Efficient and Consistent TCAM Updates

Bohan Zhao*†, Rui Li*†, Jin Zhao*† and Tilman Wolf‡
*School of Computer Science, Fudan University, China
†Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China
‡Department of Electrical and Computer Engineering, University of Massachusetts Amherst, USA
{bhzhao16, rli16, jzhao}@fudan.edu.cn, wolf@umass.edu

***Abstract*—The dynamic nature of software-defined networking requires frequent updates to the flow table in the data plane of switches. Therefore, the ternary content-addressable memory (TCAM) used in switches to match packet header fields against forwarding rules needs to support high rates of updates. Existing off-the-shelf switches update rules in batches for efficiency, but may suffer from forwarding inconsistencies during the batch update. In this paper, we design and evaluate a TCAM update optimization framework that can guarantee consistent forwarding during the entire update process while making use of a layered TCAM structure. Our approach is based on a modified-entry-first write-back strategy that significantly reduces the overhead from movements of TCAM entries. In addition, our approach detects reordering cases, which are handled using efficient solutions. Based on our evaluation results, we can reduce the cost of TCAM updates by 30%–88% compared to state-of-the-art techniques.**

***Index Terms*—software-defined networking, ternary content addressable memory (TCAM), batch update, reordering**

I. Introduction

Software-defined networks (SDN) use logically centralized controllers to route traffic at the granularity of individual flows or flow aggregates by installing forwarding rules on SDN switches. This flexible control requires frequent updates to the flow table in SDN switches during normal operation as well as during network reconfiguration [1], failure recovery [2] and topology change [3].

High-performance SDN switches need to match the header fields of every packet against a large number of forwarding rules. Ternary content-addressable memory (TCAM) allows for parallel lookup of header fields against all stored rules in one operation and supports the use of wildcard values to specify value ranges. Therefore, TCAMs are ideal hardware for implementing the rule matching operation that is performed on SDN switches.

Though fast for lookups, TCAM is less efficient for updates. Since a packet header may match multiple rules (due to the use of wildcards) but the TCAM can only return one result for a lookup, there is a priority order among the rules in memory and those stored at higher addresses win. This priority order means that an update to the forwarding rules in a TCAM can trigger a whole sequence of operations (e.g., movement of existing memory entries to make room for a new rule at the right priority order). Thus, a TCAM can match the packet header against all rules in a single operation, but the update of a single rule may take thousands of operations.

What is worse is that updates to the rules in a TCAM can lead to routing inconsistencies. As memory entries in the TCAM are added and removed during updates, packet forwarding operations may continue to avoid long queuing delays or packet drops in the data plane. Depending on the order in which update operations are performed, packets may be forwarded incorrectly as their headers are matched against rules with the wrong priority or outdated rules. This forwarding inconsistency is a critical flaw in state-of-the-art TCAM update techniques, which has been observed in off-the-shelf SDN switches [4–6].

Therefore, there are two important aspects to TCAM updates: (1) the speed and cost at which updates can be performed and (2) if the update process maintains forwarding consistency. Our work addresses both.

Despite the abundance of previous efforts on this topic, none of the existing techniques supports consistency-preserving TCAM updates efficiently. Some work proposes to minimize the size of the ruleset in order to improve the space utilization in TCAMs [7–9]. Other work aggregates rule updates into batches in the control plane and then reduce update cost in the data plane by sorting or grouping rules [10–13]. Yet other work splits the ruleset into smaller sub-rulesets and uses multiple TCAM banks to store these sub-rulesets [14, 15], which reduces the scale of the ruleset and thus speeds up TCAM update.

The concept of rule dependency was proposed by Shah and Gupta in [16] and there are methods that aim to design new firmware in switches to reduce TCAM update cost with efficient algorithms [17, 18]. The dependency relationship of rules, which can be represented by a directed acyclic graph (DAG), enables us to avoid unnecessary writing cost by only moving dependent rules when inserting a new rule. He et al. [15] use partial order theory to abstract the update process to recursive insertion. Based on these methods, Chen and Benson presented Hermes [19], which further improves update in batches by inserting rules' effective fragments at the top of the TCAM during updates. However, this method exhibits a significant delay during which forwarding inconsistencies may appear.

Thus, efficient TCAM updates that also maintains consistent forwarding during the entire update process remains an open problem that has not been addressed well in existing work.

