] for inserting r_u. Clearly, the T[i] with the smallest C[i] should be taken as the target of r_u.

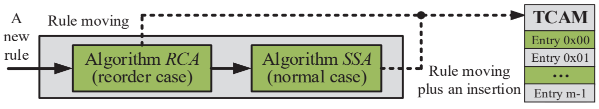
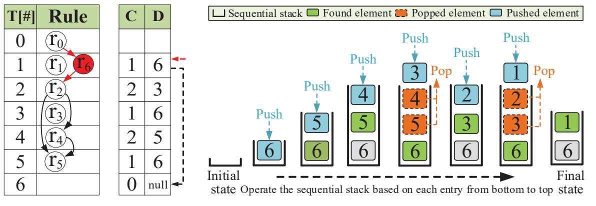


Fig. 3. The workflow of *FastUp*.



(a) DAG. (b) *DPA*. (c) *SSA*.

Fig. 4. Comparison between two cost-based approaches to insert a new rule.

To relocate the rule in T[i], according to Condition I, any entry within the available range (i, succ(i).addr] can be considered as a candidate location. For example, in Fig. 4(a), since succ(2) = r_4, T[3] and T[4] are candidates for the rule in T[2]. According to C[i]'s definition, C[i] equals the smallest moving cost among all its candidates plus 1, *e.g.*, C[2]=*min*{C[3],C[4]}+1. The idea is formulated in Equation 2.

$$\mathrm{C}[i] = \min_{j \in (i,\, \mathrm{succ}(i).\mathrm{addr}]} \{\mathrm{C}[j]\} + 1 \qquad (2)$$

Meanwhile, we introduce an assistant array D[·] for moving sequence inference. D[i] = j' records the best-candidate T[j'] that contributes to the smallest C[i]. After the computation, we can access this array recursively to assemble the moving sequence for placing r_u in T[i].

It is easy to see from Equation 2 that the cost evaluation is a typical dynamic programming process. An entry's moving cost is determined by the moving cost of each entry in its available range. For example, C[2] can be calculated after C[3] and C[4] are given. The cost calculation can be tackled by using *DPA* as shown in Fig. 4(b). To calculate C[1] and C[2] for the insertion of r_6, *DPA* starts to compute C[6] from the bottom entry of TCAM, and proceeds upward until C[1] and C[2] are calculated. After that, *DPA* chooses the entry in r_6's available range with the smallest moving cost. Since C[1] < C[2], T[1] is chosen as the best-candidate to place r_6. The final moving sequence, $r_6 \to$ T[1] $\to$ T[6], as shown by the dotted line in Fig. 4(b), can be inferred from D[·].

DPA needs to process up to m entries. For each entry, finding its best-candidate needs $O(m)$ comparisons. Therefore, the time complexity of *DPA* is $O(m^2)$. Due to the introduction of C[·] and D[·], *DPA* consumes $O(m)$ memory.

*2) **SSA Basis:*** The high time complexity of *DPA* makes the schemes based on it, such as *RuleTris* and Γ_{bh}, less suitable for practical applications. The following Theorem 1 provides us an opportunity to optimize the algorithm.

Theorem 1. During the moving cost calculation from the bottom upward for all entries according to Equation 2, after C[i] and C[j] are calculated, if $j > i$ and C[j] $\geq$ C[i], T[j] can be safely excluded from the set of the best-candidates in the subsequent calculation process.

Proof. Suppose C[j] and C[i] have been calculated and C[x] is now being calculated. We need to find the best-candidate for the rule in T[x]. If T[j] is not a candidate for the rule in T[x] at all, we can simply ignore it. Otherwise, since $i < j$, T[i] must be a candidate for the rule in T[x] as well. Since C[j] $\geq$ C[i], according to Equation 2, ignoring T[j] has no effect on C[x]. Hence, Theorem 1 is proved. ■

On the surface it appears that, even if the useless candidates are ignored, the time and space complexities for calculating the moving cost on m entries are still $O(m^2)$ and $O(m)$. In terms of the time complexity, even if we search only among the entries that are potentially to be the best-candidates for any T[i], it still takes $O(m)$ comparisons. Meanwhile, after C[i] has been calculated, according to Theorem 1, some entries are no longer needed to be taken into consideration in the subsequent computation process and should be ignored, which also takes $O(m)$ time to identify and remove such entries. In terms of the space complexity, in addition to C[$\cdot$] and D[$\cdot$], recording such entries consumes $O(m)$ extra space.

Fortunately, *SSA* adopts a ***sequential stack*** to record those entries that have the potential to be chosen as the best-candidate in the subsequent calculation process, which leads to a much lower time and space complexity than *DPA*. A sequential stack $\mathbb{S}$ is a stack implemented with continuous memory (*e.g.*, array). $\mathbb{S}$[0] indicates the bottom item.

When processing T[i], *SSA* searches for its best-candidate in $\mathbb{S}$ with which C[i] can be calculated. After that *SSA* pops some entries from $\mathbb{S}$ according to Theorem 1 and then pushes i into $\mathbb{S}$. Proposition 1 states three elegant properties of $\mathbb{S}$.

Proposition 1. $\mathbb{S}$ has the following three elegant properties: I. $\mathbb{S}[p] > \mathbb{S}[p\text{+}1]$; II. C[$\mathbb{S}[p]$] $= p$; and III. Sizeof($\mathbb{S}$) $\leq h+1$.

Proof. Any address i can only be pushed into $\mathbb{S}$ once, so each element of $\mathbb{S}$ is unique. i is pushed into $\mathbb{S}$ only after all the higher addresses have been processed. Popping addresses from $\mathbb{S}$ does not change the order of the remaining elements. Hence, the elements in $\mathbb{S}$ are strictly increasing from top to bottom, and Property I is proved.

Property II can be proved by mathematical induction. Initially, $\mathbb{S}$ is empty and the calculation starts from the empty entry T[m-1] at the bottom of TCAM. Clearly, C[m-1] = 0 and D[m-1] = *null*. After pushing m-1 to $\mathbb{S}$, we get $\mathbb{S}$[0] = m-1 and C[$\mathbb{S}$[0]] = 0, which proves the base case. Suppose Property II holds until calculating C[i] and T[$\mathbb{S}[q]$] is found to be the best-candidate for the rule in T[i]. We have,

$$\text{C}[i] = \text{C}[\mathbb{S}[q]]\text{+}1 = q\text{+}1,\ \text{D}[i] = \mathbb{S}[q]$$

According to Theorem 1, before i is pushed into $\mathbb{S}$, some elements $\mathbb{S}[p]$ that meet the following inequality should be removed,

$$\text{C}[\mathbb{S}[p]] \geq \text{C}[i]$$

From this inequality, it is easy to deduce that $p \geq q$+1, which means all of the elements above the one we just found should be popped. Since the elements above $\mathbb{S}[q]$ are popped before i is pushed, i is placed at $\mathbb{S}[q\text{+}1]$. It follows that,

$$\text{C}[\mathbb{S}[q\text{+}1]] = q\text{+}1$$

Hence, Property II is proved.

