BubbleTCAM: Bubble Reservation in SDN Switches for Fast TCAM Update

Cong Luo[§], Chuhao Chen[§], Hao Mei[§], Ruyi Yao[§], Ying Wan[†], Wenjun Li[‡◇], Sen Liu[§], Bin Liu[†◇], and Yang Xu[§◇*]

[§]School of Computer Science, Fudan University, Shanghai, China
[†]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[‡]Harvard University, MA, USA [◇]Peng Cheng Laboratory, Shenzhen, China

2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS) | 978-1-6654-6824-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/IWQoS54832.2022.9812904

***Abstract*—The unique hardware structure of Ternary Content-Addressable Memory (TCAM) enables its unparalleled lookup throughput but also causes slow update due to the Priority Order Constraint (POC). With the increase of application demands, TCAM update has become a bottleneck in the network. This paper proposes a new TCAM management mechanism named BubbleTCAM to enable fast TCAM update, in which available empty entries are defined as bubbles. The core idea of BubbleTCAM is to uniformly distribute bubbles and dependency chains in TCAM, which is beneficial to updates. BubbleTCAM consists of two components: *bubble management* and *rule insertion*. Bubble management enables TCAM to have uniformly distributed bubbles at all times through three key procedures: *bubble lock reservation*, *bubble lock release* and *bubble generation*. Rule insertion ensures that dependency chains of rules are uniformly stretched and distributed in TCAM. In addition, BubbleTCAM avoids the reorder problem by pre-sorting. Our evaluation based on the rulesets generated by ClassBench shows that BubbleTCAM effectively reduces the average cost and worst cost (in units of rule movements) during rule updates by at least 48% and 50%, respectively. Especially for the worst cost, the performance can be improved by up to 196x.**

***Index Terms*—TCAM, Update, Bubble**

I. Introduction

Software-Defined Networking (SDN) [1] is increasingly being adopted by datacenter and enterprise networks because of its flexibility. The flexibility allows SDN to support a variety of applications (such as access control [2], traffic inspection [3] and flow filtering [4]) by customizing the flow table rules in switches. Due to its unparalleled lookup throughput and support for various matching patterns, Ternary Content-Addressable Memory (TCAM) [5] becomes the most widely used memory in SDN switches to store the flow table.

However, different from traditional memory devices (e.g., disk, SRAM, DRAM), TCAM suffers from slow updates due to the so-called Priority Order Constraint (POC). When a packet header matches multiple rules (due to the overlap among ternary rules), TCAM uses a priority encoder to return only the lowest matching rule (or the highest matching rule, dependent on the implementation of TCAM). Therefore, to ensure semantic correctness, rules overlapped must be stored in TCAM with decreasing priority order (i.e., POC). During TCAM update, a new rule must be placed in an empty entry without violating POC. If there is no such an entry, some existing rules need to be moved to make one. What is worse is that TCAM usually allocates entries continuously from top to bottom (or from bottom to top) [6], which causes a significant number of movements to keep POC when new rules arrive.

During a rule update, the regular lookup operations have to be suspended for the sake of consistency. The longer the update takes, the greater delay packets will experience, which may even lead to packet loss and degradation of Quality of Service (QoS). Meanwhile, more and more applications require strict update delay. For example, carrier networks have a strict 50ms requirement for failure recovery [7], which means that the re-routing rules must be installed within 25ms [8]. Traffic engineering only reserves 20ms to activate a new rule [9] [10]. In security systems, the throughput requirements are more stringent. However, the update delay in today's OpenFlow switches is far from what is required. As the recent measurements show, the delay of commercial OpenFlow switches to install rules ranges from 33ms to 400ms [11].

Despite a huge gap between update speed and delay requirements, the demand for rule update frequency is increasing. SDN introduces a high policy churn rate due to its flexible control of the network [12] [13] [14] [15]. As reported in [16], in a datacenter composed of 4k servers, 200k flows arrive every second, resulting in a large number of rules to be installed in the switch.

It is essential to achieve efficient TCAM updates to meet requirements of various applications. However, the existing solutions have some limitations. First, although TCAM has a completely different hardware structure and features from traditional storage devices, most of the previous work has adopted similar continuous memory (or entry) allocation methods [17] [18] [19]. The continuous allocation is aimed to exploit data locality to speed up data access [20] [21],

* Corresponding author: Yang Xu (xuy@fudan.edu.cn)

This work is sponsored by Key-Area Research and Development Program of Guangdong Province (2021B0101400001), National Natural Science Foundation of China (62150610497, 62172108, 62002066, 62102203, 62032013, 61872213, 61432009), Shanghai Pujiang Program (2020PJD005), China Postdoctoral Science Foundation (2021M690705, 2020TQ0158, 2020M682825, PC2021037), Basic Research Enhancement Program of China (2021-JCJQ-JJ-0483), Open Research Projects of Zhejiang Lab (2022QA0AB07), and the Major Key Project of PCL (PCL2021A15, PCL2021A02, PCL2021A08).

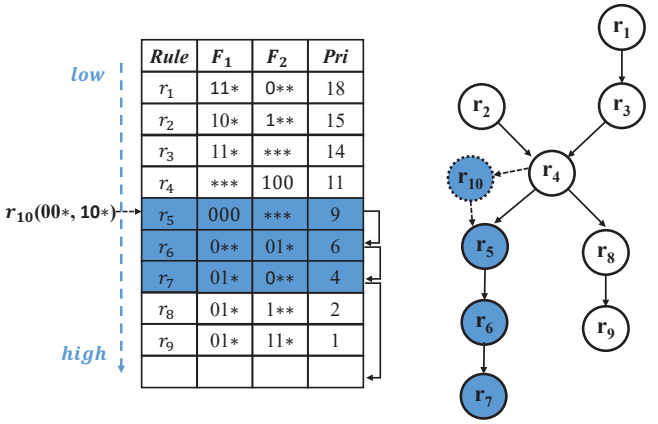
but in TCAM, it is unnecessary due to the parallel lookup. Meanwhile, continuous allocation leads to a large number of movements of existing rules when new rules are inserted. Memory interval allocation may be a better choice. Second, except for solutions using TCAM as a cache [22] [23] [24], most of them only consider the scenario where the SDN controller installs one rule at each time. But in reality, the controller may install overlapping rules together [22] [23] [24] [25] [26].

To overcome these limitations, we propose BubbleTCAM, a new TCAM management mechanism to enable fast rule updates. We propose a new concept: bubbles, which are the available empty entries for new rules. The core idea is to make bubbles and dependency chains (of existing rules) uniformly distributed in TCAM to speed up the insertions of future new rules. Through our analysis and experiments, maintaining the uniform layout for both bubbles and dependency chains is critical to accomplishing fast TCAM update. Therefore, we introduce *bubble management* and *rule insertion*. The former enables TCAM to have uniformly available bubbles at all times through three key procedures: *bubble lock reservation*, *bubble lock release* and *bubble generation*. The latter designs a novel cost calculation method that allows the dependency chains of rules to be uniformly distributed. The two core components are both based on memory interval allocation and are adapted to the scenario where the SDN controller delivers an overlapping ruleset each time.