In this paper, we present a framework that we call Consistency-preserving Optimization framework with Layered TCAM cache for Asynchronous updates (COLA). COLA is a part of the firmware between the switch OS and the TCAM

API and can handle temporary inconsistencies between the control plane and the data plane due to update delays.

The specific contributions of our work are:

1) We present a new batch-update method to improve the efficiency of TCAM update. COLA maintains a virtual TCAM to track the insertion of rules and marks all modified entries. When writing back rules, COLA rewrites all these marked entries and employs a modified-entry-first (MEF) strategy to reduce writing cost.
2) We achieve consistent forwarding throughout the batch-update process. We design a layered TCAM structure for COLA to store temporary rules, which maintain the semantic correctness of the TCAM throughout the update process. And we propose a special writing back algorithm to further ensure the consistency during migration.
3) We propose a new solution to the "reordering case" in TCAM update, where traditional update algorithms may get trapped. Our experiments show this reordering case occurs at a frequency of 13% on average and causes potential forwarding errors. Our solution in the context of COLA outperforms existing approaches.

The remainder of this paper is organized as follows: We introduce the TCAM update problem and the design space of relevant algorithms in Section II. In Section III, we describe the principle of asynchronous batch updates and the framework of COLA. We also introduce a new write-back algorithm based on the MEF strategy of COLA. In Section IV, we discuss the reordering case and our new solution in COLA. In Section V, we evaluate COLA experimentally. We summarize and conclude our paper in Section VI.

II. TCAM Update Problem and Strategies

This section introduces a more formal definition of the update problem and presents some general strategies.

A. Naïve TCAM Update

Packet classification requires the matching of packet header fields to rules that determine how to handle that packet. Typical header fields include link layer, network layer, and transport layer header (e.g., Ethernet, IP, TCP), but may also include application layer protocol fields (e.g., HTTP).

TCAMs can perform a comparison of header fields with *all* rules stored in memory in a single operation. Therefore, classification with TCAMs can achieve very high lookup speeds. A challenge, though, is that a packet may match multiple rules in memory because of ternary match fields. TCAMs are designed to handle multiple matches by giving preferences to rules with higher physical addresses. Thus, the *rule order* in TCAM memory is of critical importance to ensure correct implementation of forwarding policies.

When updating the ruleset in a TCAM, the need for maintaining a priority order that leads to correct forwarding may require several time-consuming operations. Consider a rule sequence R that contains N rules, ordered from low priority to high priority (with ascending address) and denoted as $\{r_1, r_2, ..., r_N\}$. To insert a new rule r into the TCAM, a naïve method is to shift up all rules of higher priority than r by one TCAM entry to create an empty space for r (similar to insertion sort), which requires $O(N)$ movements. On a practical system (e.g., 400 MHz TCAM, 10K entries), a single movement costs 2.5 nanoseconds and the entire update may require up to 25 microseconds in the worst case. This overhead is equivalent to forwarding nearly 500 64-byte packets on a 10 Gbps link. Buffering such a large number of packets may be impractical, especially when updates occur frequently.

B. TCAM Update Based on Rule Dependency

Improvements to the naïve update methods have been proposed by Shah and Gupta [16]. In their work, they observe that to maintain forwarding correctness only those rules need to be shifted that r covers (recursively). We define that r_A covers r_B (or r_B depends on r_A) if all of their match fields overlap and r_A has higher priority. We denote this relationship as $r_B \rightarrow r_A$ in the remainder of this paper. The physical address of rule r and the priority of r is denoted as $Phy(r)$ and $Prio(r)$, respectively. The proposition put forward by Shah and Gupta can be easily proved since two rules could match the same packet only if they have a dependency relationship with each other. Therefore, if we assure the relative position between rules with dependency relationship is correct, all packets are matched properly after an update.

This dependency relationship between rules has been shown to be a partial order relationship in [15]. That means the dependency relationship is transitive: a rule r could not lie on a higher physical address than rule r' if there exists r'' satisfying $r \rightarrow \cdots \rightarrow r'' \rightarrow \cdots \rightarrow r'$.

The partial order relationship theory inspires us to use a Directed Acyclic Graph (DAG) to describe the dependency of the ruleset. If we regard each rule as a vertex and the dependency relationship as a directed edge, we can transform R into a DAG and any topology sorting of this DAG is a correct rule sequence (because the topology sorting of a DAG guarantees to maintain the order between any vertices connected by edges).