According to the definition of C[i] and h, C[i] is at most h for any i. Property II ensures that the index of the top item in $\mathbb{S}$ is at most h, so the size of $\mathbb{S}$ is at most $h+1$. Hence, Property III is proved. ■

Since T[i]'s candidates are sequentially stored in $\mathbb{S}$, and the one closest to the bottom of $\mathbb{S}$ is exactly T[i]'s best-candidate, a binary search in $\mathbb{S}$, which takes $O(\log h)$ time, is sufficient to find it. After finding the best-candidate, all the elements above it are popped from $\mathbb{S}$. These operations can be done by simply resetting the size of $\mathbb{S}$, which takes $O(1)$ time. In summary, *SSA* takes only $O(m \log h)$ time to calculate the moving cost for m entries.

The usage of $\mathbb{S}$ is more than reducing the computation time. Due to the following propositions, $\mathbb{S}$ eliminates the need of both C[$\cdot$] and D[$\cdot$], so the space complexity is also reduced.

Proposition 2. For any address i in $\mathbb{S}$, $\mathbb{S}$ records a moving sequence to relocate the rule in T[i], which is identical to the one that can be derived from D[i]. Specifically, if i is placed at $\mathbb{S}[x]$, the corresponding moving sequence is,

$$\text{T}[\mathbb{S}[x]] \rightarrow \text{T}[\mathbb{S}[x\text{-}1]] \rightarrow ... \rightarrow \text{T}[\mathbb{S}[0]]$$

Proof. This can be proved by mathematical induction. After the bottom empty entry T[m-1] is processed, $\mathbb{S}$[0]=m-1, which sets up the base case. Suppose Proposition 2 holds before calculating C[i], and T[$\mathbb{S}[q]$] is found to be the best-candidate for the rule in T[i]. According to Property II, the elements above $\mathbb{S}[q]$ are popped from $\mathbb{S}$ and i is placed at $\mathbb{S}[q\text{+}1]$. Now, the best-candidate for the rule in T[$\mathbb{S}[q\text{+}1]$] is exactly T[$\mathbb{S}[q]$]. This effectively adds one more step to the moving sequence in $\mathbb{S}[0{:}q]$, and our hypothesis holds for the induction as well. Hence, Proposition 2 is proved. ■

Based on Proposition 2, D[$\cdot$] is not needed if the best-candidate for the new rule is guaranteed to be in $\mathbb{S}$ after the calculation of C[$\cdot$]. Proposition 3 shows this is indeed the case.

Proposition 3. On completion of moving cost calculation for all the candidates of the new rule r_u, the address of best-candidate of r_u must stay in $\mathbb{S}$. When multiple candidates of r_u have the same smallest moving cost, without loss of generality, the uppermost one in TCAM is the best-candidate.

Proof. This can be proved by contradiction. If T[i] is the best-candidate for r_u, according to Proposition 1, i should be placed at $\mathbb{S}$[C[i]]. Suppose at some point i is popped from $\mathbb{S}$ due to the insertion of the address j. It is easy to see $j < i$ and T[j] is also a candidate for r_u. Moreover, j cannot be placed above $\mathbb{S}$[C[i]], so C[j] $\leq$ C[i] must be true according to Proposition 1, which contradicts our assumption. Hence, Proposition 3 is proved. ■

According to Property II, C[$\cdot$] is also no longer needed. The information of it is fully embedded in $\mathbb{S}$. Hence, the space complexity of *SSA* is $O(h)$.

3) ***Example of SSA****:* Fig. 4(c) shows how *SSA* inserts r_6 to TCAM. The corresponding pseudo-code is Line 14∼21 in Algorithm 1.

Similar to *DPA*, *SSA* starts from the bottom empty entry (*i.e.*, T[6]) and ends at r_6's successor, succ(r_6) (*i.e.*, T[0]). In the beginning, $\mathbb{S}$ is cleared (Line 15). Since T[6] is an empty entry, *SSA* directly pushes “6” into $\mathbb{S}$ (Line 15). Then *SSA* processes the next entry T[5]. *SSA* finds the first element not greater than succ(5).addr in $\mathbb{S}$ (Line 17∼18). Since Des(5) is $\varnothing$, succ(5).addr can be viewed as ∞ and $\mathbb{S}$[0] is found. Then, *SSA* pops out all the elements above $\mathbb{S}$[0]. After that, *SSA* pushes “5” into $\mathbb{S}$. T[4] to T[1] are processed similarly.

Finally, *SSA* finds the first element not greater than succ(r_6).addr in $\mathbb{S}$ (Line 20), which is $\mathbb{S}$[1]. The moving sequence to insert r_6 is given by $\mathbb{S}$[0:1] (Line 21). That is, $r_6 \rightarrow$ T[$\mathbb{S}$[1]] $\rightarrow$ T[$\mathbb{S}$[0]], which is identical to the solution calculated by *DPA*.

*4) **Optimization of SSA**:* The above example only describes how *SSA* moves existing rules downward to insert a new rule, based on the assumption that the empty entries are concentrated at the bottom of TCAM. However, in addition to rule insertion, flow table updates also include rule deletion and modification. A rule modification can be decomposed as a rule deletion plus a rule insertion, so here we only need to consider the rule deletion. By nullifying an entry, a rule deletion effectively generates an empty entry.

Since empty entries can spread everywhere, *SSA* uses the method in Algorithm 1 to avoid wasting any empty entry and further reduce the required rule moves. For a new rule, if there exists any empty entry within its candidate locations, *SSA* simply inserts it there (Line 3∼5). If all empty entries are below the candidate locations, *SSA* calls the function DownCostSeq() to find a feasible solution (Line 6∼8). If all empty entries are above the candidate locations, *SSA* calls the function UpCostSeq() (Line 9∼11). UpCostSeq() is almost identical to DownCostSeq(), except that it reverses the search direction. If empty entries exist in both directions, *SSA* calls both functions and picks a better solution (Line 12).

