MagicTCAM: A Multiple-TCAM Scheme for Fast TCAM Update

Ruyi Yao[a], Cong Luo[a], Xuandong Liu[a], Ying Wan[b], Bin Liu[bc], Wenjun Li[c], and Yang Xu[ac]

[a]School of Computer Science, Fudan University, Shanghai, China
[b]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[c]Peng Cheng Laboratory, Shenzhen, China

***Abstract*—Ternary Content-Addressable Memory (TCAM) is a popular solution for high-speed flow table lookup in Software-Defined Networking (SDN). Rule insertion in TCAM is a time-consuming operation. To ensure semantic correctness, rules overlapped must be stored in TCAM with decreasing priority order and many rule movements may be needed to make space for a single inserted rule. When a rule insertion is in progress, the regular flow table lookup will be suspended, which could lead to a degraded user experience for SDN applications. In this paper, we propose a multiple-TCAM framework named MagicTCAM to reduce the rule movements during a rule insertion. The core of MagicTCAM lies in three operations: *layering*, *partitioning* and *rotating*. By layering, rules with the least overlapping will be grouped (i.e., layered) into a sub-ruleset. The number of rule movements is therefore greatly reduced as most of rules in a sub-ruleset are non-overlapped. To achieve balanced load in TCAMs, rules in each sub-ruleset are further partitioned and dispatched into different TCAMs in a rotating manner. In addition, an inter-TCAM movement algorithm is proposed to allow rules to be moved between TCAMs for reduced rule movement. Experiment results show that with two half-sized TCAMs, MagicTCAM reduces the rule movements by 39% on average compared with the state-of-the-art work while the computation time is shortened by half as well.**

***Index Terms*—Ternary Content Addressable Memory, Update**

I. Introduction

The flexibility of Software-Defined Networking (SDN) [1] allows it to support a variety of application scenarios. The way to cater for applications is to customize ruleset R in the flow table. For its ultra-fast lookup, Ternary Content-Addressable Memory(TCAM) [2] stands out and becomes the most widely used memory in SDN to store the flow table. However, faced with the ever-increasing line rate, TCAM update is the main bottleneck for performance [3].

More and more applications bring gigantic update demands. For example, when TCAM is used as a cache [4] in OpenFlow switches, replacement happens frequently. In addition, the network configuration often needs to be updated in order to accommodate various demands in multi-tenant data centers [5]. What's more, the emerging Intent-Driven Networking (IDN) [6] and autonomous networks apply faster rule updates in real time intelligently.

Despite the huge amount of rule insertion, deletion and modification, applications require strict update delay [7]–[12]. To avoid congestion or packet loss in carrier network [13], once failure detected, re-routing rules have to be implemented within 25ms. Besides, traffic engineering SDN control programs such as Google's B4 [14] or Microsoft's SWAN [15] leave only a 20ms delay budget for update [16]. Furthermore, strong performance guarantees are especially needed in security systems and critical infrastructures. As the threat detection is done at line rate and defenses must take effect as soon as possible, advanced malware quarantine [17] [18] in enterprise networks has an even stricter delay bound. Similarly, for cyber-physical systems [19], security relies on the rapid response to the environment, which makes low update delay indispensable.

The lookup performance also imposes requirements on the update delay. When the update is in progress, the lookup has to be suspended. As the recent measurements show, commercial OpenFlow switches install rules with delay varying from 33ms and 400ms [20] [21], which means 600-7000 packets suspending on an OC-192 interface. It will be a greater disaster on the commonly used 100Gbps interface.

There is still a great gap between the modern switches update delay and the application requirements [4] [22]. Among TCAM updates, the insertion is the most time-consuming operation. Insertion delay consists of computation delay and placement delay. To ensure semantic correctness, rules in TCAM must be stored in order of decreasing priority. The condition requires that when a rule comes, existing rules may need to be moved to make space for it to keep the order constraint. A solution which achieves the least placement delay in a short time is pursued by academia and industry.

The placement delay is proportional to the number of rules that need to be moved per insertion. There is already a large corpus of literature dealing with TCAM update, devoted to minimizing the move cost and achieving good average performance. However, there is still room for improvement. Previous works have revealed the regularity of the difficulty in updating TCAM: 1) Rules overlapped must be stored in TCAM with decreasing priority order [23]; 2) The higher

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

This work is supported by Key-Area Research and Development Program of Guangdong Province (2021B0101400001), National Natural Science Foundation of China (62172108, 62032013, 61872213, 61432009), Shanghai Pujiang Program (2020PJD005), NSFC-RGC Joint Research Scheme (62061160489) and China Postdoctoral Science Foundation (2020TQ0158, 2020M682825).

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

the degree of overlapping among the rules, the more rule movements needed to insert a rule [24].

Based on the above observations, we propose MagicTCAM, a multiple-TCAM scheme for TCAM update acceleration. The rationale behind MagicTCAM is to minimize the overlapping relationship among rules by dividing the ruleset and placing the sub-rulesets into different TCAMs, thus reducing the rule movements. To separate the wheat from the chaff, there are mainly two kinds of TCAMs: DataTCAM and spTCAM. The complicated dependencies are stored in spTCAM and DataTCAMs are kept clean. New rules will only be inserted in DataTCAMs to avoid suffering from complex dependencies. To eliminate the effect of unbalanced sub-ruleset size, rules in a sub-ruleset are further partitioned and dispatched into different TCAMs to make TCAMs space load balanced.

Both putting rules into different TCAMs and partitioning the rulesets help reduce the overlapping relationship of rules, which will be small enough in MagicTCAM to guarantee the placement delay.

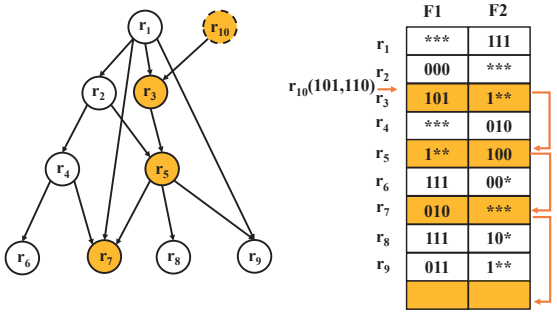
The contributions of this paper can be summarized as follows:

1) We design a novel framework MagicTCAM, which employs multiple TCAMs to provide an ideal solution for the challenge of slow rule insertion. It is compatible with any existing rule insertion algorithm for a single TCAM.
2) To accelerate the rule insertion speed and keep TCAM space load balance, layering, partitioning and rotating are proposed. Both layering and partitioning split the rulesets and reduce the dependencies, while rotating the partitions helps balance the TCAM space load.
3) We creatively propose spTCAM, which stores the majority of the dependencies that are difficult to cut off. As new rules are only inserted in DataTCAMs, insertion can be quickly finished under the shield of spTCAM. Tricks like inter-TCAM movement are also proposed to further reduce the rule insertion cost.
4) Through experimental verification, insertion algorithm using the MagicTCAM framework can improve the insertion performance and the calculation speed significantly. Meanwhile, compared with the state-of-the-art multiple TCAM update scheme BW-split [25], the insertion rule movement number is reduced by 39% and the computation time is shortened by 2.25 times.