Algorithm 1: RecursiveInsert (r,R)

```
1 $Inf(r) \leftarrow FindInf(r, R)$
2 $Sup(r) \leftarrow FindSup(r, R)$
3 $r' \leftarrow FindRule(Inf(r), Sup(r))$
4 $WriteTCAM(Phy(r'), r)$
5 if $r' \neq \emptyset$ then
6     RecursiveInsert ($r'$ ,R)
7 end
```

Algorithm 1 shows the general update process of inserting a rule based on the DAG. We define the rule of highest physical address depending on r as $Inf(r)$, and the rule of lowest physical address that r depends on as $Sup(r)$. The algorithm ends when there is an empty entry in $(Phy(Inf(r)), Phy(Sup(r))]$ or rule r depends on nothing (so we can directly write r on the empty TCAM entry of lowest physical address). This algorithm outperforms naïve insertion because there is no need to shift up all rules of higher priority. Finding $Inf(r)$ and $Sup(r)$ can be done in $O(1)$ operations if we store rules in an efficient data structure(e.g., DAG). Therefore, the critical issue influencing update efficiency is the $FindRule(Inf(r), Sup(r))$ operation, for which we can use different strategies.

C. TCAM Rule Selection Strategies

The update algorithm based on rule dependencies includes two parts that contribute to the total time cost: the time to perform the shifting of rules in TCAM and the time to calculate which rule to shift. An ideal selection strategy (i.e., $FindRule(Inf(r), Sup(r))$ operation) should minimize the total time cost. The strategy needs to operate in an online fashion, where we do not know which rule is updated next. Previous work has presented the following update strategies:

1) **Supremum Strategy** [15]: When recursively inserting a rule r, we can simply heuristically select $Sup(r)$ to swap out. The strategy has an $O(N)$ time complexity. In the worst case, each rule in the TCAM depends on the next rule; we then have to shift all rules up to insertion point (i.e., the recursive insertion degenerates into the naïve insertion). For computation, we need to find $Sup(r)$ for $O(N)$ rules, which can be done with a cost of $O(1)$ for each. However, as we fix the swapped rule to $Sup(r)$, some better choices may be ignored.
2) **Greedy Strategy** [17, 18]: First we define the total movement cost of selecting a rule r in the TCAM to swap out in a specific rule sequence R as $C(r, R)$. If we limit the selection range to $(Phy(r), Phy(Sup(r))]$ in every recursion, the rule selection will become a problem of greedy-choice property. Therefore, before every update, we can precompute the movement cost of swapping out each rule: $C(r, R) = Min(C(r_1, R), C(r_2, R), ...) + 1$, where $Phy(r_i) \in (Phy(r), Phy(Sup(r))]$. Then we get a locally optimal solution by choosing the rule of lowest $C(r, R)$ to swap out. This strategy was first proposed in RuleTris [17] with an $O(N^2)$ implementation. In FastRule [18], the authors reduce the time complexity to $O(C_{avg}N)$, where C_{avg} is the average diameter of subgraphs in the DAG.

Comparing the above two strategies, the greedy strategy usually performs better because of the larger selection range, but it costs more time to calculate. Since both strategies are heuristics, their efficiency is not stable for different rulesets. In Section V, our experiment shows that the supremum strategy may win out in some circumstances.

D. Update in Batches

To further decrease the update cost from moving entries, a well-known approach is to collect incoming rules in a batch and then write them into TCAM together at one time. The general process of batch update is shown in Fig. 1. This batch processing reduces entry movements because we can precompute the final state of the TCAM and omit unnecessary movements that would occur during incremental updates. Fig. 2 shows an example of this advantage. Batch update can be implemented in either the control plane (as policy compiler extension in the controller) or the data plane (as firmware extension in switches) or even both.

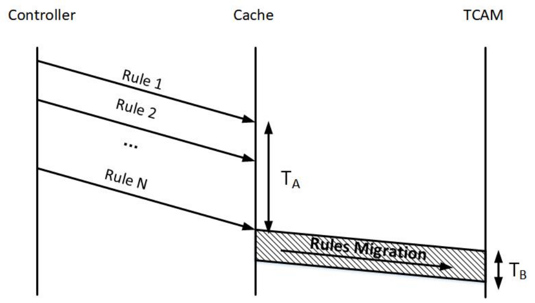


Fig. 1. A general process of update in batches. First, rules generated by the controller are stored in a cache (in the control plane or data plane). We define this period as T_A. After the cache is filled up, the scheduler writes back rules in the cache to TCAM. We define this period as T_B.