In summary, the paper makes the following contributions:

1) We design a novel scheme BubbleTCAM, which adapts to realistic TCAM update scenarios, to achieve faster updates. To our best knowledge, it is the first scheme to systematically manipulate bubbles and use interval entry allocation to effectively reduce the rule movements during rule insertions.
2) To obtain uniformly distributed bubbles, we propose *bubble management*, which achieves the goal through three key procedures: *bubble lock reservation*, *bubble lock release* and *bubble generation*. Meanwhile, we introduce *rule insertion* to further accelerate the update, which ensures that dependency chains of rules are uniformly stretched and distributed in TCAM.
3) The advantage of BubbleTCAM in two different phases, warm-up phase and stabilization phase, is verified by simulation. Compared to the state-of-the-art schemes, BubbleTCAM reduces the average cost and the worst cost by at least 51% and 12 times in the warm-up phase. In the stabilization phase, the reduction is at least 48% on the average cost and 50% on the worst cost. Even for the worst cost, the performance can be improved by up to 196x.

The rest of this paper is organized as follows. Section II provides the background. The core motivation is presented in Section III. Section IV discusses two strawman schemes *bubble management* and *rule insertion*. Then, in Section V, we give our optimization of the strawman schemes and the final insertion algorithm of BubbleTCAM. Evaluation results are reported in Section VI. Section VII summarizes the related work. Finally, Section VIII concludes the paper.



(a) Flow table (b) Dependency graph

Fig. 1: A TCAM update example

II. Background

A. Priority Order Constraint

As mentioned above, the reason for slow update is that rules overlapped must be stored in TCAM with decreasing priority. We call this constraint Priority Order Constraint (POC). To understand it better, we briefly introduce some definitions.

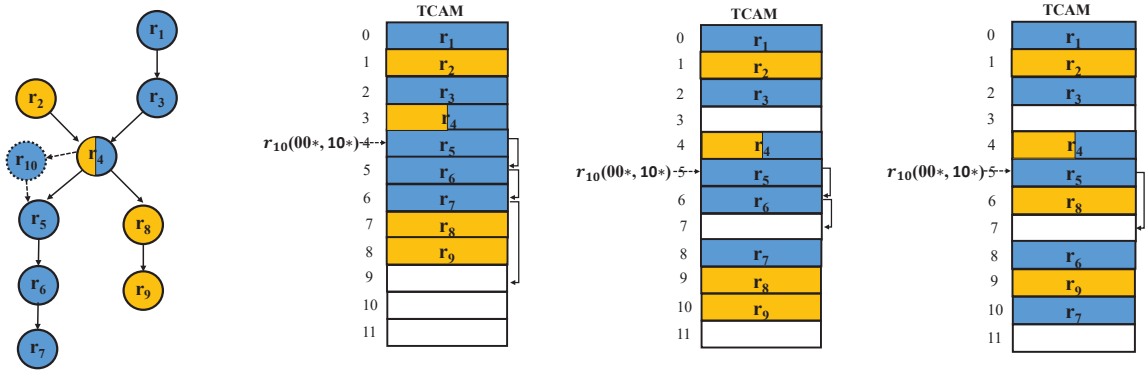
Application policies can be converted to one or more ternary rules and stored in a flow table. A rule $r = (pri, match, action)$ comprises three parts [27]. $r.pri$ denotes the priority of a rule, and a larger value means a higher priority. $r.match$ defines a set of packet header prefixes or ranges, all of which must be met to match the rule. $r.action$ specifies how to process the packet according to the matched rule. Most commonly used actions include dropping packets, forwarding packets to specific output ports, modifying packet headers (e.g., decreasing TTL) and so on. r_i and r_j overlap when $r_i.match \cap r_j.match \neq \varnothing$. A packet may match multiple rules, and only the rule with the highest priority will win. TCAM implements this mechanism by assigning low addresses (i.e., high positions) to rules with higher priority in the overlapping rules. If $r_i.match \cap r_j.match \neq \varnothing$, and $r_i.pri > r_j.pri$, we say r_j is dependent on r_i and use $r_i \rightarrow r_j$ to denote the dependency. The position constraint is modeled as POC, which can be formulated as the following equation:

$$\forall r_i, r_j \in R,\ if\ r_i \rightarrow r_j, r_i.addr < r_j.addr \qquad (1)$$

$r.addr$ indicates the address of r in TCAM. In [18], it shows that the dependency is a partial order relationship. That means the relationship $\rightarrow$ is transitive. If $r_i \rightarrow r_j$ and $r_j \rightarrow r_k$ then $r_i \rightarrow r_k$, even though $r_i.match \cap r_k.match = \varnothing$, r_i must be put above r_k.

B. TCAM Update

TCAM update includes rule deletion, insertion and modification. Both deletion and modification are simple and fast.



(a) Two dependency chains in the dependency graph

(b) Back to back layout (BBL)

(c) Back to back layout with uniform bubble (BBL-UB)

(d) Stretch layout with uniform bubble (SL-UB)

Fig. 2: The impact of layout mode on TCAM update

Thus, in this paper, the TCAM update refers to insertion as it is the most time-consuming operation. Fig. 1(a) shows a flow table with 9 rules. Actions are omitted for convenience. r_{10} is a new rule to be inserted and $r_{10}.pri = 10$. According to match field, r_{10} directly depends on r_5 and indirectly on $\{r_6, r_7\}$. In order to insert r_{10} correctly, and assuming that the TCAM (addresses from low to high) is arranged in the same way as the flow table, we have to make three movements.

To better guide the TCAM updates, we introduce the rule dependency graph, which is commonly used to describe the dependency in the flow table [17] [28] [29] and is a kind of Directed Acyclic Graph (DAG). Fig.1(b) is the rule dependency graph of Fig.1(a). Each node represents a rule. If r_j is dependent on r_i, a directed edge is formed, pointing from r_i to r_j. We call r_i is the ancestor of r_j, and accordingly r_j is the child of r_i. This relationship is transitive, for example, the child of r_j is also the child of r_i. A node without ancestors is named root, and a node without children is named leaf. The path starting from a root to a leaf is a dependency chain. Generally speaking, the number of movements required to insert a rule is related to the length of the dependency chain in which it is located. Since a node can be located in more than one dependency chain, there can be multiple movement schemes for inserting a rule. The shorter the dependency chain, the better the scheme. For r_{10}, as shown, the blue dependency chain is the best solution requiring three movements.