II. Background and Motivation

A. Flow Table

Whatever the application, the customized rules are abstracted as $r = (prio, match, action)$ [26]. $r.match$ consists of zero or more constraints on individual fields, all of which must be met to satisfy the match. When a packet comes, the flow table is looked up to find which rule it matches and this process is called packet classification. For packet forwarding, the match field is the packet destination IP address, while for other applications, the match field(s) can be 5-tuple or even more packet header fields. $r.action$ specifies how to process the packet according to the matched rule. Most commonly used actions include decreasing TTL, dropping packets, outputting the packet to a port and so on. Rules in a flow table may overlap (i.e., $r_i.match \cap r_j.match \neq \varnothing$). $r.prio$ denotes the priority of a rule. If multiple rules are matched, the one with the highest priority should be taken. A smaller number indicates a lower priority. Zero is usually assigned to the default rule.



(a) (b)

Fig. 1: Minimum dependency graph and rule placement in TCAM

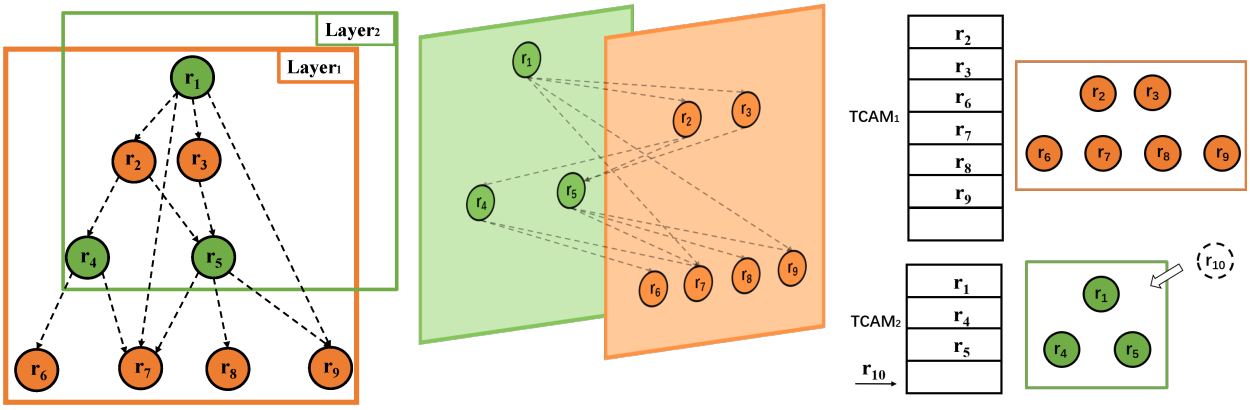
B. Rule Graph for TCAM Update

TCAM is the most widely used memory to store the flow table. When rules are matched, the principle of TCAM is to return the lowest address of them. TCAM update includes rule insertion, deletion, and modification. Deletion is a simple and fast operation and modification can be constant [27]. As insertion is the most time-consuming operation, we are committed to reducing the insertion time. To ensure the correctness of the lookup, rules should be placed in TCAM in descending priority order. This results in multiple movements to make room for the newly inserted r_i. Fortunately, it is not necessary to move all rules with low priority. Since non-overlapping rules will never be matched together, only overlapping rules should keep the relative descending priority order. This relative rule position constraint is modeled as the rule dependency. When r_i and r_j are matched together, if r_i should be returned, we say r_j is dependent on r_i and use $r_i \rightarrow r_j$ to denote. The TCAM addresses storing the rules should conform to the condition [28]:

$$\forall r_i, r_j \in R,\ if\ r_i \rightarrow r_j, A[r_i] < A[r_j] \tag{1}$$

$r_i \rightarrow r_j$ here indicates that $r_i.prio > r_j.prio\ and\ r_i.match \cap r_j.match \neq \varnothing$. $A[r]$ is the TCAM address of rule r. A movement sequence or a rule placement scheme is valid as long as the $Condition\ 1$ is satisfied.

The minimum rule dependency graph is commonly used to describe all the dependencies in the flow table [23] [29] [30]. 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 their relationship ancestor and child. The path starting from the oldest ancestor to the youngest child is the dependency chain. Many chains are intertwined to form the directed acyclic graph (DAG). When a new rule is to be inserted, chains or a segment



(a) (b) (c)

Fig. 2: Broken dependencies by 2 TCAMs

of chains are candidate movement paths, some of which are valid while others invalid according to $Condition$ 1. Fig. 1 (a) shows an example of the rule dependency graph formed by rules $r_1 \sim r_9$ with descending priority. Without loss of generality, their matching fields consist of F1 and F2. The rule placement in TCAM is shown in Fig. 1 (b). Rule r_{10} is to be inserted, and it overlaps with r_3. $r_{10} \rightarrow r_3 \rightarrow r_5 \rightarrow r_7$ is the shortest valid movement path and $r_{10} \rightarrow r_3 \rightarrow r_5 \rightarrow r_8$ is invalid. If r_5 replaces r_8, $Condition$ 1 will be violated as $A[r_5] > A[r_7]$.

Although focusing only on overlapping rules reduces the movement steps, it brings a challenge. Non-overlapping rules are placed in an arbitrary order. If a new rule is inserted in the TCAM, chances are non-overlapping rules become dependent and $Condition\ I$ is violated. For example, it is fine to place r_7 below r_9 as they are independent. However, if a new rule r_n is to be inserted, and $r_7 \rightarrow r_n$, $r_n \rightarrow r_9$, then r_7 must be moved above r_9 to satisfy $Condition\ I$. This is the reorder update case. The movement cost of reorder case is usually 10 times as large as that of the common case [25].

C. Motivating Example

Recent studies of TCAM update performance [24] [25] have been devoted to finding the shortest valid movement path quickly. Here we summarize some of their key findings and use them to help motivate our design.

The placement cost of insertion is proportional to the number of movement steps. It is caused by the order constraints that stem from rule dependency and related to the distribution of empty entries. In general, the longer the chain length of the DAG, the more movements are needed. The more evenly dispersed the empty entries are, the smaller the average cost of insertion. The worst case happens when all the empty entries are at the low addresses in TCAM. According to the $Condition\ I$, the number of movements is bounded to the longest chain length, i.e. the dependency graph diameter. Since rule dependencies are the root cause of slow insertion, by deliberately cut the dependencies and make the chain length of the DAG formed by the rules in a TCAM as short as possible, the insertion can be accelerated.

In Fig. 1, at least 3 movements are needed to make room for r_10. However, movement numbers can be reduced by dividing ruleset into independent subsets and placing these subsets in different TCAMs. As shown in the Fig. 2 (b), the ruleset is split into 2 subsets. If we place rules $r_2, r_3, r_6, r_7, r_8, r_9$ in $TCAM_1$ and rules r_1, r_4, r_5 in $TCAM_2$, then all the edges originally created by the overlapping rules will disappear, which means that no dependency exists. In such induced sub-graphs composed of independent sets, rules can be placed in any order. The cost of inserting new rules can be as low as only one write, which means no movements.

When inserting rule r_{10}, both TCAMs can be chosen to insert. As shown in Fig. 2 (c), choosing $TCAM_2$ to insert is better as it requires 0 movement while r_3 has to be moved if r_{10} is to be placed in $TCAM_1$. As for the lookup, the two TCAMs are inquired in parallel, and the rule with higher priority is returned as the result.

III. MagicTCAM

By reasonably dividing the ruleset and selecting TCAM for insertion, we can significantly reduce the rule dependency and achieve fast insertion.