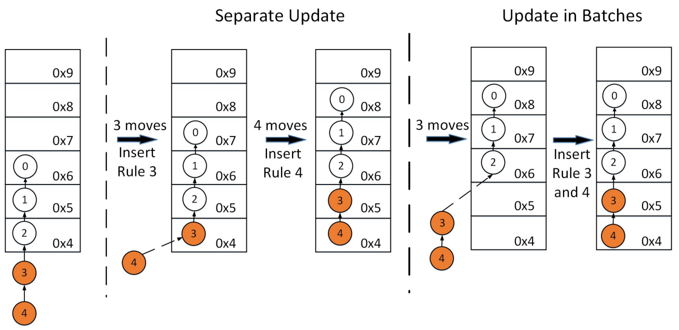


Fig. 2. An example to show the advantage of update in batches. Suppose that we are planning to insert rules 3 and 4 into TCAM. If we insert 3 and 4 step-by-step, we have to shift $0-2$ up by one entry to create an empty space for 3, and then shift $0-3$ for 4. The total movement overhead of is 7. However, if we insert 3 and 4 together, we can shift $0-2$ up by two entries at one time and the total movement overhead is only 3.

Batch update in the control plane can minimize the size of the ruleset sent to switches. ESPRES [12] groups a batch of update operations into several sub-updates to execute them in parallel. Tango [11] changes rule installation order to reduce different update cost. Mazu [10] redistributes stored rules in different switches to speed up update. FLIP [8] also addresses the rule consistency problem in the control plane.

RuleTailor [13] seeks to speed up batch update in data plane through rule transformations. Measurement studies [4–6] also revealed that several commercial off-the-shelf SDN switches perform updates in batches. In general, batch update in data plane includes two phases: rule accumulation T_A and rule migration T_B, as shown in Fig. 1. However, its main drawback is that forwarding correctness of a TCAM is violated during both T_A and T_B because of the delayed update and modified order (i.e., some rules that were supposed to be inserted into the TCAM are still in the cache). Kuzniar et al. [6] show a significant duration of such inconsistencies by measuring the delay between the control plane confirmation time and when data plane probes are affected, which was up to 3 milliseconds per rule. This is a critical problem that may cause forwarding errors, such as evil loops and black holes.

To address this inconsistency problem, Hermes [19] seeks to insert new rule's effective fragments on the top of the TCAM when accumulating rules, which in turn reduces the time of inconsistency during period T_A. While this preserves consistency in T_A, it is highly inefficient in update calculation because Hermes uses batch update strategies similar to the control plane counterparts [10–12]. Meanwhile, even if Hermes improves the issue of migration inconsistency by deleting existing rules after a new rule covering them is inserted, there still exists inconsistency during Period T_B because the packets that only match the deleted rule cannot be forwarded correctly. Since the calculation time and writing cost of Hermes in the migration phase is high (more than 1K writing operations for a 20k-size TCAM according to their experiment), there remains a significant possibility for TCAM inconsistency that causes forwarding errors.

Here, we ask is it possible to achieve both efficiency and consistency throughout the batch update process? To this end, we propose a consistency-preserving optimization framework with layered cache for asynchronous TCAM updates.

III. The Architecture of COLA

Motivated by the advantages of batch updates, we present COLA, our TCAM update optimization framework. COLA includes a new update strategy, named Modified-Entry-First (MEF), a layered TCAM structure for update consistency guarantee, and an asynchronous update pattern based on batch update. COLA enables us to tackle the update performance bottleneck of general TCAM update algorithms and, more importantly, avoid the inconsistency in both accumulation phase and migration phase.

A. Asynchronous Updates in Batches

Fig. 3 shows the architecture of COLA. COLA includes a policy compiler, a TCAM update scheduler, and a specialized logical TCAM structure. The policy compiler receives rule update requests from the switch OS and converts them into node insertion requests for the DAG. The TCAM update scheduler gets information from the DAG about the inserted rules and all dependency requirements that they must satisfy. After that, the scheduler detects if there are any reordering cases, inserts the rules in a software virtual TCAM, generates corresponding temporary rules, and calls the TCAM API to apply COLA's update strategy in the physical TCAM.