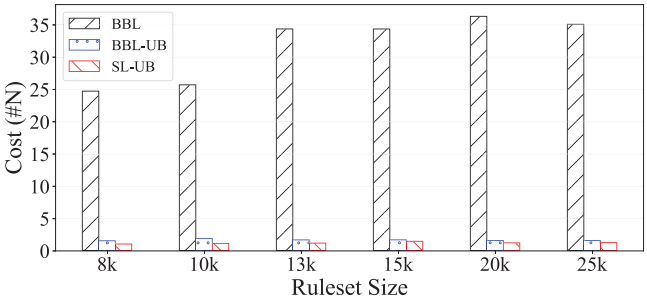
Although it is difficult enough to insert a rule, the actual situation is more challenging. In many cases, the ruleset is in the controller. When a packet $P_1(001, 100)$ arrives at the SDN switch and fails to match any rule, the corresponding rule will be installed in the switch. From the perspective of the controller, $P_1(001, 100)$ matches r_4 and r_4 should be installed. However, installing r_4 merely will lead to false matching for new packets matching the overlapping part of r_4 and r_3 or the overlapping part of r_4 and r_2. For example, when a packet $P_2(110, 100)$ arrives, it should be processed by r_3 as it has a higher priority than r_4. But the real process is that $P_2(110, 100)$ hits r_4 and performs the wrong action. Therefore, when a rule is to be inserted, it is necessary to install all the rules that overlap with it and have higher priorities (i.e., all its ancestors). This makes TCAM update tougher.

III. Motivation

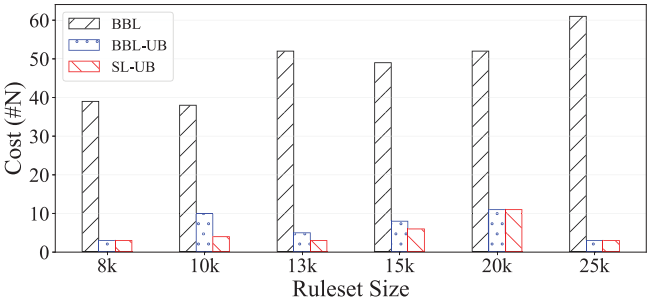
Inserting a rule is essentially a procedure to find an empty entry for the new rule. When there is no empty entry for the rule to satisfy the POC, existing rules have to be moved to make space. Under such circumstances, the number of movements is proportional to the length of the dependency chain between the desired insertion position and the nearest empty entry. We call this length the *interval chain length*. Therefore, shortening the interval chain length can speed up TCAM updates. The following examples provide us with a specific approach.

For the flow table in Fig. 1(a), Fig. 2(a) shows two dependency chains in its dependency graph, which are represented in blue and yellow respectively. The TCAM entries are numbered from 0. We denote the $i-$th position of TCAM by T_i. If we put the rules back to back as adopted by the previous scheme (i.e., the empty entries are concentrated at the bottom), then inserting r_{10} requires three movements as shown in Fig. 2(b). The insertion position is T_4, the nearest empty entry is T_9, and the length of the blue dependency chain between them is three, which results in the three movements. The worst case happens when a new rule is inserted at the top and the longest dependency chain needs to be moved.

One improvement is to make the empty entries uniformly distributed in the TCAM. The essence of this method is to reduce the interval chain length by shortening the distance between the insertion position and the nearest empty entry. In Fig. 2(c), we manually place an empty entry every three rules. When r_{10} is inserted, the insertion position is T_5, the nearest empty entry is T_7, and the distance between them is two, which is much closer than the previous continuous allocation scheme. The interval chain length is also shortened to two, which requires two movements accordingly. The worst cost is reduced from the length of the longest dependency chain to the longest interval chain length. As shown above, wisely



(a) Average cost



(b) Maximum cost

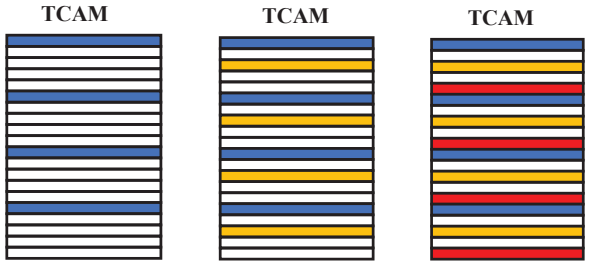
Fig. 3: Comparison among three layouts

managing empty entries can speed up the TCAM update. We refer to such artificially manipulated empty entries as bubbles.

Movements numbers can be further reduced by making the dependency chain uniformly distributed in the TCAM, which indirectly shortens the length of the dependency chain between two bubbles (i.e., the interval chain length). As shown in Fig. 2(d), while making the bubbles uniformly distributed, we also intersperse the blue and yellow dependency chains to make them uniformly distributed. In this way, most rules can be inserted in one movement, so can r_{10}. We call this layout of bubbles and chains uniformly distributed in TCAM as Stretch Layout with Uniform Bubble (SL-UB) because it likes stretching the dependency chains by the length of TCAM. From the above example and analysis, we know that keeping the TCAM in the SL-UB will benefit the TCAM update.

In order to verify the conclusion, we did some preliminary simulation experiments. The experiments compare the average cost and maximum cost of updating the same rules for the three layouts. The cost is the number of movements, which is represented by #N. The specific process is as follows. Given a flow table, we randomly select a subset of rules as base rules and pre-install them in three TCAMs according to the three layouts. Then use the remaining rules as updates. During the update, we only evaluate the cost of each insertion but do not actually insert the rule into the TCAM. Since the insertion may change the TCAM layout, the virtual insertion ensures that the comparison object is always the same.

Due to space constraints, we only show the results of FireWall (FW) rulesets generated by ClassBench [30] in Fig. 3. Ruleset sizes are from 8k to 25k. Access Control List(ACL) and IP Chain (IPC) rulesets have similar conclusions. Fig. 3(a) and (b) present the average cost and the maximum cost respectively. The trends are roughly the same and consistent with the above example and analysis. Compared with BBL, BBL-UB reduces the average cost by 15 times and the maximum cost by 3.3 times. As for the SL-UB, it decreases the average cost by a further 17% compared to BBL-UB. The maximum cost is reduced on most datasets. In summary, keeping the TCAM in the SL-UB can greatly speed up the update, in which case maintaining the bubbles' uniformity is the key and maintaining the dependencies chains' uniformity is the icing on the cake.



(a) Release G_0 (b) Release G_2 (c) Release G_4

Fig. 4: Bubble lock release

IV. Strawman Scheme

Based on the above observation, keeping the TCAM in the SL-UB is the key to achieving fast insertion. The implementation of SL-UB is divided into two parts: manipulating bubbles to be uniformly distributed by *bubble management* and making the dependency chains uniformly distributed in *rule insertion*. In this section, we will present the strawman schemes to accomplish the two parts respectively.

A. Bubble Management

Generally speaking, there are two ways to adjust the bubble distribution. The first is to passively wait for rules to be deleted, which will leave some scattered bubbles. The second is to manually move the bubbles to some specific locations at extra cost. However, both methods seem a bit extreme. The former completely depends on the location of the deleted rules. If there are no rules to be deleted, no bubbles will be generated. The latter is too proactive, and the design of the scheme is complicated due to rule dependency.