B. Algorithm RCA

*1) **Example of Reorder Case**:* *SSA* assumes that a new rule r_u can always find one or more candidate locations. If succ(r_u).addr $<$ pred(r_u).addr, the reorder problem happens. *FastUp* must first relocate succ(r_u) or pred(r_u) in order to create some candidate locations, which is a prerequisite for *SSA*. Fig. 5 (a) shows an example. The initial layout of the rules $\{r_0 \sim r_5\}$ is feasible according to Condition I. However, the new rule r_6 causes a reorder problem, because $r_6 \succ r_0$ and $r_6 \prec r_3$. To resolve it, *FastUp* needs to either move r_0 downward or r_3 upward until r_0.addr $>$ r_3.addr.

*2) **Why Not DPA**:* Intuitively we can resolve the reorder problem by computing the lowest-cost move sequence for those out-of-order rules using the similar dynamic programming approach as discussed above. Γ_{bh} claims that the *DPA*-based method is not attractive because of the excessively long computation time. Although *SSA* has significantly reduced the computation time, we find such a method is unusable at all, because it can lead to an ***infinite loop*** in some cases.

For example, in Fig. 5 (a), the only empty entry is at the bottom, so r_0 should be moved downward. If the *DPA*-based method is used, the resulting moving sequence is T[0] $\rightarrow$ T[3] $\rightarrow$ T[6]. The TCAM table layout after the moving is shown in Fig. 5(b). Surprisingly, the reorder problem is not eliminated because r_3 is also pushed downward and it is still below r_0. Now the only empty entry is at the top, so we can try to move r_3 upward in the hope of solving the problem. As shown in Fig. 5(b), the resulting moving sequence is T[6] $\rightarrow$ T[3] $\rightarrow$ T[0]. The final layout shown in Fig. 5(c) is the same as before. This example is sufficient to show that the *DPA*-based method may fail to resolve the reorder problem.

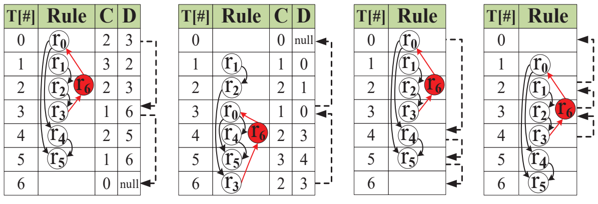
Algorithm 1: *SSA* for Rule Insertion in *FastUp*

```
   Input: G=(V, E), the rule graph of those existing rules;
          $r_u$, the new rule needed to be inserted
   Output: $\mathbb{S}$,sequential stack recording the moving sequence
1  Identify Des($r_u$) and Asc($r_u$) by traversing G=(V, E)
2  $a$ = pred($r_u$).addr, $b$ = succ($r_u$).addr
3  if any empty entry is within the range ($a$, $b$) then
4      T[$start$]: any empty within the available range
5      $\mathbb{S}$.push($start$)
6  else if all empty entries are below the range ($a$, $b$) then
7      T[$start$]: the first empty entry below T[$b$]
8      $\mathbb{S}$ = DownCostSeq($start$, $a$, $b$)
9  else if all empty entries are above the range ($a$, $b$) then
10     T[$start$]: the first empty entry above T[$a$]
11     $\mathbb{S}$ = UpCostSeq($start$, $b$, $a$)
12 else  $\mathbb{S} \leftarrow$ the shorter one between above two stacks
13 return $\mathbb{S}$ // $r_u \rightarrow$ T[$\mathbb{S}$[$end$]] $\rightarrow$ T[$\mathbb{S}$[$end$-1]] $\rightarrow$ ... $\rightarrow$ T[$\mathbb{S}$[0]]
14 Function DownCostSeq($start$, $end$, $mid$)
15     $\mathbb{S}$.clear(), $\mathbb{S}$.push($start$)
16     for ($i = start{-}1$ ; $i > end$ ; $i = i{-}1$) do
17         $key$ = succ($i$).addr
18         $\mathbb{S}$[$q$]: the first element not greater than $key$
19         $\mathbb{S}$.size = $q$+1, $\mathbb{S}$.push($i$)
20     $\mathbb{S}$[$q$]: the first element not greater than $mid$
21     return $\mathbb{S}$[0:$q$]
22 Function UpCostSeq($start$, $end$, $mid$)
23     $\mathbb{S}$.clear(), $\mathbb{S}$.push($start$)
24     for ($i = start{+}1$ ; $i < end$ ; $i = i{+}1$) do
25         $key$ = pred($i$).addr
26         $\mathbb{S}$[$q$]: the first element not less than $key$
27         $\mathbb{S}$.size = $q$+1, $\mathbb{S}$.push($i$)
28     $\mathbb{S}$[$q$]: the first element not less than $mid$
29     return $\mathbb{S}$[0:$q$]
```

*3) **Example of RCA**:* *FastUp* adopts *RCA* described in Algorithm 2 to resolve the reorder problem. It moves either succ(r_u) downward or pred(r_u) upward, while keeping the other one in place, depending on the position of empty TCAM entries. *RCA* ensures the resolution of the reorder problem with randomly distributed empty entries.

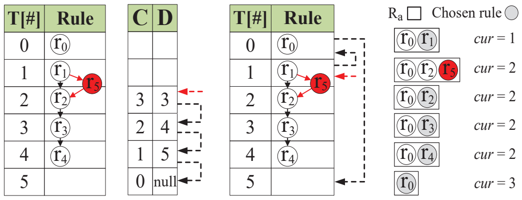
As shown in Fig. 5(c), the only empty entry T[6] is below r_0, *RCA* adopts SuccChain$(0, 6)$ to move r_0 downward. Since no empty entry is above the successor of r_0, r_0 is moved to its successor’s entry. In the following steps, each kicked-out rule is moved to its successor’s entry until a kicked-out rule is settled in an empty entry. The corresponding moving sequence T[0] $\rightarrow$ T[4] $\rightarrow$ T[5] $\rightarrow$ T[6] is shown in Fig. 5(c). Now, r_3 is above r_0, and the reorder problem is resolved.

If the only empty entry T[0] is above r_3, as shown in Fig. 5(d), *RCA* adopts PredChain$(4, 0)$ to move r_3 upward to its



(a) *DPA*. (b) *DPA*. (c) *RCA*. (d) *RCA*.

Fig. 5. Using *RCA* as reorder resolution to avoid infinite loop.



(a) DAG. (b) *DPA*. (c) OTU. (d) *BBA*.

Fig. 6. Comparison between *DPA* and *BBA* to insert a new rule.