In COLA, we propose a new method for batch updates named asynchronous update: When a new rule arrives, COLA inserts the rule in a virtual TCAM in the software plane and installs its temporary rules in a special space of the physical TCAM to guarantee forwarding correctness. When that space is full, COLA writes modified entries back into the physical TCAM according to the virtual TCAM. During asynchronous update, the status difference between the virtual and real TCAM is transparent beyond the TCAM. There is no need to clear rules in the special space when writing back for efficiency. We record the real number of rules in this space and cover the entries when inserting new temporary rules. Asynchronous update is efficient since both insertion and deletion of temporary rules have low cost and batch updates omit unnecessary movements.

The key of asynchronous update is how to generate temporary rules for the TCAM without violating any forwarding policy. As the TCAM gives priority to rules from higher physical addresses to lower, the relevant information a rule provides is the part that is not covered by rules at higher physical addresses. A naïve method of designing temporary rules is transforming the uncovered part of the inserted rule into sub-rules and putting them in empty TCAM space higher than all existing rules. Fig. 4 shows this process.

To implement asynchronous update, we first classify an incoming rule r into one of the three classes according to the dependency constraint in the TCAM:

- Top Rule: if $\forall r : r \rightarrow r_0$ or $r \cap r_0 = \emptyset$, r_0 is a top rule.
- Middle Rule: if $\exists r_1, r_2 : r_1 \rightarrow r_0, r_0 \rightarrow r_2$, r_0 is a middle rule.
- Bottom Rule: if $\forall r : r_0 \rightarrow r$ or $r \cap r_0 = \emptyset$, r_0 is a bottom rule.

Top rules (rule $0, 1$ in the DAG of Fig. 3) are not covered because there are no rules they depend on. Therefore, the temporary rule of a top rule is itself. Middle rules (rule $2, 3, 5, 6$ in Fig. 3) can be replaced by uncovered parts as shown in Fig. 4. For bottom rules (rule $4, 7, 8$ in Fig. 3), as they usually have large match fields (because large-field rules usually have lower priority and are more likely to be a bottom rule), the uncovered part may be wide and irregular, which would cause many sub-rule insertions. To improve this, we make room at the lowest address of TCAM to store bottom rules, specifically. Since there are no rules that depend on bottom rules, these rules can be placed anywhere in empty space of lowest addresses of the TCAM. By this method, the temporary rule for a bottom rule is itself, too.

Based on this rule classification, COLA layers the physical TCAM into three logical parts: the cache table at the highest addresses for storing temporary rules of top and middle rules, a root table at the lowest addresses to store temporary rules for bottom rules, and a main table for storing the original rules. All temporary rules can be placed directly in empty space in the rule table's direction of growth. Here, we make a trade-off between the performance improvement and space overhead of the cache and root tables. In practice, this overhead is less than 3% in our evaluation tests. And both the virtual TCAM and the data structure of DAG are implemented in $O(N)$ space complexity in the software plane, which is small even for a 2K size TCAM.

B. Practical Considerations

To support existing control-to-data-plane interface, such as OpenFlow, we discuss the management of temporary rules. We establish a mapping from original rules to their temporary rules in COLA and transform instructions for original rules into

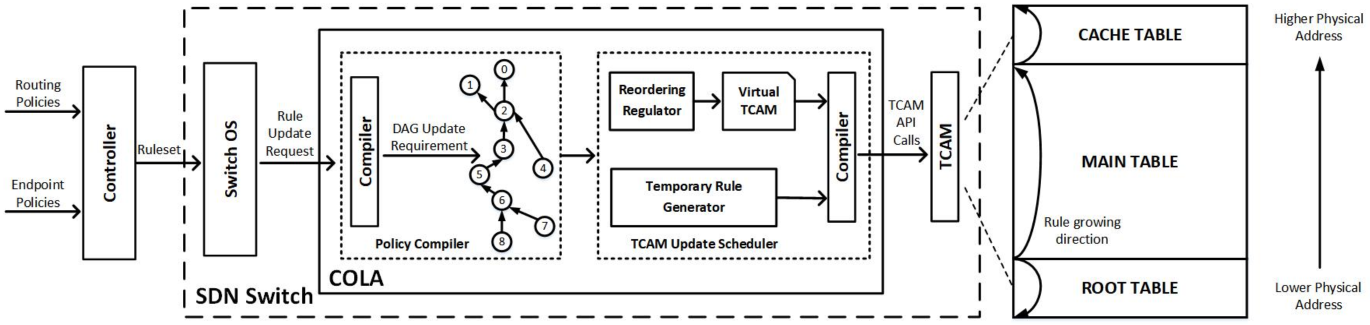


Fig. 3. COLA architecture showing the process of updating rules in batches and internal structure of TCAM.