In this section, we first introduce the main idea of MagicTCAM, which proposes layering, partitioning, rotating to make the chain lengths as short as possible and rules in each TCAM balanced. Then we show the architecture and workflow of MagicTCAM. Finally, we present three optimization methods. New rules will only be inserted in DataTCAMs to avoid suffering from complex dependencies and inter-TCAM movement provides more choice for rule insertion.

A. Insights

1) Layering: Based on the above rationale, we divide rulesets and put subsets in different TCAMs. In the division process, the goal is to minimize the rule dependencies in each TCAM, which brings the least rule insertion cost.

The process is like dividing a photo into multiple layers, as shown in Fig. 2 (a). The DAG is divided into two layers, in which $Layer_1$ has 6 rules and $Layer_2$ has 3 rules. A well-done layering enables the chain of induced sub-graphs in each

TCAM to be short. In the best case, rules do not overlap with each other and no movements are needed when insertion. What's more, each rule has multiple TCAMs to choose from, which means that we can achieve the least insertion cost under the controller's view.

We carve out the maximum independent set from the rule dependency graph, and put the subset of rules into a TCAM. An independent set is a layer of rules. This process is repeated n_t times, generating n_t layer rule sets. With n_t the number of DataTCAMs in MagicTCAM, each layer is placed into one DataTCAM. In the experiment, ACL rulesets with various sizes are used. After taking out 2 maximum independent rules, no more than 15% rules remain. This portion of rules are placed in a smaller TCAM called spTCAM. The larger number of TCAMs, the fewer residual rules.

A problem that cannot be ignored is that the sizes of independent sets differ greatly. The layer-to-TCAM one-to-one placement scheme renders the utilization of DataTCAMs severely unbalanced. Generally, $Layer_1$ will be an order of magnitude larger than $Layer_2$. As the number of layers increases, the independent set taken out gets smaller and smaller. With $num(R)$ representing the size of ruleset R and L_i represents the ith layer, we have $num(L_i) > num(L_j)$ when $i < j$.

We propose partitioning and rotating to figure out the unbalance. Not only the number of rules in each TCAM is balanced, but also the dependency chain between each partition can be further cut off.

2) Partitioning: In order to solve the unbalance of the number of rules in layers, we further divide the ruleset of each layer by evenly partitioning. A simple sort-based heuristic is opted for where the dimension with the most unique projections of the rules is chosen. We sort the endpoints of the projections and evenly split the rules into n_p partitions.

The number of cross-partition rules is about 15%, which is determined by the rule distribution and the partitioning algorithm. It can be reduced by modifying the algorithm. In order to eliminate the rule replication, we put rules across multiple partitions into the spTCAM. With several cuts, large rules are extracted. In this way, space waste caused by rule replication is reduced, and overlaps among small and large rules are broken. Although taking away these rules will make the number of rules in each partition uneven, the degree is tolerable.

3) Rotating: Supposing that each layer is divided into 2 partitions, A and B with equal size. Namely, L_i is divided into L_{iA}, L_{iB}, and $num(L_{iA}) = num(L_{iB})$. Since $num(L_1) > num(L_2)$, we have $num(L_{1A}) > num(L_{2A}), num(L_{1B}) > num(L_{2B})$. If we put L_{1A} and L_{2B} in $DataTCAM_1$, and L_{2A} and L_{1B} in $DataTCAM_2$, then we can make the DataTCAMs space load balanced as shown in Fig.3.

It is feasible on modern TCAMs [31] which are organized in slices. One or multiple slices are mapped to a logical group to store rules [32]. Groups storing rulesets are called buckets in this paper. We can designate the slices number for buckets and the buckets hold partitions from larger layer are allocated

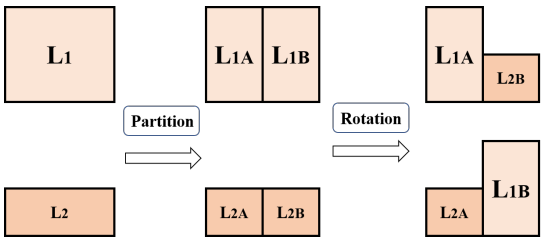


Fig. 3: Mixture of Layer 1 and Layer 2 for balanced size

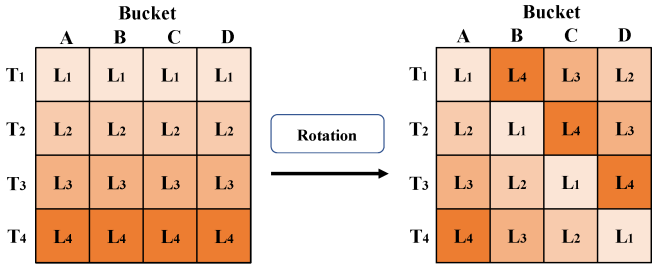


Fig. 4: Rotating placement of partitions

more slices. For example, if $DataTCAM_1$ stores L_{1A} and L_{2B}, the bucket for L_{1A} is allocated more slices than the bucket for L_{2B}. While in $DataTCAM_2$, the bucket for L_{1B} is allocated more slices than the bucket for L_{2A}. As long as the partition-to-TCAM placement is determined, the slices numbers allocation is set correspondingly.

A DataTCAM has n_p buckets, assigning one bucket for each partition. Buckets for a partition can store rulesets from any layer, as long as the rulesets belong to this partition. After the rotating, each DataTCAM stores part of multiple layers and the rulesets in all the DataTCAMs are balanced.

As for the rotating, we place the partitions in DataTCAMs layer by layer. Firstly, $Layer_1$ is placed, and partitions spread in order from the first DataTCAM to the last DataTCAM. If the number of DataTCAMs n_t is less than the number of partitions n_p, the remaining partitions will continue to be placed from the first DataTCAM. As for the second layer, partitions are put from the second DataTCAM. By analogy, partitions of the $i-th$ layer is put from the $i-th$ DataTCAM. Use T_i to represent the $i-th$ DataTCAM, the placement scheme for 4 partitions A, B, C, D of 4 layers are shown in Fig. 4. We use T_{1A} to represent the bucket assigned to partition A in T_1. T_{1A} stores rules in partition A of layer 1. Partitions B,C and D are put in T_{2B}, T_{3C}, T_{4D} respectively. Similarly, rules of layer 4 are put in $T_{4A}, T_{1B}, T_{2C}, T_{3D}$.

For the convenience of packet classification, partition cutting is performed first, then we layer rules in each partition. This means that the match domain of the partitions in all layers are the same and the number of index rules is small, which equals to np. Even without TCAM, it is easy to quickly find out which partition a packet belongs to. Finally, rotating is performed to achieve TCAM balance. When the packet comes, it will only hit rules in one partition according to index rules or rules in spTCAM. As each of np partitions is placed in a TCAM bucket, instead of searching the entire DataTCAMs,

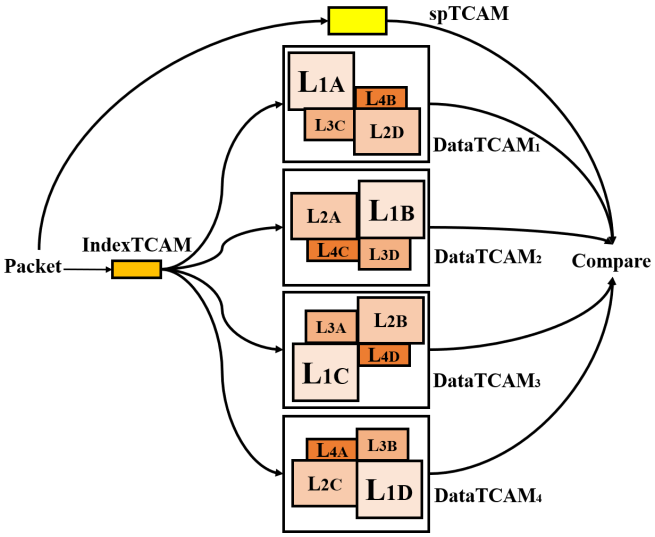


Fig. 5: Architecture of MagicTCAM

only one bucket in each DataTCAM will be enabled. So only rules in the same bucket (partition) need to maintain relative priority order and there is no need to pay attention to dependencies of rules between partitions in a DataTCAM. In other words, dependencies are further broken.