Algorithm 2: *RCA* for Reorder Resolution in *FastUp*

```
while succ($r_u$).addr $<$ pred($r_u$).addr do
    if empty entries below T[succ($r_u$).addr] then
        T[$end$]: the first empty entry below T[succ($r_u$).addr]
        $Seq$ = SuccChain(succ($r_u$).addr, $end$)
    else
        T[$end$]: the first empty entry above T[pred($r_u$).addr]
        $Seq$ = PredChain(pred($r_u$).addr, $end$)
    Moving by T[$Seq$[0]] → T[$Seq$[1]] → ... → T[$Seq$[$end$]]
Function SuccChain($start$, $end$)
    $Seq$.push($start$)
    while succ($start$).addr $\leq$ $end$ do
        $start$ = succ($start$).addr
        $Seq$.push($start$)
    $Seq$.push($end$)
    Return $Seq$
Function PredChain($start$, $end$)
    $Seq$.push($start$)
    while pred($start$).addr $\geq$ $end$ do
        $start$ = pred($start$).addr
        $Seq$.push($start$)
    $Seq$.push($end$)
    Return $Seq$
```

predecessor's entry and so forth. The corresponding moving sequence T[4] → T[3] → T[2] → T[0] is shown in Fig. 5(d). However, after the move, r_3 is still below r_0. They become closer but the reorder problem persists. In such a case, *RCA* repeats the procedure of moving r_0 downward or r_3 upward until the problem is resolved. Since the only empty entry T[4] is below r_0, *RCA* calls SuccChain$(1, 4)$ to move r_0 downward to T[4]. The reorder problem is resolved.

4) ***Analysis of RCA:*** When moving succ(r_u) downward according to *RCA*, it is impossible to kick out pred(r_u), because pred(r_u) is not on the chain of the successor starting from succ(r_u). Similarly, when pred(r_u) is moved upward, succ(r_u) is kept in place. Since *RCA* always moves one rule (succ(r_u) or pred(r_u)) while keeping the other in place, in one round, if the reorder is not resolved, the gap between the rules is reduced. By repeating the recursive procedure, the reorder is guaranteed to be resolved.

V. Optimality Analysis of TCAM Update

Some previous *DPA*-based schemes claim they achieve OTU [4], [42], [48]. We can easily come up with a counterexample to show that the dynamic programming fails to find the optimal solution. For the rule update shown in Fig. 6(a), the result of *DPA* is shown in Fig. 6(b), which requires three rule moves. However, as shown in Fig. 6(c), the OTU solution requires only two rule moves, by allowing rules to move in both directions.

A. Analysis of OTU

Condition I establishes that any topological order of the rule graph is a feasible TCAM placement scheme and vice versa. If n rules are inserted into m TCAM entries ($m \geq n$), the number of feasible TCAM layouts N_L is:

$$N_L = N_{\boldsymbol{\pi}} * \binom{n}{m} = N_{\boldsymbol{\pi}} * \frac{m!}{n!(m-n)!} \qquad (3)$$

where $N_{\boldsymbol{\pi}}$ is the number of topological orders of the n rules.

For any feasible TCAM layout L, the number of required TCAM operations C_L equals to the number of TCAM entries that change their content (*i.e.*, the rule in an entry is cleared or changed to another one, or an empty entry is filled with a rule). Hence, OTU can be formulated as follows.

Formulation of OTU: To insert a new rule to TCAM, among all the N_L feasible TCAM layouts, find the one requiring the minimum number of TCAM entry changes. ■

It is infeasible to conduct brute-force search in such a large space. In fact, just finding the number of all topological orders $N_{\boldsymbol{\pi}}$ has been proven to be NP-hard. To the best of our knowledge, no existing scheme can solve the OTU problem.

B. Algorithm BBA for Small Scale OTU

It is interesting to understand how close a practical scheme such as *FastUp* is to the optimality. The degree of optimality, λ, for a Design Under Test (DUT) is defined as,

$$\lambda_{\text{DUT}} = \frac{N_{\text{OTU}}}{N_{\text{DUT}}} \times 100\% \qquad (4)$$

N_{DUT} and N_{OTU} are the number of TCAM operations for the DUT and OTU to insert a rule. λ can be used to guide further algorithm optimizations.

The naive brute-force search for OTU can take hours for even n=m=20, as the complexity is $O(m!/(m-n)!)$. Instead, we propose *BBA*, which is capable of searching for OTU in just tens of minutes when $n \leq m = 1000$.

An example is illustrated in Fig. 6(d). *BBA* processes each entry from top to bottom. During the process, for any entry T[i], R_a records any rule r if it has not been placed yet but the rules in Asc(r) have been placed in entries above

Algorithm 3: *BBA* for Finding OTU

```
   Input: G=(V, E), the rule graph of those existing rules;
          $r_u$, the new rule needed to be inserted
   Output: opt, the number of rule moves for OTU
1  construct $G' = (V', E')$ by adding $r_u$ to $G = (V, E)$
2  $R_a$=∅, opt=∞, cur=0, addr=0 // initialization
3  for each $r \in V'$ do
4      if Asc($r$)==∅ then
5          $R_a = R_a + r$
6  PerEntryProcess($R_a$, cur, addr)
7  return opt
8  Function PerEntryProcess($R_a$, cur, addr)
9      if addr == $m$ and cur < opt and $R_a$==∅ then
10         opt = cur
11     else if cur ≥ opt or addr ≥ $m$ then
12         return
13     else //R[addr]: the rule originally placed in T[addr]
14         if R[addr] ∈ $R_a$ then
15             //{$r'$}: available rules after R[addr] is settled
16             PerEntryProcess($R_a$-R[addr]+{$r'$},cur,addr+1)
17         for each $r \in R_a$ and $r \neq$ R[addr] do
18             //{$r'$}:newly available rules after $r$ is settled
19             PerEntryProcess($R_a - r + \{r'\}$,cur+1,addr+1)
20         if T[addr] is originally empty then inc=0 else inc=1
21         PerEntryProcess($R_a$, cur+inc, addr+1)
```

T[i]. According to Condition I, these rules are all eligible candidates to be placed in T[i] (*e.g.*, R_a={r_0, r_1} for T[0]). So *BBA* tries to place each rule of R_a in T[i] (Line 14∼19), or leave T[i] empty (Line 20∼21). cur and opt represent the cumulative rule moves for the currently searching process and the rule moves for currently found optimal solution, which are initialized to 0 and ∞, respectively (Line 2). Since r_1 is not originally in T[0], cur should be increased by one. Meanwhile, r_1 is temporarily settled so it is removed from R_a. The settlement of r_1 makes r_2 and r_5 available for the entries below T[0], so they are added into R_a (Line 18∼19).