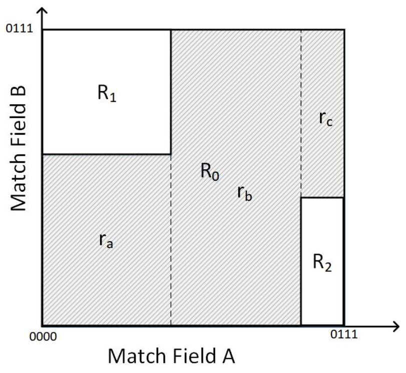


Fig. 4. An example of generating temporary rules. R_0 (0***,0***) is partially covered by R_1 (00**,01**) and R_2 (011*,00**). So, the relevant part of R_0 is the dashed area in the figure. As any area can be represented by several rules (i.e., rectangles in figure), we use sub-rules r_a(00**,00**), r_b (010*,0***) and r_c (011*,01**) as the temporary rules of R_0.

instructions for temporary rules. For example, in OpenFlow switches, if any packet hits a temporary rule, the idle timeout of all temporary rules from the same original rule will be refreshed. And the original rule's hit count (times that a rule is matched with packets) will be the sum of the hit counts of all temporary rules. A challenge occurs when a rule is changed. As rule modification and deletion may cause some covered parts of original rules in the batch to be exposed, we have to examine all these rules and modify existing temporary rules at significant cost. To avoid this cost, we write back original rules in advance and then execute modification or deletion operations directly.

In addition, as many commercial off-the-shelf switches support multiple TCAM tables to create a pipeline, COLA needs to support multiple forwarding tables. For each TCAM table, COLA shares the action of an original rule with all its temporary rules and any packet matched with temporary rules can be dropped, forwarded, or sent to the next table correctly. This approach guarantees the independence of TCAM tables. That is, the operation of COLA is transparent beyond a single TCAM and thus will not disrupt the workflow of a TCAM pipeline.

C. Write-Back Algorithm

The asynchronous update requires that after saving a batch of rules, we write all these rules back. This process also needs to be consistent and efficient. In addition, we need a way to determine the write-back time so that we can clear temporary rules in time before they fill up the cache table and root table.To further improve the write-back algorithm in both aspects, we design a new update strategy based on recursive insertion, named Modified-Entry-First (MEF).

During asynchronous update, every time a rule comes in, we insert the rule in the virtual TCAM and insert its temporary rules in the physical TCAM. MEF requires us to mark the modified entries in the virtual TCAM during this update process, and the rules in marked entries are preferred to swap out in subsequent virtual TCAM updates. If there are no modified entries in the selection range, a heuristic method is used instead (e.g., choose the $Sup(r)$ to swap out). Through MEF, we reduce the number of modified virtual TCAM entries for the update. When writing back original rules, we write those marked entries from the virtual TCAM into the physical TCAM. Fewer marked (modified) entries mean less write-back overhead. Fig. 5 shows the process of the MEF-based write-back algorithm. Compared with Hermes, our tests show that MEF is effective for updating in batches and reduces migration cost by 53% on average.

Since we update the virtual TCAM status during every insertion, it takes only $O(N)$ time complexity to calculate and write back original rules in the physical TCAM. This approach avoids the significant delay in calculating migration, which we observe in other algorithms. When writing back original rules, COLA first rewrites entries in the main table from higher physical addresses to lower (empty entries first) and then deletes temporary rules last. This order guarantees the consistency of TCAM while writing back because we always backup a rule on a higher address before deleting it. Therefore, the delay of migration is not a consistency bottleneck anymore. Considering that our temporary rules ensure consistency throughout rule accumulation, the TCAM can forward correctly during the entire asynchronous update process without being locked or forwarding errors. We will not clear temporary rules during writing back as they will not disturb the forwarding policy. Instead, we change the record