B. Architecture and Workflow

The architecture consists of 3 kinds of TCAMs, namely IndexTCAM, DataTCAM and spTCAM. Layering, partitioning and rotating are employed during initialization. IndexTCAM stores index rules for partitions. Independent sets generated by layering are placed in DataTCAMs. Rules sit in multiple partitions are held in spTCAM, as well as residual rules apart from independent sets.

In the system, the number of TCAM n_t and the number of partition n_p are adjustable parameters. Here we take 4 DataTCAMs and 4 partitions as example to show the architecture and work flow. The ruleset is first divided into four partitions: A, B, C, and D. Then each partition is split into four layers 1, 2, 3, and 4. Finally, subsets are placed in the buckets according to the rotating scheme. The residual rules and cross-partition rules are left in the spTCAM. A larger colored block in DataTCAMs shown in Fig. 5 indicates a larger bucket allocated more slices.

1) Lookup: The lookup includes three stages. When a packet comes, first determine which partition it belongs to according to indexTCAM. Then the corresponding partitions in the 4 DataTCAMs and spTCAM are activated at the same time. Finally, the one with the highest priority among all the matched rules will be the result. With a pipelined design, MagicTCAM can maintain the line-rate search with just two clock cycles of extra latency for each packet, with one cycle spent in indexTCAM and one spent on comparing priorities of rules returned by TCAMs.

2) Update: The first step of insertion is also identifying which partition the rule locates. If it crosses multiple partitions, the rule is put in spTCAM. Otherwise, the rule is evaluated whether it will form dependency in the bucket of the corresponding partition in all DataTCAMs. No dependency formation is the prerequisite to enter a DataTCAM. Rules that fail to be placed in DataTCAM will be stored in spTCAM. In our experiment, about 30% of the rules are put into spTCAM. Both cross-partition rules and rules that fail to form independent sets account for about 15% of the total volume. The law of ratios is the same as mentioned in Section III-A1 and Section III-A2.

All the DataTCAMs and spTCAM are checked for rule deletion and modification. This process can be operated in parallel, so MagicTCAM does not consume extra time compared with a single TCAM.

3) Compatibility with Modern Switch: MagicTCAM can be an enhanced flow table searching and updating engine for the next generation of programmable switch chips. With the total TCAM resources not changed, a TCAM can be easily split into multiple small TCAMs in ASIC design to support the MagicTCAM architecture. Extended primitives can be added in the future version of P4 language to support simultaneous access to multiple TCAMs. Details are reserved for future work.

C. Optimizations

Although most rules can be inserted directly without moving, spTCAM's complex dependencies make insertion in it more costly. Since all DataTCAMs and spTCAM need to be activated during a query, the update delay in spTCAM affects the overall throughput. We take three actions to mitigate the burden in spTCAM and improve the overall performance.

Firstly, we relax the independent set and allow sub-DAGs with chain lengths no longer than threshold C when initialization. Secondly, new rules are placed in the DataTCAM with the least insertion cost. If the cost exceeds C, to quickly free up space for the new rule and reduce the terrible reorder cases, inter-TCAM movement is proposed. Thirdly, cross-partition rules are replicated and inserted in DataTCAMs.

After optimization, all new rules are placed in DataTCAMs instead of spTCAMs. Subsets of rules with complex dependencies are sealed in spTCAM, escorting rule updates in DataTCAM. Although the first and the third actions will make chain lengths in DataTCAMs a little longer, these actions effectively improve the overall insertion speed.

1) Bounded Chain Length: As spTCAM only works for lookup, given the limited capacity of spTCAM, the fuller the spTCAM, the better the insertion performance. There are too many rules in the original spTCAM. With limited capacity and TCAM numbers, we no longer extract the largest independent sets as layers, but take the largest independent branches, allowing all nodes whose formed chain lengths do not exceed C to be put together. The residual rules are still put in spTCAM, of which the number is much smaller. The larger C, the fewer residual rules are put in spTCAM. The number of cross-partition rules in spTCAM is fixed, so we can adjust C to fill up spTCAM when initialization.

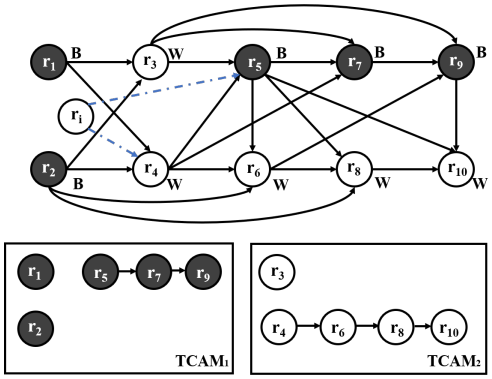


Fig. 6: Unbounded BW-split chain

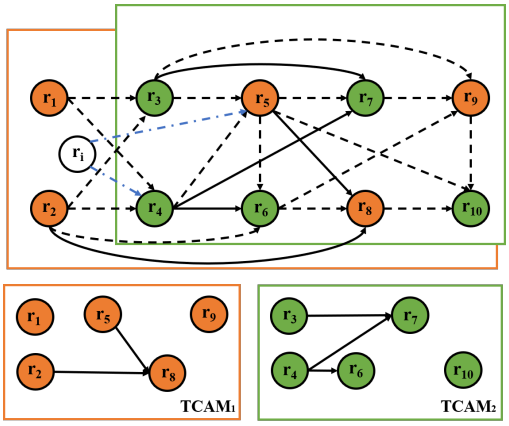


Fig. 7: Bounded MagicTCAM chain

Bounded chain length also helps MagicTCAM further reduce the insertion cost with the well limited dependencies. It outperforms BW-split, which tries to break dependency by coloring rules in black and white. BW-split first color rules without ancestors in black. For the residual rules, they are colored in a color opposite to the majority of their direct ancestors. Black rules and white rules are placed in different TCAMs. From Fig. 6 and Fig. 7, we can see that if r_i is to be inserted, the expected least cost in BW-split is 3: $r_i \rightarrow r_5 \rightarrow r_7 \rightarrow r_9$. while in MagicTCAM, only 2 movements are needed. The worst case in BW-split can be very large in some cases, which can be reduced by our rule placement which bounds chain length.

Strictly bounded chain length is mainly maintained when initialization. As the rules insertion number increases, the chain length will inevitably exceed the bound. To reduce the placement cost in single rule insertion while avoiding large chain length increment, we heuristically place new rules in DataTCAM with the least insertion cost. Inter-TCAM movement is proposed to limit the number of moves in each bucket not to exceed C. Although the chain length will exceed the bound, the inter-TCAM movement can reduce the insertion cost in most cases.