Therefore, we want to find a new method that can make the bubbles uniformly distributed in the TCAM simply and effectively. We introduce the concept of bubble lock to achieve the goal. When an entry is locked by the bubble lock, no rules can be inserted into the entry even if it is empty. We adjust the bubble layout in TCAM by using bubble locks wisely. The scheme can be summarized in two phases. The first phase is called bubble lock reservation. In this phase, we use bubble locks to lock all TCAM entries and divide them into p groups according to their positions. The second phase is called bubble lock release. Every time a new rule is inserted, we judge whether its insertion cost c is greater than a given threshold

k. If $c < k$, we insert the new rule directly, otherwise, we randomly select a group to release their bubble locks so that there are some uniform bubbles available in TCAM. We named this scheme *bubble management*.

A concrete example is shown in Fig. 4. White indicates that the entry is locked by a bubble lock. The other colors represent the bubble locks that are released each round (i.e., available bubbles). We label the TCAM entries from 0, like in Fig. 2. Then we take the label modulo 5, and the entries with the same remainder are regarded as the same group. Thus, the TCAM entries are divided into five equal parts, named G_0, G_1, G_2, G_3 and G_4 respectively. After bubble lock reservation, we insert the first rule. The rule cannot be inserted because the entries are all locked. We record the cost of not being able to insert as infinity, which is greater than the threshold k. So we release the bubble locks of G_0 (G_1, G_2, G_3 and G_4 are also possible), marked in blue in Fig. 4(a). As rules are inserted, the bubbles will become fewer and fewer, so the insertion cost will gradually increase. When the cost is greater than the threshold k again, we will release the bubble locks of G_2 as shown in Fig. 4. Similarly, we will release the bubble locks of $G4$ in the same case.

During the dynamic insertion process described above, there are uniform bubbles available in the TCAM most of the time. Of course, the selection of the threshold k and the number of aliquots p are both important. k indicates our definition of whether bubbles are sufficient in TCAM. When the insertion cost is greater than k, we consider that there are not enough bubbles. p represents how finely we control the bubble distribution. Continuous allocation of entries (bubbles at the bottom) is the special case of $p = 1$.

However, there is an obvious problem with bubble management. After all bubble locks are released, we will no longer be able to adjust the bubble layout. To solve the problem, we need a bubble generation scheme.

B. Rule Insertion

Making the dependency chain uniformly distributed in the TCAM is equivalent to making the rules the same relative position in the chain and TCAM. If a rule is in the middle of the dependency chain, then it should be inserted in the middle of the TCAM. Specifically, as shown in Fig. 2(a), r_{10} is a new rule and the other rules are already in the TCAM. The ancestors of r_{10} are $\{r_1, r_2, r_3, r_4\}$ and the children are $\{r_5, r_6, r_7\}$, which should be above and below r_{10} respectively. Therefore, to make the dependency chain uniformly distributed, r_{10} needs to be inserted in the position of $N * \frac{4}{4+3}$. N is the TCAM capacity. We call the scheme *rule insertion* and define the optimal position as Rule Insertion Position (RIP). $\forall r \in R$, if it has a ancestors and b children, then $RIP(r) = N * \frac{a}{a+b}$.

It seems that we can perfectly achieve the goal by inserting each rule into its RIP. However, the RIP of a rule changes dynamically as the rule is inserted. For example, before r_{10} is inserted, $RIP(r_4) = N * \frac{3}{3+5}$. But after r_{10} is inserted, b of r_4 changes from 5 to 6, so $RIP(r_4) = N * \frac{3}{3+6}$. The difference becomes apparent when N is large or the dependency chain is short (i.e., $a + b$ is small). The essence of this problem is that we cannot predict the future. It is vital to find a more appropriate RIP calculation method to reduce the impact of subsequent insertion rules.

V. BubbleTCAM

Both the *bubble management* and the *rule insertion* are inherently methods of implementing interval memory (entry) allocation. *Bubble management* forces rules to be inserted at intervals by restricting certain bubbles from being available. The *rule insertion* actively inserts rules into the corresponding RIP to achieve interval insertion. As demonstrated in Section III, making TCAM use interval entry allocation is an effective way to accelerate TCAM update.

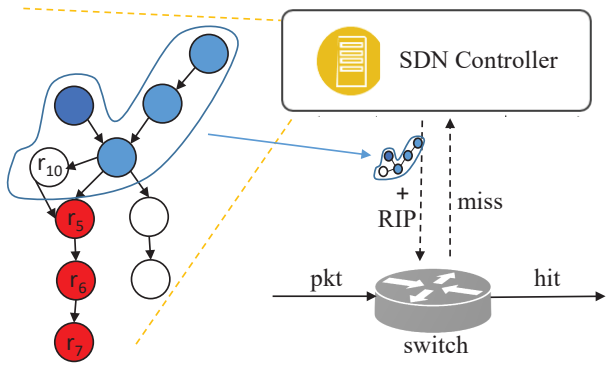
Our final solution, BubbleTCAM, inherits the above ideas. Moreover, it solves the problems of the strawman scheme in the real scenario where overlapping rules need to be installed together when inserting or deleting a rule. In the following, we first introduce the optimization of *rule insertion* and *bubble management*. Then we show the specific insertion algorithm of BubbleTCAM. It designs a novel cost calculation method, which makes *rule insertion* perfectly fit the three key procedures of bubble management: bubble lock reservation, bubble lock release, and bubble generation, to achieve fast TCAM update.

A. Optimizations

1) ***More Reliable RIP****:* If r has a ancestors and b children, then $RIP(r) = N * \frac{a}{a+b}$. But as new rules are inserted, the values of a and b may change. Therefore, the RIP calculated at the time of insertion is not the real optimal position.

In reality, we actually have more information to exploit during the insertion process. Fig 5(b) shows the workflow of packet processing in SDN. When a packet comes, if it does not hit any rules, the switch will ask the controller how to handle this rule. The controller performs the same matching process in its ruleset and returns the highest priority rule and its ancestors, which is to maintain semantic correctness. So during insertion, the number of ancestors is determined and will not be changed by the rules of subsequent insertions. Only the number of children is dynamically changing. The information can be used to optimize the calculation of RIP.

Fig 5(a) is the ruleset of the controller. We assume that the highest-priority rule hit by the packet is r_{10}. Its ancestors are identified in blue and its children in red. When installing r_{10}, the blue rules will be installed together, which means that r_{10} is determined to have four ancestors in TCAM. But red rules are full of uncertainty. In the subsequent process, they may or may not be delivered to the switch. If the probability of the three children r_5, r_6, and r_7 being the highest priority matching rule for a certain packet is x, y, and z, respectively. Then a more acceptable RIP for r_{10} should be $N * \frac{4}{4+x+2y+3z}$ (When installing r_7, r_5 and r_6 must be installed together. When installing r_6, r_5 must be installed.). But the probability of the rule being installed is still not available to us. It is



(a) Ruleset in controller (b) Workflow with controller

Fig. 5: Packet processing in SDN

also complicated to calculate RIP when the dependencies are complex. So we have to approximate the probability with the parameter α. For a rule r, $RIP(r) = N * \frac{a}{a+\alpha b}$. Through our experiments, 0.3 is suitable for most datasets.