Each of the following entries is processed sequentially. During the searching process, only if cur is less than opt, will the process continue (Line 11∼12), so *BBA* can avoid searching in the space where OTU cannot exist, which significantly speeds up the searching process. If all empties are processed and a better solution is found, opt is updated (Line 9∼10). When trying to place each rule of R_a in T[i], *BBA* gives priority to the rule originally placed in T[i], which helps to find the smallest opt faster (Line 14∼16). The result of *BBA* in Fig. 6(d) is the same as OTU shown in Fig. 6(c).

VI. Implementation and Evaluation

A. Experimental Setup

We compare *FastUp* with *RuleTris* and Γ_{bh}, because these schemes achieve the shortest T_i so far. We also compare with Γ_{down}, because it notices the reorder problem and represents the works that are in favor of shortening T_c at the cost of longer T_i. We implement them in C++ and extend the firmware on ONetSwitch [57], a programmable OpenFlow switch with an 800Mhz Cortex-A9 CPU and 512MB DDR3 RAM. To ensure fair comparison, different schemes use the same rule graph implementation. We use the *SDNet* Development Environment [58] to configure the TCAM flow tables. ONetSwitch's TCAM supports at most 4,096 entries, but its firmware can support software-based flow tables with arbitrary size.

TABLE II
An insight into flow table and the corresponding rule graph.

Type	ACL									
$\boldsymbol{S_1}(k)$	1.0	1.8	2.8	3.6	4.5	5.5	6.5	7.5	8.5	9.5
$\boldsymbol{S_2}(k)$	2.0	3.4	5.6	7.4	9.5	11.4	13.1	15.4	17.3	18.6
$\boldsymbol{h}$	53	75	93	106	119	125	129	134	140	146

Type	FW									
$\boldsymbol{S_1}(k)$	0.8	1.7	2.7	3.7	4.7	5.7	6.6	7.5	8.4	9.4
$\boldsymbol{S_2}(k)$	3	6	9	13	16	19	22	24	27	33
$\boldsymbol{h}$	33	35	45	48	54	59	61	65	71	78

To test different schemes on large flow tables, we first run them on the ONetSwitch's CPU to calculate the number of rule moves, p, for an insertion update. Both rule move and insertion use the same API function provided by *SDNet* to access TCAM, which takes about 0.6ms. We execute it p times to get an accurate measurement of T_i. Each experiment is repeated 10 times and the average is taken.

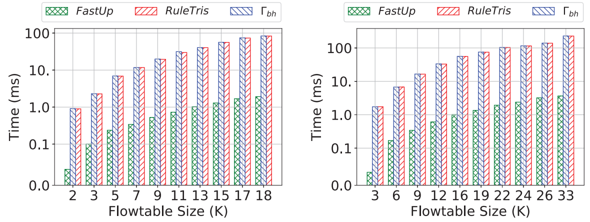
B. Update Mode and TCAM Table Layout

For fair comparisons, given a flow table, we randomly choose a subset of rules as base rules to be pre-installed in TCAM and then take the remaining rules as updates. We evaluate two update modes: Virtual Insertion (VI) and Continuous Insertion (CI). VI runs the schemes and evaluates the TCAM operation times for each update, but it does not actually insert the rule into TCAM. Since different schemes may produce different moving sequences for a new rule and result in different TCAM table layouts, VI guarantees an identical comparison basis for each scheme for each update. In contrast, CI runs the schemes and conducts the actual TCAM operations, which can reflect the accumulative result.

Due to the limitations of some previous schemes, we need to consider the initial TCAM table layout. Γ_{down} and Γ_{bh} can only use the empty entries below the candidate locations of a new rule, so the initial layout keeps the empty entries at the bottom. *RuleTris* cannot handle the reorder problem, so the base rules are arranged in strict priority order to avoid it. The subsequent updates in CI mode may introduce the reorder problem at some point, causing the failure of *RuleTris*.

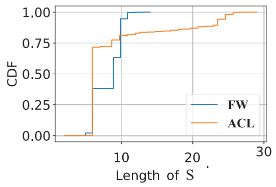
C. Dataset

It is difficult to acquire large-scale real-world rule sets due to privacy concerns. Alternatively, synthetic rule sets bearing the characteristics of the real-world rule sets are widely adopted for scheme evaluation [59]. We use two representative types of flow tables, Access Control List (ACL) and Firewall (FW), generated by ClassBench [59]. Each rule contains five IP header matching fields. The rules that cannot be directly stored in TCAM due to the range-based matching fields(*e.g.*, source



(a) ACL (b) FW

Fig. 7. Time consumption in the worst case.



Type	S_2	h	l_{avg}	l_{max}
ACL	$18k$	146	6.2	29
FW	$33k$	78	8.2	14

Fig. 8. Statistical characteristics of $\mathbb{S}$'s length when S_1 is $10k$.

TABLE III
INTERRUPT TIME, RANDOMLY DISTRIBUTED EMPTY ENTRIES, ACL.

$S_2(k)$	#Updates	Average time (ms)			Maximal time (ms)		
		FastUp	Γ_{bh}	Γ_{down}	*FastUp*	Γ_{bh}	Γ_{down}
3.4	340	0.62	0.65	2.31	1.2	1.8	6.6
7.4	740	0.64	0.78	3.16	2.4	3.6	10.8
11.4	1140	0.65	0.85	4.23	2.4	4.2	10.8
15.4	1540	0.70	0.86	4.57	3	4.8	11.4
18.6	1860	0.72	0.95	4.78	3.6	6	18

and destination ports) are converted by ClassBench-ng [60] into a set of prefix-based rules first.

The characteristics of the flow tables are summarized in Table II. S_1 and S_2 are the sizes of the original and the converted rule tables, respectively. The diameter of the rule graph, h, largely determines the performance of *FastUp*. h is roughly two orders of magnitude smaller than S_2. For example, the values of h are only 146 and 78 when S_2 is $18.6k$ and $33k$ for ACL and FW, respectively. While h represents the upper bound of the size of $\mathbb{S}$, experiments show that the actual size of $\mathbb{S}$ is much smaller than h.

D. Experimental Results

*1) **The Advantage of** $\mathbb{S}$:* To emulate a worst-case scenario, we place all flow table rules in TCAM as base rules in priority order and leave empty entries only at the bottom of TCAM. Fig. 7 shows the time taken by each scheme to calculate the moving cost for all entries, which represents the worst-case T_c. *FastUp* is about 40× to 100× better than *RuleTris* and Γ_{bh}. The T_c performance gain becomes greater as S_2 increases.