2) Inter-TCAM Movement: Set the rule movement cost value C in a bucket is 2, and the placement of rules in T_{1A} and T_{2A} is shown in Fig. 8. The rule dependency graph is

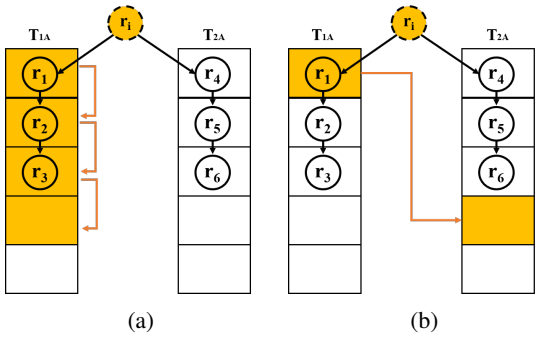


Fig. 8: The power of inter-TCAM movement

also depicted. When the new rule r_i comes, it can be placed in either of T_{1A} and T_{2A}. The insertion movement costs in both buckets are the same and r_i is to be inserted into T_{1A}. Its optimal movement path is $r_1 \rightarrow r_2 \rightarrow r_3$ as shown in Fig. 8 (a). However, the least movement cost in T_{1A} is 3, which exceeds C.

Instead moving along the optimal path, we kick out the first node r_1 on the path to make room for r_i and find the optimal insertion for r_1 in T_{2A}. As r_1 does not overlap with rules in T_{2A}, it can be inserted into T_{2A} with 0 movements as shown in Fig. 8 (b). Finally, the insertion of r_i only needs 1 movement, which is greatly improved. The insertion cost of r_i is changed to $1+cost(r_1)$, which can be abstracted as $1+cost_{avg}$. Many bad cases can be avoided.

During the process of inter-TCAM moving, once the optimal insertion movement cost does not exceed C in a bucket, the insertion is successful. Otherwise, we continue to perform inter-TCAM movement until a successful insertion or the number of iterations is larger than K. In rare cases, too many iterations are needed, then the inter-TCAM moving fails and the rule is placed in the original TCAM bucket. By moving among TCAMs, the growth rate of chain length is limited, which means a lower average movement cost. Meanwhile, entries can be fast freed for insertion.

Inter-TCAM movement also helps reduce reorder cases. Supposing that there are two rules r_a, r_b in TCAM and $r_a.prio > r_b.prio$, $A(r_a) > A(r_b)$. Originally there is no dependency between r_a and r_b, but due to the insertion of r_n, the rules r_a and r_b form an indirect dependency. According to the Condition 1, there should be $A(r_a) < A(r_n) < A(r_b)$. FastUp [28] proposes an efficient solution to reorder cases. In FastUp, the rules can be moved in any direction. Firstly the reorder is eradicated by moving r_a up or r_b down, and then r_n is inserted. However, not all insertion algorithms have solutions for the case. The reorder can be reduced by kicking r_b to other TCAMs, thus any insertion algorithm with our MagicTCAM architecture can deal better with the reorder case. Once reorder cases are detected, inter-bucket movement is triggered as well, so that the order of rules can be adjusted to avoid difficulty in updating.

3) Rule Replication: Before optimization, the cross-partition rules are placed in spTCAM. Since they have a large match fields domain, long dependency chains are easily

formed, making the update cost in spTCAM large. To avoid huge insertion cost, we put the cross-partition rules in all the partitions they locate when updating. Replicated rules are inserted independently, choosing the best TCAM on their own. Multiple TCAMs can perform the insertion in parallel. Obviously, the insertion movement cost should be bounded as well.

Any insertion algorithm for a single TCAM can be utilized to insert rules and MagicTCAM is responsible for choosing the best DataTCAM to place rules. The general process of rule insertion is as follows.

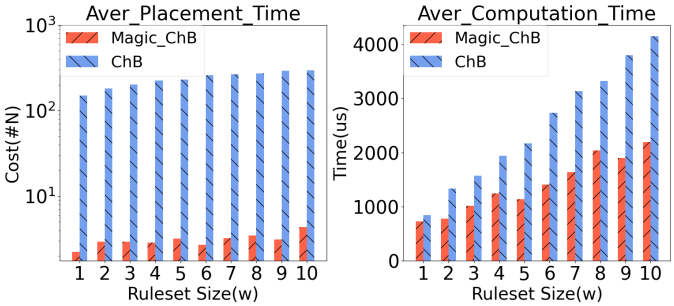
1) When a rule comes, first lookup the IndexTCAM to find in which partition it lies, so that the candidate buckets in all DataTCAMs are determined. If it crosses multiple partitions, insert all the replicates into corresponding partitions. Go to Step 2. Replicates are deemed as an independent new rule.
2) If the insertion of the new rule does not cause a reorder case, and the least cost is less than C, then go to step 4. Otherwise, go to step 3.
3) Perform the inter-TCAM movement to deal with the reordering and guarantee that movement cost in a bucket caused by a rule does not exceed C. If the accumulated number of iterations is larger than K, go to step 4, otherwise, repeat the rule kickout into other TCAMs until the number of iterations exceeds K.
4) Insert the rule into the corresponding bucket in the TCAM with the least insertion cost.

IV. Evaluation

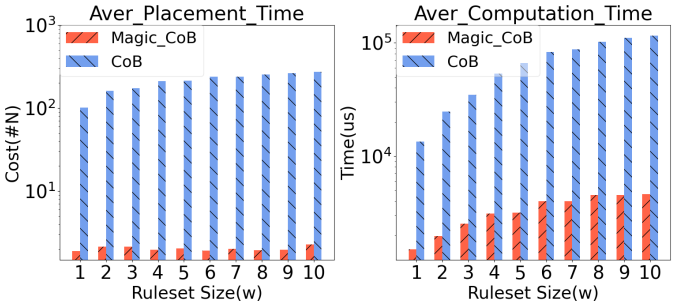
MagicTCAM is a framework whose core lies in breaking the dependency by splitting rulesets and it is compatible with any insertion algorithm in a single TCAM. We use two practical algorithms shown in PoT [25] to perform the insertion. One is cost based(CoB), which computes the moving path with the minimum length and huge computation time; and the other is chain based(ChB), which quickly selects the child node with the highest priority each time. In addition to revealing how our framework improves the performance of the insertion algorithm, we also compare it with the BW-split algorithm.

The test rulesets are Access Control List (ACL), FireWall (FW) and IP chain (IPC) with sizes ranging from 10k to 100k, which are generated by ClassBench [33]. As mentioned earlier in Section II, the cost of insertion movement is not only related to the average chain length of the dependency graph formed by rules, but also to the distribution of empty entries. In the experiment, all empty entries are placed at the bottom positions, i.e., at the high address positions.

In order to reduce the lookup suspending caused by updates, speeding up the updates should not only reduce the rule insertion move path length, but also reduce the time to make a decision about which TCAM to insert and how to insert. Therefore, our main metrics are the insertion placement time and the computation time. Placement time is measured by #N, which stands for the number of movement steps.