*2) **A Uniform Bubble Generation Scheme**:* Once all the bubble locks are released, we lose control of the bubble distribution. It also means that TCAM is about to fill up and run out of bubbles. In order to take over the management of bubble layout again and alleviate the contradiction that the TCAM capacity is smaller than the size of the ruleset that needs to be installed, we tailored a bubble generation scheme for BubbleTCAM. It is also the third phase of bubble management, called bubble generation.

Combined with BubbleTCAM's unique insertion algorithm, the deletion-based scheme can generate uniformly distributed bubbles. The motivation is as follows. When deleting rules, we also need to consider the rule dependency. The logical relationship is the opposite of insertion. To insert a rule we need to be careful about rules with a higher priority. But during deletion, we should pay attention to lower priority rules. Suppose the ruleset in Fig. 5(a) is already installed in TCAM. If we want to delete r_{10}, we must delete the red rules as well. Otherwise, the false matching described in section II may occur. Put it simply, to maintain semantic correctness, deleting a rule from TCAM requires deleting its children along with it. So if we delete a root node, we have to delete all dependency chains starting from this root. At the same time, one of the core ideas of BubbleTCAM is to make the dependency chain uniformly distributed in TCAM. Therefore, if embedded insertion can complete the task well, we can select a root node and delete its corresponding dependency chain to generate uniform bubbles.

The only criterion for selecting the root node is the popularity of the rules. We hope that the dependency chain we choose to remove will not be reinstalled later. In other words, we don't want the hit rate to be damaged due to the generation of bubbles. Like most cache methods, we use the past network traffic to predict the subsequent network behavior. Each entry in TCAM has an associated counter that records the number of times the rule in the entry is matched. Considering that the popularity of a root r represents the popularity of the dependency chains starting from it, we use $\frac{CNT(r)}{n\_child(r)}$ to estimate it. The smaller the value the more we tend to delete r. The $n\_child(r)$ denotes the number of children, and $CNT(r)$ is the sum of the counters of all children. To make the count time-sensitive, we reset it periodically. Whenever the insertion cost is greater than the threshold k and there is no bubble lock to release, we will select the most outdated root node to delete according to the above method.

The performance of bubble generation depends on the uniformity of the dependency chains in TCAM after bubble lock release is completed. Therefore, we want to adjust the relevant parameters as strictly as possible to make the dependency chain more uniform.

B. Insertion Algorithm

With the above optimizations, the key modules for TCAM to maintain the SL-UB layout are in place. Next, we will introduce the insertion algorithm of BubbleTCAM, which puts everything together. It has a novel method for calculating insertion cost, which is based on RIP. And it enables TCAM to have uniformly distributed bubbles by bubble management.

When a miss is triggered, the controller will deliver the corresponding rules and their RIPs as shown in Fig. 5. Then the switch will insert the rules according to BubbleTCAM's insertion algorithm. Fig. 6 shows the overall workflow of the insertion algorithm. Generally, it is divided into two parts. On the left is the rule insertion, on the right is the bubble management. The arrow in the middle represents their interaction. Then we will show the workflow step by step.

Step 1: From rules to be inserted in a bunch, BubbleTCAM first sorts the rules in descending order of priority and processes them in sequence. This is mainly to avoid the reorder problem. The main reason for the reorder problem is that the dependency is transitive. If we insert r_1 and r_4 first, then insert r_3. Before r_3 is inserted, there is no dependency between r_1 and r_4. It is fine to place r_4 above r_1 in TCAM without violating POC. However, if r_3 is to be inserted, which happens to be a child of r_1 and an ancestor of r_4, r_1 must be moved above r_4 because they have a dependency due to the transitivity. Such a reorder problem requires a lot of effort to solve. By sorting, rules are processed in the direction of dependency transitions (i.e, ancestors must be inserted first), thus avoiding the problem.

Step 2: Then for a single rule, e.g., r_1 in Fig. 6, BubbleTCAM first identifies the candidate positions where it can be inserted. If we refer to the ancestor with the lowest position in TCAM as its youngest ancestor and the child with the highest position as its oldest child, then all the entries between its youngest ancestor and its oldest child are its candidate positions. The youngest ancestor is the upper bound and the oldest child is the lower bound. BubbleTCAM will calculate an insertion cost for each candidate position and select the smallest one for subsequent steps.

In BubbleTCAM, the insertion cost is divided into two parts. One is the minimum number of movements required to insert

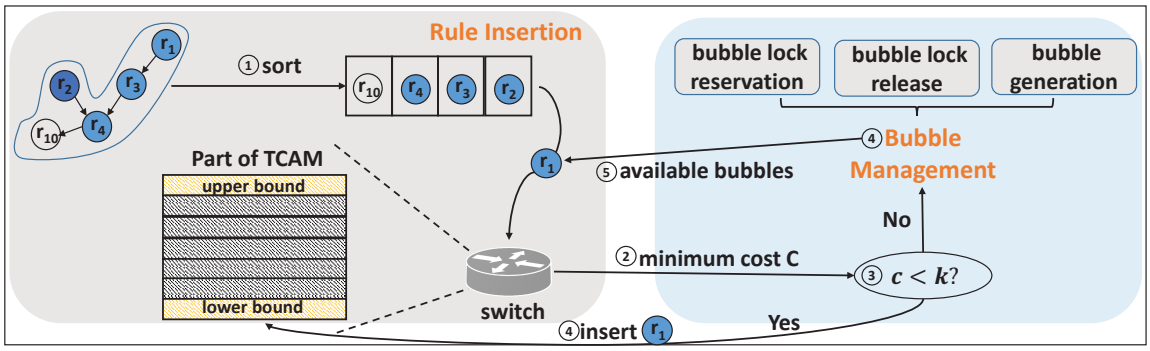


Fig. 6: The workflow of BubbleTCAM's insertion algorithm

the position. It can be easily calculated by dynamic programming [6] [8] [18] [31]. In the implementation, we adopt the calculation method of Fastup [32]. The time complexity is $O(m \log h)$, which is the best we know so far. m is the number of TCAM entries and h is the diameter of the rule dependency graph. We denote this part as $Mcost(T_i)$. T_i is the ith position in the TCAM.

The other part represents the impact on the TCAM layout caused by the insertion into that candidate position. This part can be divided into two sections. The first section is $diff(r, T_i) = |RIP(r) - i|$, which denotes the distance between the optimal position RIP of the new rule r and the candidate position T_i. The second section is $\sum_{r_i \in R}[diff(r_i, T^i_{after}) - diff(r_i, T^i_{current})]$. R is the ruleset that has been inserted into the TCAM. $T^i_{current}$ is the current position of r_i. T^i_{after} is the position where r_i is moved after inserting the new rule. So the section reflects the influence of the insertion on other rules. Specifically, a positive number means they are further away from their RIP and a negative number means they are closer to their RIP. The larger the absolute value, the greater the impact. We also note that not all rules will be affected. In the real case, only the rules on the moving path need to be calculated. The longest moving path is the longest dependency chain. Therefore, the time complexity of this part is $O(mh)$, which is acceptable. And due to bubble management, the moving path is usually much shorter than h.