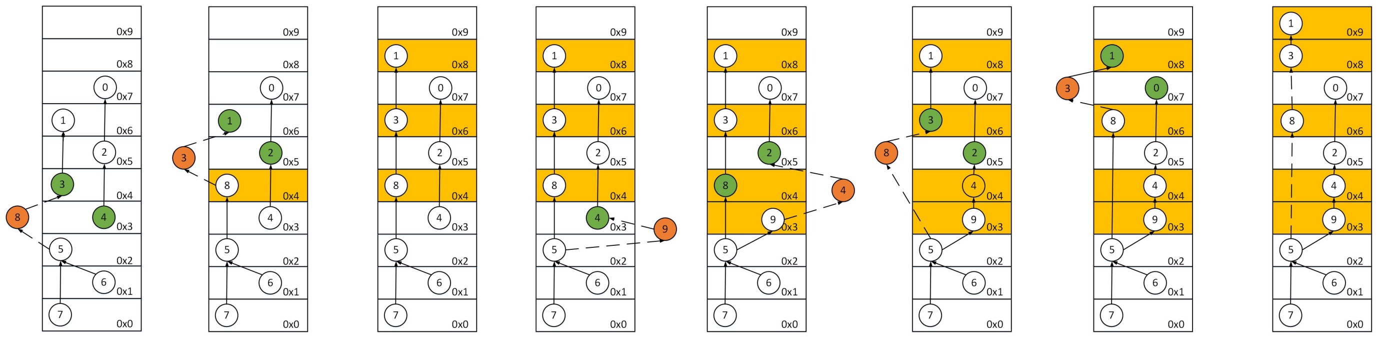


Fig. 5. The process of MEF based write-back algorithm. The orange node denotes the rule to be inserted. Green nodes denote rules in the selection range. Yellow entries denote entries that have been modified. At first our goal is to insert rule 8. Therefore we swap out rule 3 and mark entry $0x4$ as modified. Then to insert rule 3 we mark two entries: $0x6, 0x8$. Now the first inserting process ends. Next, we plan to insert rule 9. We prefer those modified entries to swap out and thus entries $0x4, 0x6, 0x8$ are selected when they are in the selection range. The total number of modified entries is 5 as shown in the last figure. When writing back, we just need to update these modified entry.

of rule number in these two small tables to zero. When next temporary rule comes in, it will be inserted into the first entry.

The other important part of the write-back algorithm is to determine when to begin writing back. First, COLA records the rule arrival rate (i.e., rules per second) of the last update cycle (i.e., time between two write back instances) as $SampleRAR$. The estimated rule arrival rate ($eRAR$) can be determined by any predictive algorithms, such as Exponentially Weighted Moving Average (EWMA) [20]. COLA also estimates the time needed to write back modified entries (not including the time to clear temporary rules) as $n \times t$, where n is the number of modified entries and t is time to write a TCAM entry.

With eRAR and the estimated write-back time, COLA can dynamically adjust the maximum occupancy of the cache and root table. We note that replacing some middle rules needs a very large number of temporary rules (e.g., hundreds of rules for a 10K TCAM, as we observe in our experiments) and instead we insert these rules directly into the main table (this process can be also consistent if executed from higher physical address to lower as mentioned before). We denote the maximum temporary rules generated from a single rule that COLA can accept as λ, which is a default constant depending on the size of the TCAM. We also denote the size of a small table (cache table or root table) as S_T. COLA guarantees that both small tables can always provide enough space for incoming rules, which means that new rules arriving while COLA writes rules back cannot fill up the remaining space of small tables. Therefore, the maximum occupancy of small tables is computed as:

$$MaxOccupancy = S_T - eRAR \times n \times t \times \lambda. \qquad (1)$$

Algorithm 2 shows the asynchronous update process used by COLA. We use R and R' to denote the rule sequence of the physical TCAM and virtual TCAM, respectively. G represents the rule DAG of the virtual TCAM, which is used to find $Sup(r)$ and $Inf(r)$ and calculate temporary rules. ϵ is a default constant; when the difference between $Occupancy$ and $MaxOccupancy$ is smaller than ϵ, COLA begins writing back original rules.