(a) Performance improvement on ChB



(b) Performance improvement on CoB

Fig. 9: Performance comparison for ChB and CoB with and without MagicTCAM

A. Update Cost Improvement on CoB and ChB

In testing the performance improvement of the MagicTCAM framework for the insertion algorithms, our parameters are set to 2 for the number of DataTCAMs n_t, 4 for the number of partitions n_p, 3 for the chain length bound C and 2 for the max number of iterations K. spTCAM accounts for 15% of the total TCAM capacity and DataTCAMs share the rest of the capacity equally. For the fairness of the experiment, the capacity of a single TCAM is the same as the total capacity of DataTCAMs and spTCAM. These are also the default settings in other tests.

Fig. 9 shows the performance improvement on ChB and CoB by MagicTCAM in ACL rulesets. We use Magic_ChB and Magic_CoB to represent MagicTCAM employing ChB and CoB to insert rules. As can be seen in Fig. 9 (a), compared with ChB, the average movement cost of Magic_ChB is reduced by 77 times. Magic_ChB is also 1.7 times better for computation time. As for CoB shown in Fig. 9 (b), the average movement number and the computation time are reduced by 104 and 18 times respectively. The reduction in computation time is mainly because the ruleset is layered and partitioned into smaller subsets and the sizes of rulesets for calculation are decreased.

B. Comparison with Other Ruleset Splitting Algorithms

We demonstrate the superiority of MagicTCAM by comparing it with the state-of-the-art BW-split algorithm and employing the random cutting (RC) algorithm as a baseline. 10k-80k ACL, FW and IPC rulesets are used for tests. To ensure fairness, with 2 DataTCAM and 1 spTCAM for MagicTCAM, we assign three TCAMs to BW-split and random

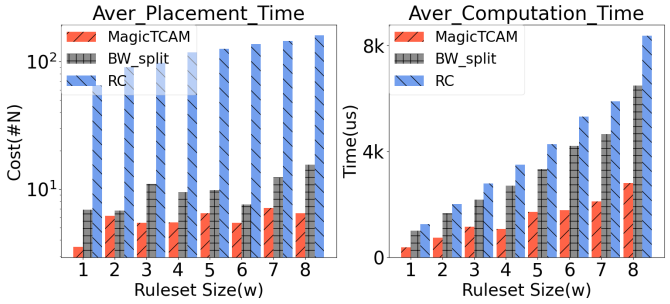


Fig. 10: Performance comparison with BW-split and random cutting in ACL Rulesets

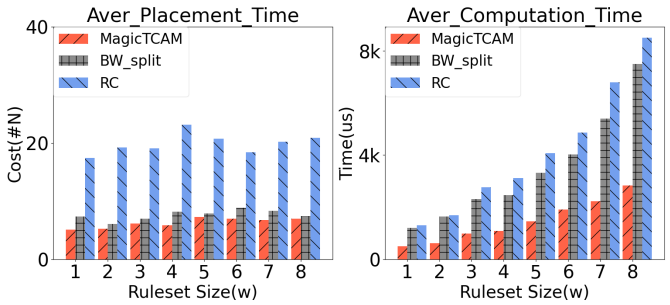


Fig. 11: Performance comparison with BW-split and random cutting in FW Rulesets

cutting with the same total TCAM capacity. BW-split renders unbalance ruleset size in TCAMs, while random cutting is balanced. All of them employ ChB algorithm to insert rules. Random cutting is to randomly divide the rule into three even parts and put them into different TCAMs. Compared to random cutting, we have a significant improvement in insertion performance. The rationale of BW-split is similar to ours, which heuristically reduces the dependencies between rules according to coloring. We still perform better in insertion, and our balanced TCAM load ratio is more practical.

Fig. 10 shows the performance comparison in ACL rulesets. Compared with BW-split, MagicTCAM reduces the average movement number by 39%, with computation time reduced 2.25 times. As for random cutting, the two metrics are shortened by 20 and 2.85 times respectively. Performance comparison in FW rulesets is shown in Fig. 11. The average movement cost is reduced by 17% and the computation time is shortened by 2.38 times over BW-split. The two metrics are reduced by 3.19 and 2.79 times for random cutting. As Fig. 12 shows, the performance in IPC rulesets is similar to that of ACL rulesets, 32% of movement cost and 2.95 times the computation time is reduced over BW-splitting. Compared with random cutting, MagicTCAM decreases average movement cost and computation cost by 16 and 3.36 times respectively. The improvement on the two algorithms in FW rulesets is the least, this is because FW has many large rules [34]. These large rules are likely to bring many dependencies and cross partitions. When new rules come, MagicTCAM will insert all the replicates. A better partitioning algorithm will be helpful to increase performance gains.

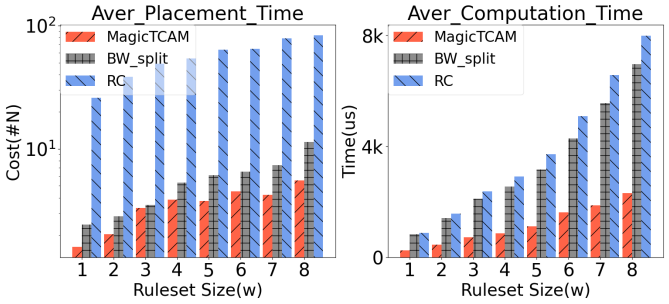


Fig. 12: Performance comparison with BW-split and random cutting in IPC Rulesets

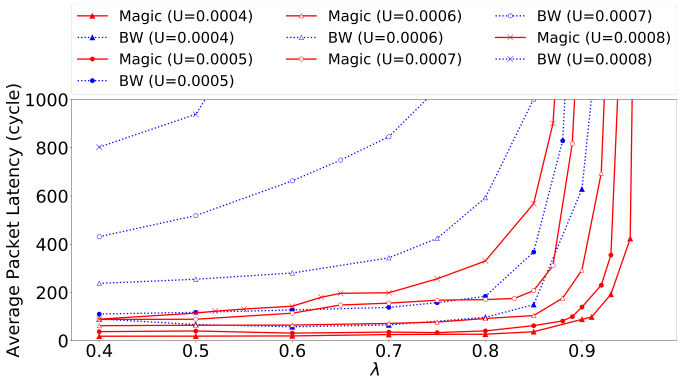


Fig. 13: Latency comparison with BW-split

Supposing TCAM runs at 500 MHz clock rate (2 ns per clock cycle) and the TCAM data bus width is 64 bits. The estimated total number of TCAM accesses for adding a 356-bit OpenFlow rule [35] is 12, loading both rule and its mask which must be installed to set the corresponding wildcard bits [36]. When a packet comes, it costs 6 cycles to load the search key (namely the corresponding packet headers). Assume the sizes of packets are 128 bytes, TCAM can support packet lookup as fast as 85 Gbps without rule updates. Setting the packet arrival rate as λ and the new rule arrival rate as U, we measure the throughput and latency of MagicTCAM with different U.

Fig. 13 shows the average packet latency when $U = 0.0004 - 0.0008$ (200k-400k new rules per second) with λ increasing from 0.2 to 1.0 (representing a packet arrival rate of 16.7M-83M PPS). We use the TCAM cycle to measure the packet latency. Compared with BW-split, MagicTCAM reduces the latency significantly. Fig. 14 shows the lookup throughput of MagicTCAM and BW-split. When $U = 0.0005$, BW-split can work well and achieve a 89% lookup throughput. While when $U > 0.0005$, its lookup throughput degrades rapidly due to the lookup suspension during rule updates. We can see that MagicTCAM performs even better with $U = 0.0008$ than BW-split with $U = 0.0006$. The development of virtualization technology enables a controller to support multiple applications, resulting in ever-increasing update rates. MagicTCAM will be an attractive option for the frequent update scenarios. Due to space constraints, we do not show the packet loss rate, which is also greatly reduced.