Therefore, for the candidate position T_i of the new rule r, its cost is formulated in Equation 2.

$$Cost(T_i) = Mcost(T_i) + \lambda * [diff(r, T_i) + \sum_{r_i \in R} [diff(r_i, T^i_{after}) - diff(r_i, T^i_{current})]] \quad (2)$$

λ is the trade-off between the current and future interests. A larger λ indicates that we are more strict about TCAM layout.

Step 3: After calculating the cost of all candidate positions, BubbleTCAM compares the minimum cost C with the threshold k.

Step 4: If C is less than k, we insert the new rule into the corresponding candidate position. Then move on to the next rule. Otherwise, it means that there are not enough bubbles in the TCAM. At this point, we need to obtain uniformly distributed bubbles through the bubble management. Then the bubble management first tries to release the bubble lock. If successful, go to the next step. Otherwise, the bubble generation selects a root node and deletes it along with the associated dependency chains as described above.

Step 5: Once uniformly available bubbles are obtained, BubbleTCAM will repeat the above steps for the rule until it is successfully inserted into the TCAM.

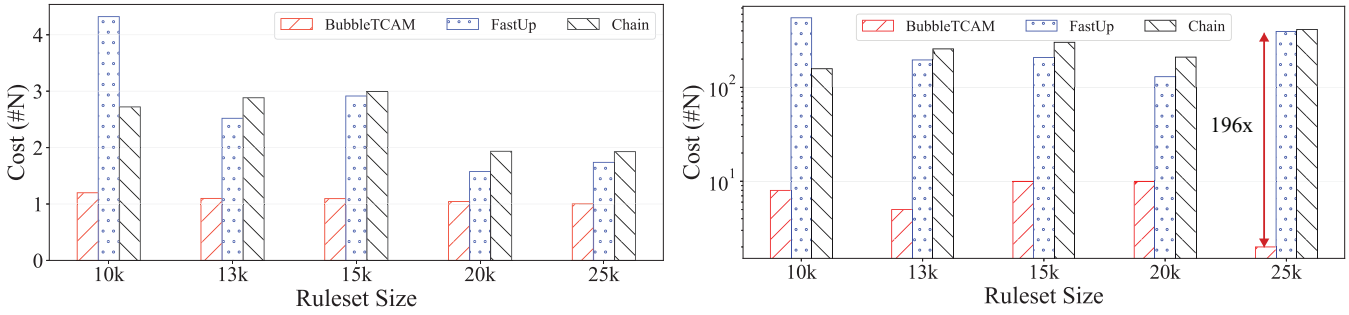
As a feature, BubbleTCAM has several adjustable parameters. Although the parameter adjustment is a bit complicated, it also means that BubbleTCAM can be more flexible to adapt to a variety of scenarios. First, when calculating RIP, we use α to approximate the probability of the children being installed. For datasets with poor locality, we take larger values. Then in Equation 2, we use λ to make a trade-off between the current insertion cost and the TCAM layout. We can make the dependency chain distribution more uniform by making λ larger. Finally, in the bubble management, we use the threshold k as a sign that there are not enough bubbles in the TCAM. The number of bubble locks released each time is also adjusted by p. In the next section, we experimentally illustrate the impact of some parameters on BubbleTCAM performance. Most of these parameters have a unique inflection point in performance, which is helpful for us to find the optimal value.

BubbleTCAM assumes that the ruleset in the controller is known in advance, which is true in most cases. Although the ruleset may change dynamically over time, we can adjust the scheme accordingly. If there are a few rules changed, we can ignore them and apply the traditional insert algorithm (such as [18]) to those rules, which will not affect BubbleTCAM. If a lot of rules are changed, we can periodically update the dependency graph of BubbleTCAM and recalculate the RIP to make the error acceptable.

VI. Evaluation

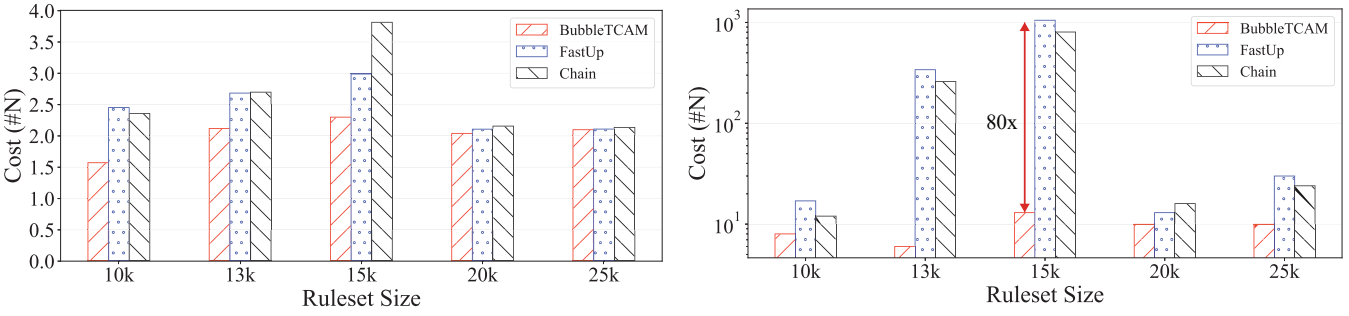
A. Experimental Setup and Dataset

We compare BubbleTCAM with FastUp [32] and Chain [18]. FastUp selects the best of the candidate positions and Chain always chooses the lower bound (or upper bound) as its insertion position. These two selection methods have different effects on the TCAM layout. We implement them in C++ and compile them using g++. The simulators are run on a commodity server with the Ubuntu 18.04-LTS operating system. As the Firewall (FW) rulesets have the most complex dependencies, we use ClassBench [30] to generate FW rulesets



(a) Average cost (b) Maximum cost

Fig. 7: Comparison with FastUp and Chain in warm-up phase



(a) Average cost (b) Maximum cost

Fig. 8: Comparison with FastUp and Chain in stabilization phase

with size from 10k to 25k to test the performance of all the update algorithms. Packet traces are generated twice the size of the corresponding rulesets. In all experiments the TCAM capacity is 5k.