The reasons for the gain can be explained by Fig. 8. For up to 80% of cases for ACL and 95% for FW, $\mathbb{S}$'s length is below 10. The average length of $\mathbb{S}$ is only 6.2 and 8.2 for ACL and FW, respectively. The worst-case length of $\mathbb{S}$ is 29, which requires only $\lceil \log_2 29 \rceil$=5 comparisons for *FastUp* to calculate an entry's moving cost. In contrast, *RuleTris* or Γ_{bh} takes $7.7k$ and $12k$ comparisons for ACL and FW when S_2 is $18k$ and $33k$, respectively.

*2) **TCAM Utilization**:* We first pre-install 10% rules to the top of TCAM in priority order, and then continually insert (in CI mode) the remaining rules to TCAM as updates until the scheme cannot find a solution any more. At this point, we consider the fill ratio of the TCAM as its utilization ratio. We exclude *RuleTris* because it cannot handle the reorder problem.

In this experiment we only conduct rule insertions. Although the empty entries concentrate at the bottom in the beginning, they may be distributed anywhere after reorder resolution. The experiments show that Γ_{bh} achieves at most 95% utilization ratio while *FastUp* can always make full use of TCAM. Note that rule deletions will also generate random empty entries in TCAM, which can worsen the utilization ratio of Γ_{bh}.

*3) **Computation and Interrupt Time**:* Unless otherwise specified, the initial TCAM table layout follows the strict priority order. We conduct the experiments in VI mode so *RuleTris* and Γ_{bh} are immune to the reorder problem. We increase the proportion of the base rules to 90% to measure the scheme performance under severe conditions.

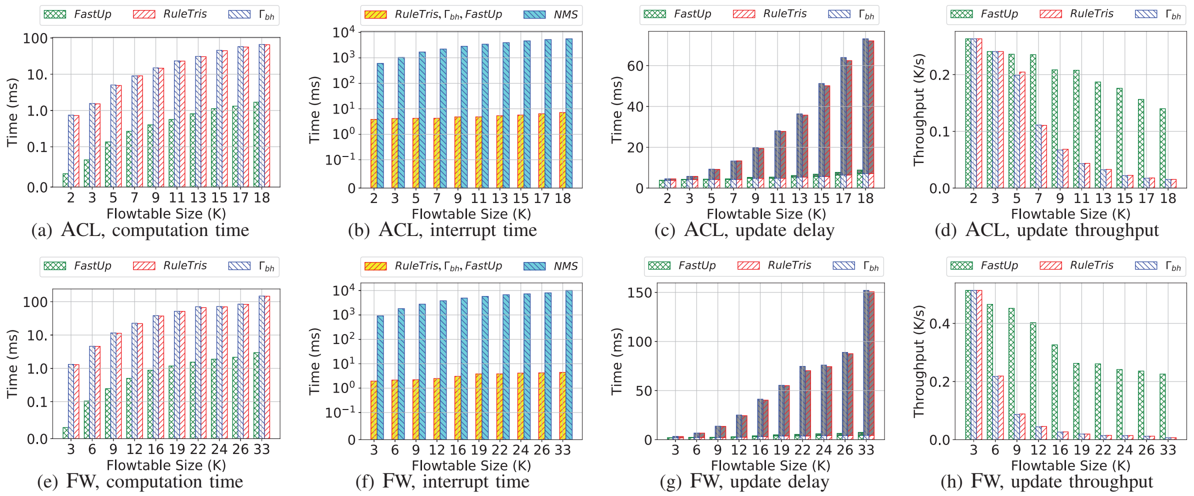
Fig. 9 (a) and (e) show the T_c for per rule insertion. While T_c grows with the increase of S_2, *FastUp* remains about two orders of magnitude better than *RuleTris* and Γ_{bh}. When S_2 is $33k$, *FastUp* takes only 2ms to process a new rule while *RuleTris* and Γ_{bh} take more than 200ms.

To illustrate the benefit of the cost-based schemes on T_i, we compare these schemes with *NMS*. As shown in Fig. 9 (b) and (f), the cost-based schemes shorten the T_i of *NMS* by up to 1000× and need less than 5ms for a new rule insertion. For these schemes, T_i is insensitive to the table sizes.

We relax the initial rule layout by allowing random distribution of empty entries in TCAM to show the advantage of *FastUp* over the other schemes in terms of T_i. Due to limited space, we only show the results on ACL in Table III. We draw three conclusions. First, *FastUp* and Γ_{bh} are much better than Γ_{down}. The T_i of Γ_{down} can be 18ms, which may degrade the packet forwarding performance. Since T_c is usually much shorter than T_i for *FastUp*, it makes little sense to optimize T_c at the cost of T_i like Γ_{down}. Second, the difference between the results in Fig. 9 (b) and Table III shows that the randomly distributed empty entries help to shorten T_i. Third, for large tables, *FastUp* is 1.3× and 1.6× better than Γ_{bh} for the average and maximal T_i, respectively. This is because *FastUp* can utilize any empty entry while Γ_{bh} can only use the empty entries in or below the candidate locations of the new rules.

*4) **Update Delay and Throughput**:* In Fig. 9 (c) and (g), the shaded portion of the bar represents T_c and the remaining portion represents T_i. The update delay of *RuleTris* and Γ_{bh} increases significantly with the increase of S_2, which is largely due to the increase of T_c. In contrast, *FastUp* has a small update delay regardless of S_2. Its T_i is the dominant factor and remains relatively stable. The update delay of *FastUp* is 15× shorter than that of *RuleTris* and Γ_{bh} when S_2 is $33k$.

Fig. 9 (d) and (h) compare the update throughput. When we pipeline the computation and interrupt processes, the update throughput is determined by the process that takes a longer time. When S_2 is small, T_i is dominant, so they all exhibit similar throughput performance. With the increase of S_2, the



(a) ACL, computation time (b) ACL, interrupt time (c) ACL, update delay (d) ACL, update throughput

(e) FW, computation time (f) FW, interrupt time (g) FW, update delay (h) FW, update throughput

Fig. 9. Experimental results on VI mode.

TABLE IV
REORDER EFFICIENCY, RANDOMLY DISTRIBUTED EMPTY ENTRIES, ACL.

$\boldsymbol{S_2}(k)$	**#Updates**	**#Reorder**	**#Resolved**		**Average time**(ms)	
			FastUp	Γ_{down}, Γ_{bh}	*FastUp*	Γ_{down}, Γ_{bh}
3.4	340	1	1	1	13.10	16.20
7.4	740	5	5	5	19.72	21.96
11.4	1140	9	9	8	20.52	23.79
15.4	1540	14	14	13	20.70	24.17
18.6	1860	18	18	16	21.34	25.47

TABLE V
COMPARISON OF THE INTERRUPT TIME BETWEEN *FastUp* AND *BBA*.