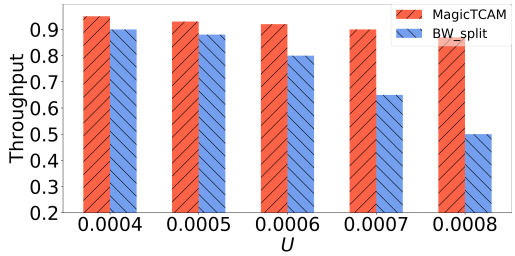


Fig. 14: Throughput comparison with BW-split

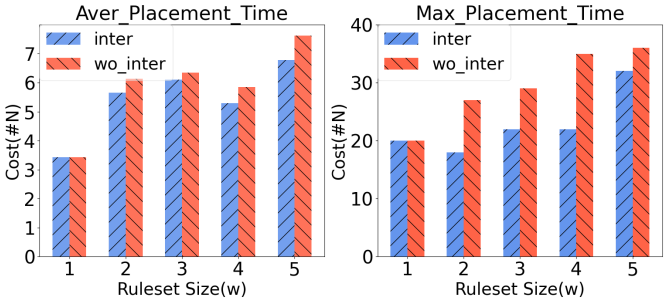


Fig. 15: Performance comparison with and without Inter-TCAM movement

We carry some further experiments on MagicTCAM to verify the tricks proposed in optimization.

C. Effect of Inter-TCAM Movement

In order to verify the effect of the Inter-TCAM movement, ACL rulesets of 10k to 50k are used. The comparison of the average insertion cost and the worst insertion cost with and without Inter-TCAM movement is presented in Fig. 15. As mentioned earlier, inter-TCAM movement is mainly aimed at bad cases, so the improvement on average movement cost is relatively small. With huge rulesets, the improvement is still attractive. Inter-TCAM movement works effectively in the worst cases, where multiple replicates are installed for a rule and some or all of them trigger reorder problems. Any of the replicates successfully utilize inter-TCAM movement will reduce the cost. The larger the max number of iterations K, the more choices for rules, and the better inter-TCAM works.

D. Sensitivity to The Number of DataTCAMs and Partitions

We use ACL rulesets with sizes ranging from 10K to 50K to test the influence of the number of DataTCAMs, that is, the effect of the number of layers on insertion performance. The total capacity of DataTCAMs is kept the same. With the increase in the number of DataTCAMs, the dependency between rules can be broken more thoroughly, and thus the number of moves required for rule insertion is reduced. The results are shown in Fig. 16. When the number of DataTCAMs is 4, the average performance is almost twice better than that of only 2 DataTCAM. The performance of 6 DataTCAMs only improves 32% and 39% on average movement cost and worst movement cost over 4 DataTCAMs. The improvement is only 14% by 2 more DataTCAMs. The benefits of increasing the number of DataTCAMs are diminishing and the performance

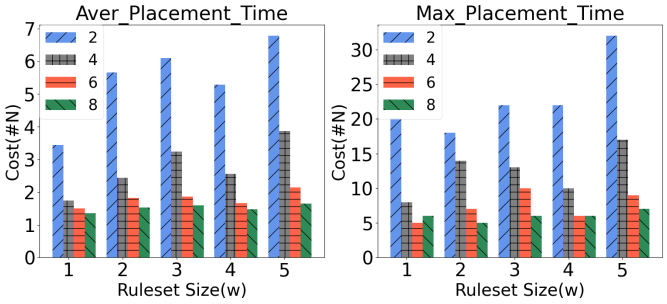


Fig. 16: Update performance with different number of DataTCAMs

TABLE I: Expansion by cross-partition rules

Sum	Ruleset Size	Trans	Rep	Trans Ratio	Rep Ratio
15729	13558	1186	2171	0.087	0.160
16772	14651	1539	2121	0.105	0.145
30305	26753	1968	3552	0.074	0.133
41858	36644	2837	5214	0.077	0.142
52544	46275	3567	6269	0.077	0.135
60991	54649	3568	6342	0.065	0.116
70816	63502	4175	7314	0.066	0.115

is good enough when 2 or 4 DataTCAMs are employed, so 2 and 4 DataTCAMs are recommended.

As for partitioning, on the one hand, it breaks the dependency among rules, on the other hand, it introduces rule replication. Replication not only brings additional TCAM space overhead, but also makes a rule need to be inserted into multiple partitions. This leads to an irregular effect of partition numbers on rule insertion. The main purpose of partitioning is to solve the problem of unbalanced rule numbers in TCAMs. In order to facilitate the balancing of the number of rules, we set the number of partitions to 4.

E. Replicate Ratio

Partitioning introduces additional replicates, and the replicate ratio has been verified on ACL rulesets ranging from 10k to 60k. The results are shown in Table I. The number of cross-partition rules accounts for 6% to 10% of the total number of rules. The final rule expansion rate is 11% to 16%. Per cross-partition rule has fewer than 2 replicates, this is because some cross-partition rules are installed in spTCAM during initialization. Only new rules which cross partitions have to be replicated. Compared to our performance improvement, we think the replicate is an acceptable sacrifice. Our partitioning algorithm is rather simple, and a better algorithm can be utilized to reduce the replicate. We leave this for future work.

F. TCAM Balance

BW-split faces a severe unbalance in TCAM space load. Three subsets are generated by 2 iterations of black and white coloring. They are placed in three TCAMs which we call black TCAM, white TCAM and green TCAM here. It is difficult to maintain complete same load all the time, and we demonstrate that MagicTCAM shows a good balance.

As shown in Fig. 17, the difference in the TCAM space load of BW-split can reach up to 19.4 times between the largest

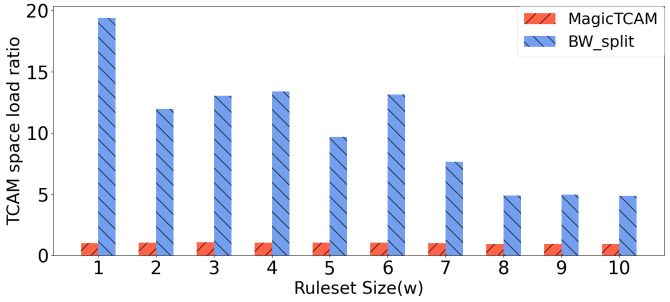


Fig. 17: Balanced MagicTCAM vs Unbalanced BW-split

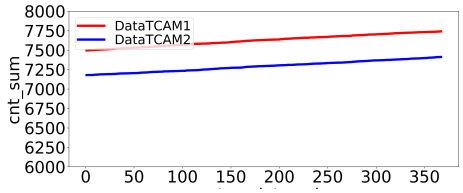


Fig. 18: TCAM load changes with rule insertion

one and the smallest one. In contrast, the ratio of the number of rules in MagicTCAM is controlled to be around 0.94 to 1.07. As the rules continue to come, Fig. 18 shows that the number of rules inserted into DataTCAM1 and DataTCAM2 is basically the same. Therefore, under our framework, the number of rules in TCAM is always well balanced.