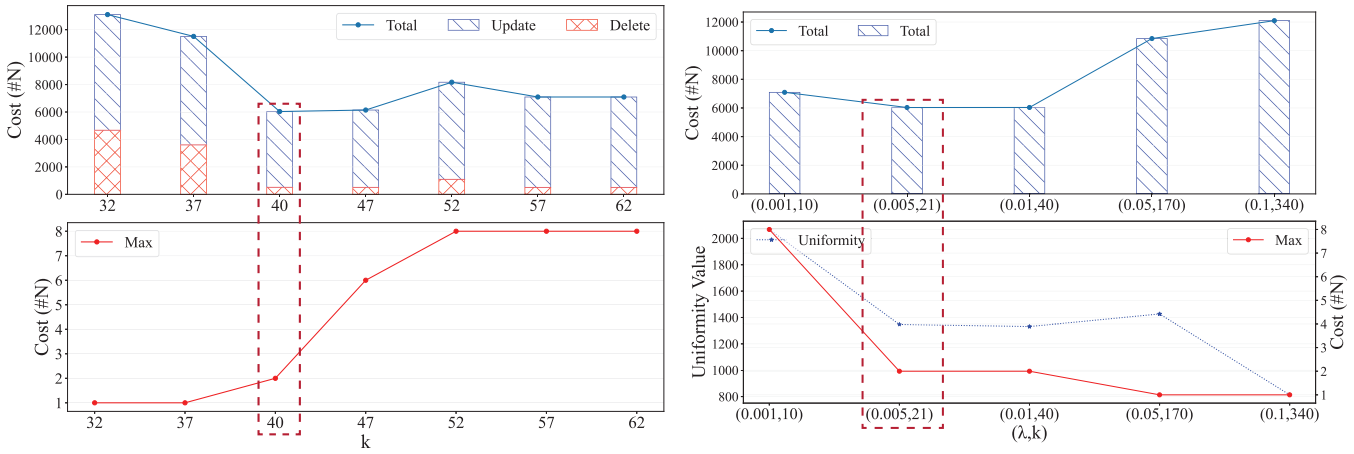
Due to the gap between TCAM and ruleset size, we set a threshold M to define large rules to prevent too many rules in a single installation. A rule is considered to be a large rule if it has more ancestors than M. When a packet hits a large rule in the controller, the controller no longer delivers the rule and its ancestors, but generates a new rule based on the packet header and sends it. The main metric used in experiments is cost (measured by #N, which stands for the number of movement steps). We do not compare the time to calculate the moving path with FastUp and Chain, as we introduce the deletion scheme. According to our experiments, if we don't need to delete the rules, the calculation time of BubbleTCAM is about 2ms per rule. If we need to delete the rules, it takes about 8ms per rule.

B. Comparison with Fastup and Chain

In order to reflect the contribution of each procedure of bubble management to the performance of BubbleTCAM, we divide the update process into two phases: warm-up phase and stabilization phase. The warm-up phase demonstrates the advantages of bubble lock reservation and bubble lock release. And the performance of bubble generation is presented in the stabilization phase. Once we need to delete rules, we consider the warm-up phase over and enter the stabilization phase. Since the previous algorithm does not consider rule deletion, for fairness, we make FastUp and Chain delete the same rules as BubbleTCAM. The cost of deleting a rule is 1. Parameters are set to $\alpha = 0.3$, $\lambda = 0.001$, $k = 12$, $p = 2$, $M = 1500$ (for 10k, 13k, 15k) or $M = 750$ (for 20k, 25k).

*1) **Warm-up Phase**:* The warm-up phase contributes the most to BubbleTCAM's superiority. Fig. 7(a) and (b) shows the comparison of the average cost and the maximum cost respectively. Compared to FastUp, BubbleTCAM reduces the average cost by 0.51∼2.6 times and the worst cost by 12∼196 times. As for Chain, the reduction is 0.85∼1.74 times on the average cost and 19∼206 times on the worst cost. The reduction in the average cost of our scheme is mainly due to the bubble lock which makes TCAM always have uniform bubbles in the warm-up phase. The worst-case improvement is because we avoid the reorder problem by sorting. Worst-case tends to have a dramatic impact on the network, which indicates tremendous packets may be suspended and consequently dropped. Therefore, the improvement of the worst case cost is critical.

*2) **Stabilization Phase**:* Fig. 8 presents the performance comparison of stabilization phase. As above, the left and right figures are the average cost and the maximum cost, respectively. For FastUp, the average and worst costs are reduced by 2%∼75% and 0.3∼ 80 times respectively. Chain is similar. BubbleTCAM reduces the average cost by 3%∼107% and the worst cost by 0.5∼61 times. The performance gain is less pronounced, especially on the 20k and 25k datasets. The main reason is that the huge gap between the ruleset size and the TCAM capacity makes us have to adjust M to avoid frequent misses. A smaller M means that the dependencies of the rules in TCAM become simpler, which makes the performance of all three schemes better. For the 10k dataset,



(a) The effect of K

(b) The effect of λ

Fig. 9: Effects of parameters

the reduction in the worst-case cost is not as significant as before. This is due to the dataset size being only twice the capacity of TCAM. After the first phase, the number of insertions is small and the load pressure is light, resulting in no reorder problem for the other two schemes. Except for the two reasons, in general, BubbleTCAM reduces the average cost by at least 48% and the worst cost by at least 42 times.

C. Effects of Parameters

Due to space constraints, we only present the effects of k and λ. We use the 10k dataset and corresponding 20k packet traces. Other parameters are set as $\alpha = 0.3$, $p = 2$, $M = 1500$. The number of rule updates varies with the parameters, so it is not appropriate to compare the average cost. We take the total cost instead.

Fig. 9(a) shows the effect of k with $\lambda = 0.01$. The upper and lower sub-figures show the total cost and the maximum cost respectively. The total cost is divided into two parts: deletion cost and update cost. A smaller deletion cost means a larger hit rate. The larger the maximum cost, the more complex the rule dependency in TCAM. It can be seen that k should not be too large or too small, and 40 is optimal. This is because if k is too small, BubbleTCAM will report to the bubble management that there are not enough bubbles when the TCAM is still empty so that frequent deletions and insertions are performed. For example, $k = 32$, the maximum cost is 1, but the total cost is the largest. If k is too large, BubbleTCAM will be insensitive and will not apply for bubbles when TCAM is almost full, resulting in large costs (such as $k = 62$, the maximum cost is 8). Therefore, for every λ there is an optimal k that minimizes the total cost.

We adjust λ and find the corresponding optimal k to show the effect of λ. We can find three conclusions from fig. 9(b). Firstly, the optimal k value increases as λ increases. Secondly, the larger the λ, the better the uniformity (the smaller the value), which is obtained by calculating the difference between the current position and RIP. The larger λ means that we pay more attention to the layout. In this case, most of the cost in BubbleTCAM comes from the change of the layout, which causes the real moving cost to be ignored. So the maximum cost is reduced. Finally, λ also has an optimal value. It can make a great trade-off between the real insertion cost and the TCAM layout so that the total cost is minimized ($\lambda = 0.005$ in the figure).

VII. Related Work

A. Computation-Based Insertion Algorithm

Rule insertion cost consists of computation delay and placement delay. The placement delay is proportional to the number of movements and the computation delay is the time it takes to find a short and correct sequence of moves. A solution that achieves the least place delay in a short time is pursued by academia and industry. RuleTris [8] and PoT [18] define the rule dependency through DAG and partial order theory respectively, and both use the method of dynamic programming to find the shortest moving path. Due to the unacceptable computation delay in dynamic programming, FastRule [6] applies a greedy algorithm and Bit Indexed Tree to calculate the moving path, and Fastup [32] employs Sequential Stack-based Algorithm. Moreover, Fastup shows that the previous solution to reordering problem may occur an infinite loop, and proposes Rule Chain-based Algorithm.