Case	**Probability**	**Interrupt time** (ms)					
		FastUp		*BBA*		λ_{FastUp}	
		Avg	Max	Avg	Max	Avg	Max
Normal	97.23%	1.29	4.2	1.24	3.0	96%	71%
Reorder	2.77%	5.20	12.6	2.30	3.0	44%	23%
Mixed	100%	1.40	12.6	1.27	3.0	90%	23%

T_c of *RuleTris* and Γ_{bh} becomes dominant, but the T_c of *FastUp* remains short. Therefore, *FastUp* eventually shows a 10× higher throughput than *RuleTris* and Γ_{bh} for a large S_2.

The update delay and throughput truly reflect a scheme's performance, which rely on both T_c and T_i. Unlike the previous schemes, *FastUp* strives to excel at both. As a result, *FastUp achieves the shortest* T_i *among all the schemes while keeping* T_c *shorter than* T_i.

*5) **Reorder Resolution**:* The pre-installed rules are placed in topological order and empty entries are randomly distributed in TCAM. We examine the new rules that incur the reorder problem and show the efficiency of different schemes in resolving the problem. Due to limited space, we only show the results on ACL in Table IV. We draw two conclusions. First, the probability of reorder occurrence is small. Second, *FastUp* outperforms Γ_{down} and Γ_{bh} in resolving the reorder problem. *FastUp* guarantees to resolve the reorder problem as long as TCAM is not completely full.

*6) **Degree of Optimality**:* We evaluate how close *FastUp* is to the optimal bound calculated by *BBA* using FW with S_2 of 1000 in VI mode. The result shown in Table V is the average of ten simulation runs with different initial table layouts. While handling the normal cases well, *FastUp* performs worse than *BBA* for reorder resolution. However, the probability of reorder occurrence is only 2.77% for all the cases, so *FastUp*'s overall performance is still within 90% of *BBA*.

TABLE VI
PERFORMANCE UNDER DIFFERENT RULE OVERLAPPING LEVELS.

Type		ACL_1				ACL_2			
$\boldsymbol{S_1}(k)$		2.0	4.0	6.0	8.0	2.0	4.0	6.0	8.0
$\boldsymbol{S_2}(k)$		2.8	5.4	8.1	10.0	5.0	10.0	14.7	19.6
$\boldsymbol{e_{avg}}$		1.4	2.0	3.0	3.3	8.0	12.0	14.6	18.7
$\boldsymbol{T_i}$(ms)		0.88	0.95	1.05	1.06	1.40	3.21	4.35	5.02
$\boldsymbol{T_c}$ (ms)	*FastUp*	0.03	0.11	0.19	0.31	0.21	0.59	1.11	1.58
	RuleTris	6.60	14.48	36.91	67.36	13.6	49.93	102.16	198.68
	Γ_{bh}	5.87	12.19	31.22	56.25	11.13	42.17	86.43	167.67

*7) **Influence of Rule Overlapping Level**:* To evaluate the influence of rule overlapping level on scheme performance, we use ClassBench to generate ACL_1 and ACL_2 with the lowest and highest level of rule overlapping configuration, and run the experiments again. The results are summarized in Table VI. e_{avg} is the average number of overlapping rules per rule. We draw two conclusions. First, a larger e_{avg} means longer T_c and T_i, because e_{avg} is directly related to the number of candidates to consider and the size of the converted table S_2. Second, the T_c of *FastUp* is much better than that of *RuleTris* and Γ_{bh}, although a larger e_{avg} tends to reduce the improvement.

VII. CONCLUSION

FastUp strives to optimize both computation time and interrupt time for new rule insertions. The use of the sequential stack is more efficient than the conventional dynamic programming in both time and space complexity. The freedom on search directions further reduces the update cost, which makes *FastUp* achieve the best interrupt time among the state-of-the-art schemes. Moreover, *FastUp* identifies and resolves the reorder problem, and ensures full utilization of the TCAM capacity. We are the first to provide a practical method to evaluate a TCAM update scheme's degree of optimality, and confirm that *FastUp* is close to the optimal.

REFERENCES

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

[2] B. Salisbury, "TCAMs and openflow-what every SDN practitioner must know," Accessed in Mar, 2021. [Online], Available: http://www.sdncentral.com/technology/sdn-Openflowtcam-need-to-know/2012/07/.

[3] M. Kuźniar, P. Perešíni *et al.*, "What you need to know about SDN flow tables," in *Proc. Passive Active Meas. Conf. (PAM)*, 2015, pp. 347–359.

[4] X. Wen *et al.*, "RuleTris: minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE ICDCS*, 2016, pp. 179–188.

[5] N. Katta *et al.*, "Cacheflow: dependency-aware rule-caching for software-defined networks," in *Proc. ACM SOSR*, 2016, pp. 1–12.

[6] X. Jin *et al.*, "Dynamic scheduling of network updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, 2014.

[7] N. Katta, O. Alipourfard, J. Rexford *et al.*, "Infinite cacheflow in software-defined networks," in *Proc. ACM HotSDN*, 2014, pp. 175–180.

[8] B. Yan *et al.*, "CAB:a reactive wildcard rule caching system for software-defined networks," in *Proc. ACM HotSDN*, 2014, pp. 163–168.

[9] F. Chang, W.-c. Feng, and K. Li, "Approximate caches for packet classification," in *Proc. IEEE INFOCOM*, 2004, pp. 2196–2207.

[10] J.-P. Sheu and Y.-C. Chuo, "Wildcard rules caching and cache replacement algorithms in software-defined networking," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 1, pp. 19–29, 2016.

[11] Z. Ding *et al.*, "Update cost-aware cache replacement for wildcard rules in software-defined networking," in *Proc. ISCC*, 2018, pp. 457–463.

[12] M. Yu *et al.*, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 351–362, 2010.

[13] R. Li *et al.*, "Taming the wildcards: towards dependency-free rule caching with FreeCache," in *Proc. IEEE/ACM IWQoS*, 2020, pp. 1–10.

[14] Z. A. Uzmi, M. Nebel, A. Tariq *et al.*, "SMALTA: practical and near-optimal FIB aggregation," in *Proc. ACM CoNEXT*, 2011, pp. 1–12.

[15] Y. Wan *et al.*, "T-cache: dependency-free ternary rule cache for policy-based forwarding," in *Proc. IEEE INFOCOM*, 2020, pp. 536–545.

[16] "Edge-core wedge100BF series switches," Accessed in Mar, 2021. [Online], Available: https://www.edge-core.com/.

[17] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches," *Comput. Netw.*, vol. 136, pp. 22–36, 2018.