V. Related Work

A. Insertion Algorithm in Single TCAM

Recently, there has been a lot of work studying how to insert rules quickly and correctly in a TCAM. PoT [25] applies partial order theory to derive fundamental constraints on any rule ordering on TCAMs. In PoT, dynamic programming is used with the least movements but huge computational overhead. Instead of the $O(n^2)$ dynamic programming solution, FastRule applies a greedy algorithm and Bit Indexed Tree to calculate the best choice with time complexity of $O(c_{avg}(logn)^2)$, where c_{avg} means the average diameter in the dependency graph. Employing Sequential Stack-based Algorithm, Fastup [28] further shortens the computation time and the average insertion movement cost, and the overlooked reorder problem is solved. MagicTCAM can cooperate with any of them.

B. Multiple TCAMs for Update Acceleration

Works applying multiple TCAMs for update acceleration include TreeCAM [37], Hermes [27] and BW-split [25]. The core idea of the TreeCAM and Hermes update schemes is: the insertion time is proportional to the number of entries that must be moved and they can bound the insertion time by limiting the number of rules in the flow table. Instead of limiting the ruleset size, BW-split focuses on breaking the dependency among rules.

TreeCAM proposes dual versions of decision tree: a coarse tree with a few thousand of rules per leaf and a fine tree version with tens of rules per leaf, which are maintained in TCAM and slow control memory respectively. The authors leverage the fine tree to reduce the TCAM's update cost in the coarse tree subarray. The rationale is that a leaf's rules have to be ordered in the TCAM only among each other but not with other leaves' rules. A leaf is similar to our bucket, but rules in a leaf are consecutive in match domain and can not be split freely, while MagicTCAM can utilize layering to break dependencies. In addition, due to the small granularity of the fine tree, it brings a large number of rule replicates, which is 35% more overhead [25].

As to Hermes, it bounds insertion time by restricting the size of the flow table. In Hermes, a monolithic TCAM table is carved into two tables: a small table (in size) that's called the shadow table and a full sized table called the main table. The shadow table is kept relatively empty and all the rule insertion/modification requests are served by it. From the perspective of these requests, the TCAM is small and mostly empty, which means only a few steps are needed when inserting a new rule. However, it supports a limited insertion arrival rate, which may be unsuitable for applications with frequent update. What's more, it needs precise prediction to avoid shadow table overloading.

Different from TreeCAM and Hermes, BW-split cuts the rule set while focusing on simplifying rule dependencies. The authors believe that the high cost of updates is caused by the order constraints that stem from rule overlapping. These overlapping relationships correspond to the edges in the overlap graph. Cutting these edges minimizes the number of constraints between the rules in each subset. Despite the NP-hard cutting problem which can be modeled as MAX-CUT, they use a simple heuristic to color rules in black and white. Although BW-split has greatly improved the average update cost by reducing rule dependencies, it fails to notice the unbalance in the number of black and white nodes.

VI. Conclusion

In this paper, we propose a novel multiple-TCAM framework, called MagicTCAM, to efficiently address the issue of terrible rule insertion cost via two rounds of ruleset splitting (layering and partitioning), delicate sub-ruleset placement (rotating) and inter-TCAM movement. The rationale of MagicTCAM is to divide the ruleset into subsets with as least overlapping relationship as possible, thereby reducing the number of moves required per rule insertion. We also propose to use spTCAM to store sub-rulesets whose dependencies are hard to break, protecting DataTCAMs from large insertion cost.

Testing rulesets with various sizes, we show that MagicTCAM performs well in update and balance: 1) Compared to the state-of-the-art rule splitting algorithm BW-split, MagicTCAM reduces the rule movements by 39% on average, with computation time shortened by 2.25 times. 2) Insertion algorithms employing the MagicTCAM architecture can improve both the computation delay and placement delay. 3) As rules continue to come, the ruleset sizes in each TCAM keep balance. With an increasing update rate and stricter latency requirements, MagicTCAM will likely be an attractive option.

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

[3] J. Zheng, Q. Ma, C. Tian, B. Li, H. Dai, H. Xu, G. Chen, and Q. Ni, "Hermes: Utility-aware network update in software-defined wans," *IEEE 26th International Conference on Network Protocols (ICNP)*, pp. 231–240, 2018.

[4] 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.

[5] Z. Wang, A. Singhal, Y. Wu, C. Zhang, H. Che, H. Jiang, B. Liu, and C. Lagoa, "Holnet: A holistic traffic control framework for datacenter networks," *IEEE 28th International Conference on Network Protocols (ICNP)*, pp. 1–12, 2020.

[6] V. Heorhiadi, S. Chandrasekaran, M. K. Reiter, and V. Sekar, "Intent-driven composition of resource-management sdn applications," *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 86–97, 2018.

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

[8] Y. Jiang, Y. Cui, W. Wu, Z. Xu, J. Gu, K. Ramakrishnan, Y. He, and X. Qian, "Speedybox: Low-latency nfv service chains with cross-nf runtime consolidation," *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 68–79, 2019.

[9] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, "Reinforce: Achieving efficient failure resiliency for network function virtualization based services," *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 41–53, 2018.

[10] L. Zhou, C.-H. Chou, L. N. Bhuyan, K. Ramakrishnan, and D. Wong, "Joint server and network energy saving in data centers for latency-sensitive applications," *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 700–709, 2018.

[11] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng, "Sentinel: Failure recovery in centralized traffic engineering," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1859–1872, 2019.

[12] 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 Journal on Selected Areas in Communications*, vol. 37, no. 5, pp. 968–981, 2019.

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

[14] 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.

[15] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pp. 15–26, 2013.

[16] 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 *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 179–188.

[17] "Sdn security considerations in the data center," https://opennetworking.org/wp-content/uploads/2013/05/sb-security-data-center.pdf.

[18] H. Pan, Z. Li, P. Zhang, K. Salamatian, and G. Xie, "Misconfiguration checking for sdn: Data structure, theory and algorithms," *IEEE 28th International Conference on Network Protocols (ICNP)*, pp. 1–11, 2020.

[19] R. Baheti and H. Gill, "Cyber-physical systems," *The impact of control technology*, vol. 12, no. 1, pp. 161–166, 2011.

[20] 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.

[21] D. Sattar and A. Matrawy, "An empirical model of packet processing delay of the open vswitch," *IEEE 25th International Conference on Network Protocols (ICNP)*, pp. 1–6, 2017.

[22] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "vcrib: Virtualized rule management in the cloud," *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.

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

[24] 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.

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

[26] 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.

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

[28] 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 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1175–1176, 2020.

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

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

[31] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended tcams," *11th IEEE International Conference on Network Protocols*, pp. 120–131, 2003.

[32] "Nexus 9000 tcam carving," https://www.cisco.com/c/en/us/support/docs/switches/nexus-9000-series-switches/119032-nexus9k-tcam-00.html.

[33] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM transactions on networking*, vol. 15, no. 3, pp. 499–511, 2007.

[34] W. Li, X. Li, H. Li, and G. Xie, "Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2645–2653.

[35] X.-N. Nguyen, "The openflow rules placement problem: a black box approach," *Université Nice Sophia Antipolis*, 2016.

[36] Z. Wang, H. Che, M. Kumar, and S. K. Das, "Coptua: Consistent policy table update algorithm for tcam without locking," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1602–1614, 2004.

[37] B. Vamanan and T. Vijaykumar, "Treecam: decoupling updates and lookups in packet classification," *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, pp. 1–12, 2011.