B. Group-Based Insertion Algorithm

Some work groups ruleset according to certain characteristics to speed up rule insertion. Mazu [33] divides the ruleset into high-priority and low-priority parts and places them in different blocks to eliminate the influence between the two parts. MagicTCAM [19] groups together rules with few dependencies, inspired by the fact that the fewer the rule dependencies, the smaller the insertion cost. ABUT [34] adopts the Topology-Order grouping when designing the batch update algorithm. A potential problem with these methods is that when a new rule is inserted, the group id of rules already installed in the TCAM will change, which may complicate the process.

C. Other Methods

Very few methods have mentioned empty entries. Most existing schemes keep the empty entries at the bottom (or top) of TCAM. Shah et al. [17] designs an algorithm with empty entries arranged in the middle. It reduces the worst case from the longest dependency chain length L to $\frac{L}{2}$. ABUT [34] indicates that the more uniform the empty entry distribution is, the better for TCAM updates. Empty entry distribution is taken into account when designing the batch update algorithm. However, ABUT has a loose control over empty entries and only uses existing empty entries. The empty entry selection is only compared when the insertion cost is the same.

Cacheflow [22] and T-cache [23] use TCAM as a cache to narrow the gap between TCAM capacity and flow table. And their algorithm is mainly designed for scenarios where multiple rules are inserted at each time due to rule overlap rules.

VIII. Conclusion

In this paper, we propose a new TCAM management mechanism, called BubbleTCAM, to reduce the gap between update latency and application demands. To our best knowledge, BubbleTCAM is the first scheme to systematically manipulate bubbles and use interval entry allocation. Through two core components, *bubble management* and *rule insertion*, BubbleTCAM always keeps TCAM as the SL-UB, which refers to the uniformity of bubbles and dependency chains and can greatly speed up the update. Testing FW rulesets with various sizes, we show that BubbleTCAM performs well in update: compared to the state-of-the-art approach, BubbleTCAM reduces the average cost and worst cost by at least 51% and 12 times in the warm-up phase. In stabilization phase, the reduction is at least 48% on average cost and 50% on the worst cost.

References

[1] J. Tourrilhes, P. Sharma, S. Banerjee, and J. Pettit, "The evolution of sdn and openflow: a standards perspective," *IEEE Computer Society*, vol. 47, no. 11, pp. 22–29, 2014.

[2] F. Klaedtke, G. O. Karame, R. Bifulco, and H. Cui, "Access control for sdn controllers," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[3] W. Queiroz, M. A. Capretz, and M. Dantas, "An approach for sdn traffic monitoring based on big data techniques," *Journal of Network and Computer Applications*, vol. 131, pp. 28–39, 2019.

[4] X. Yang, B. Han, Z. Sun, and J. Huang, "Sdn-based ddos attack detection with cross-plane collaboration and lightweight flow monitoring," in *IEEE Global Communications Conference (GLOBECOM)*, 2017.

[5] B. Salisbury, "Tcams and openflow-what every sdn practitioner must know," *See http://tinyurl. com/kjy99uw*, 2012.

[6] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "Fastrule: Efficient flow entry updates for tcam-based openflow switches," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 484–498, 2019.

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

[8] 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 *IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat *et al.*, "Hedera: dynamic flow scheduling for data center networks." in *Nsdi*, 2010.

[10] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz, "Latency inflation with mpls-based traffic engineering," in *Proceedings of the ACM SIGCOMM conference on Internet measurement conference*, 2011.

[11] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," *International Conference on Passive and Active Network Measurement*, pp. 347–359, 2015.

[12] B. Zhao, R. Li, J. Zhao, and T. Wolf, "Efficient and consistent tcam updates," in *IEEE Conference on Computer Communications (INFOCOM)*, 2020.

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

[14] G. Li, Y. Qian, C. Zhao, Y. R. Yang, and T. Yang, "Ddp: Distributed network updates in sdn," in *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[15] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM Conference on SIGCOMM*, 2013.

[16] R. Narisetty, L. Dane, A. Malishevskiy, D. Gurkan, S. Bailey, S. Narayan, and S. Mysore, "Openflow configuration protocol: implementation for the of management plane," in *second GENI research and educational experiment workshop*, 2013.

[17] D. Shah and P. Gupta, "Fast updating algorithms for tcam," *IEEE Micro*, pp. 36–47, 2001.

[18] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast tcam updates," *IEEE/ACM Transactions on Networking (ToN)*, vol. 26, no. 1, pp. 217–230, 2017.

[19] R. Yao, C. Luo, X. Liu, Y. Wan, B. Liu, W. Li, and Y. Xu, "Magictcam: A multiple-tcam scheme for fast tcam update," in *IEEE 29th International Conference on Network Protocols (ICNP)*, 2021.

[20] E. Berg and E. Hagersten, "Statcache: A probabilistic approach to efficient and accurate data locality analysis," in *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software*, 2004.

[21] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid)*, 2012.

[22] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," *Proceedings of the Symposium on SDN Research*, pp. 1–12, 2016.

[23] Y. Wan, H. Song, Y. Xu, Y. Wang, T. Pan, C. Zhang, and B. Liu, "T-cache: Dependency-free ternary rule cache for policy-based forwarding," in *IEEE Conference on Computer Communications (INFOCOM)*, 2020.

[24] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "Cab: A reactive wildcard rule caching system for software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[25] B. Yan, Y. Xu, and H. J. Chao, "Adaptive wildcard rule cache management for software-defined networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 962–975, 2018.

[26] ——, "Bigmac: Reactive network-wide policy caching for sdn policy enforcement," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2675–2687, 2018.

[27] 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.

[28] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying sdn programming using algorithmic policies," *ACM SIGCOMM CCR*, pp. 87–98, 2013.

[29] H. Song and J. Turner, "Nxg05-2: Fast filter updates for packet classification using tcam," *IEEE GLOBECOM*, pp. 1–5, 2006.

[30] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM transactions on networking*, pp. 499–511, 2007.

[31] Y. Wan, H. Song, and B. Liu, "Greedyjump: A fast tcam update algorithm," *IEEE Networking Letters*, 2021.

[32] Y. Wan, H. Song, H. Che, Y. Xu, Y. Wang, C. Zhang, Z. Wang, T. Pan, H. Li, H. Jiang *et al.*, "Fastup: Compute a better tcam update scheme in less time for sdn switches," *IEEE ICDCS*, pp. 1175–1176, 2020.

[33] K. He, J. Khalid, S. Das, A. Akella, E. L. Li, and M. Thottan, "Mazu: Taming latency in software defined networks," Tech. Rep., 2014.

[34] Y. Wan, H. Song, Y. Xu, C. Zhang, Y. Wang, and B. Liu, "Adaptive batch update in tcam: How collective optimization beats individual ones," *IEEE Conference on Computer Communications (INFOCOM)*, 2021.