[18] H. Xu, Z. Yu, X.-Y. Li, L. Huang, C. Qian, and T. Jung, "Joint route selection and update scheduling for low-latency update in SDNs," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3073–3087, 2017.

[19] G. Li, Y. Qian, C. Zhao, Y. R. Yang, and T. Yang, "DDP: distributed network updates in SDN," in *Proc. IEEE ICDCS*, 2018, pp. 1468–1473.

[20] H. Chen *et al.*, "Hermes: providing tight control over high-performance SDN switches," in *Proc. ACM CoNEXT*, 2017, pp. 283–295.

[21] G. Li, Y. R. Yang, F. Le, Y.-s. Lim, and J. Wang, "Update algebra: toward continuous, non-blocking composition of network updates in SDN," in *Proc. IEEE INFOCOM*, 2019, pp. 1081–1089.

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

[23] C.-Y. Hong, S. Kandula *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[24] J. Zheng, H. Xu, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proc. IEEE ICNP*, 2015, pp. 1–10.

[25] W. Zhang, G. Liu, A. Mohammadkhan, J. Hwang *et al.*, "SDNFV: flexible and dynamic software defined control of an application-and flow-aware data plane," in *Proc. ACM Middleware*, 2016, pp. 1–12.

[26] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1435–1461, 2018.

[27] X. Jin, J. Gossels *et al.*, "Covisor: a compositional hypervisor for software-defined networks," in *Proc. USENIX NSDI*, 2015, pp. 87–101.

[28] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai *et al.*, "Reinforce: achieving efficient failure resiliency for network function virtualization based services," in *Proc. ACM CoNEXT*, 2018, pp. 41–53.

[29] J. Zheng *et al.*, "Sentinel: failure recovery in centralized traffic engineering," *IEEE/ACM Trans. Netw.*, vol. 27, no. 5, pp. 1859–1872, 2019.

[30] J. Zheng, B. Li, C. Tian, K.-T. Foerster, S. Schmid, G. Chen, J. Wu, and R. Li, "Congestion-free rerouting of multiple flows in timed SDNs," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 968–981, 2019.

[31] H. Xu *et al.*, "Real-time update with joint optimization of route selection and update scheduling for SDNs," in *Proc. IEEE ICNP*, 2016, pp. 1–10.

[32] X. Jin, Y. Li, D. Wei *et al.*, "Optimizing bulk transfers with software-defined optical WAN," in *Proc. ACM SIGCOMM*, 2016, pp. 87–100.

[33] Z. Yu *et al.*, "Netlock: fast, centralized lock management using programmable switches," in *Proc. ACM SIGCOMM*, 2020, pp. 126–138.

[34] S. Pontarelli, R. Bifulco, M. Bonola *et al.*, "Flowblaze: stateful packet processing in hardware," in *Proc. USENIX NSDI*, 2019, pp. 531–548.

[35] A. Mohammadkhan, G. Liu, W. Zhang, K. Ramakrishnan, and T. Woodv, "Protocols to support autonomy and control for NFV in software defined networks," in *Proc. IEEE NFV-SDN*, 2015, pp. 163–169.

[36] M. N. Hall, G. Liu *et al.*, "Fighting fire with light: tackling extreme terabit DDoS using programmable optics," in *Proc. Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 42–48.

[37] B. Niven-Jenkins *et al.*, "Requirements of an MPLS transport profile," 2009. [Online], Available: http://tools.ietf.org/html/rfc5654.

[38] M. Al-Fares, S. Radhakrishnan *et al.*, "Hedera: dynamic flow scheduling for data center networks," in *Proc. USENIX NSDI*, 2010, pp. 89–92.

[39] H. Song and J. Turner, "Nxg05-2: fast filter updates for packet classification using TCAM," in *Proc. IEEE GlOBECOM*, 2006, pp. 1–6.

[40] Z. Wang, H. Che, M. Kumar, and S. K. Das, "CoPTUA: consistent policy table update algorithm for TCAM without locking," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1602–1614, 2004.

[41] D. Shah and P. Gupta, "Fast incremental updates on ternary-CAMs for routing lookups and packet classification," in *Proc. Hot Interconnects 8*, 2000, pp. 145–153.

[42] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast TCAM updates," *IEEE Trans. Netw.*, vol. 26, no. 1, pp. 217–230, 2017.

[43] 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.

[44] ——, "FastRule: efficient flow entry updates for TCAM-based openflow switches," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 484–498, 2019.

[45] B. Zhao, R. Li, J. Zhao, and T. Wolf, "Efficient and consistent TCAM updates," in *Proc. IEEE INFOCOM*, 2020, pp. 1241–1250.

[46] Broadcom, "12.8 Tb/s strataXGS tomahawk 3 ethernet switch series," Accessed: Mar. 31, 2021. [Online], Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56980-series.

[47] M. Malboubi, L. Wang, C.-N. Chuah, and P. Sharma, "Intelligent SDN based traffic (de) aggregation and measurement paradigm (iSTAMP)," in *Proc. IEEE INFOCOM*, 2014, pp. 934–942.

[48] X. Wen *et al.*, "RuleTris back-end update scheduler optimality proof," Accessed: Mar. 31, 2021. [Online], Available: http://bit.ly/1IFnxjj.

[49] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough," in *Proc. ACM SIGMETRICS*, 2007, pp. 253–264.

[50] K. Kannan and S. Banerjee, "Compact TCAM: flow entry compaction in TCAM for power aware SDN," in *Proc. Int. Conf. Distrib. Comput. Netw.*, 2013, pp. 439–444.

[51] H. Liu, "Routing table compaction in ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, 2002.

[52] X. Jin *et al.*, "Netcache: balancing key-value stores with fast in-network caching," in *Proc. ACM SOSP*, 2017, pp. 121–136.

[53] C. Monsanto, J. Reich, N. Foster, J. Rexford *et al.*, "Composing software defined networks," in *Proc. USENIX NSDI*, 2013, pp. 1–13.

[54] C. J. Anderson *et al.*, "NetKAT: semantic foundations for networks," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 113–126, 2014.

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

[56] N. Foster *et al.*, "Frenetic: a network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[57] C. Hu *et al.*, "Design of all programable innovation platform for software defined networking," in *Proc. Open Networking Summit*, 2014.

[58] L. Wirbel, "Xilinx SDNet: a new way to specify network hardware," *The Linley Group, White Paper*, 2014.

[59] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.

[60] J. Matoušek *et al.*, "Classbench-ng: recasting classbench after a decade of network evolution," in *Proc. ACM/IEEE ANCS*, 2017, pp. 204–216.