Algorithm 2: AsynchronousInsert (r,R)

```
1  ${r_T} \leftarrow CalTemporary(r, G)$
2  $WriteTCAM(r_T, R)$
3  $Occupancy \leftarrow Occupancy+ \mid \{r_T\} \mid$
4  $G \leftarrow AddNode(r, G)$
5  $RecursiveInsertion(r, R')$
6  $eRAR \leftarrow CalErar(SampleRAR)$
7  $MaxOccupancy \leftarrow CalMaxOccupancy(R', eRAR)$
8  if $MaxOccupancy - Occupancy < \epsilon$ then
9      $WriteBack(R', R)$
10 end
```

IV. An Effective Solution for Reordering Case

In the process of recursive insertion, $Phy(Sup(r)) > Phy(Inf(r))$ is a default condition for every rule r to be inserted. However, there may be exceptional cases that violate this condition and lead to consistency problems. Therefore, it is critical to detect and solve this issue during TCAM updates.

A simple, but low-performing solution to this problem was proposed in [15]. Here, we present a new, effective solution using COLA to further improve the consistency and efficiency of TCAM updates.

A. Reordering Case and Existing Solution

As mentioned in Section II-A, for any rule sequence in TCAM that achieves correct forwarding, there can exist $r, r' : Prio(r) < Prio(r'), Phy(r) > Phy(r')$. That means that if two rules are not dependent, the rule with lower priority may be stored at a higher physical address. Here, we define such a pair of rules as a reverse pair. Allowing reverse pairs could reduce the general update cost. However, as the number of reverse pairs grows, the potential for violating correctness increases.

As Fig. 6 shows, before we insert the rule r_{ins}, there is no dependency relationship between $Inf(r_{ins}) = r_1$ and

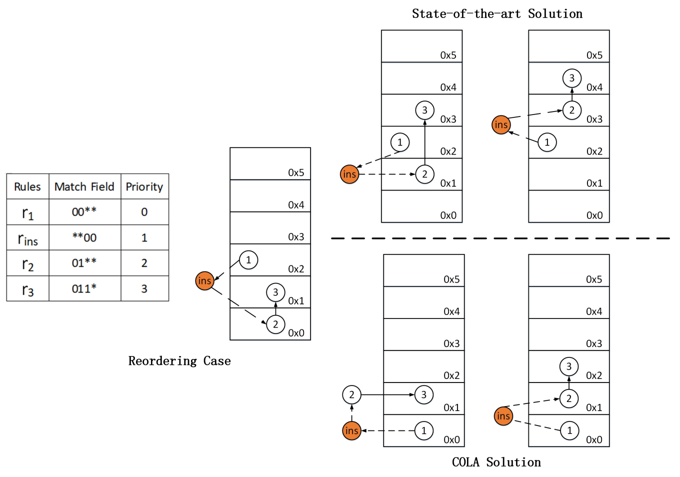


Fig. 6. An example of the reordering case and two different solutions. The existing solution in [15] keeps inserting r_{ins} into the position of $Sup(r_{ins}) = r_2$ twice until $Sup(r_{ins})$ is shifted up to a physical address higher than $Inf(r_{ins}) = r_1$. COLA's solution repeats the process: shifts r_1 down and then shifts r_2 up. After $Phy(Sup(r_{ins})) > Phy(Inf(r_{ins}))$ is satisfied, we can update r_{ins} by recursive insertion as before.

$Sup(r_{ins}) = r_2$ and the relative position between them is arbitrary. In this example, r_1 and r_2 make up a reverse pair. The inserted r_{ins} acts as a bridge between r_1 and r_2 to create a dependency relationship: $r_1 \rightarrow r_{ins} \rightarrow r_2$. This case violates the default assumption that $Phy(Sup(r)) > Phy(Inf(r))$ and is called the reordering case [15].

Ignoring reordering cases will impact the correctness of recursive insertion and cause consistency problems. For example, in the case of Fig. 6, a packet matching r_1 and r_{ins} would be forwarded according to r_1 instead of r_{ins}, which is a forwarding error.

The solution for reordering cases in [15] shifts $Sup(r_{ins})$ up by inserting r_{ins} into the position of $Sup(r_{ins})$ using the supremum strategy until the rule sequence in TCAM satisfies $Phy(Sup(r_{ins})) > Phy(Inf(r_{ins}))$ as shown in Fig. 6. This approach is very costly as it requires several repetitive insertions. For example, He et al. [15] show that in a TCAM of size 20Kfor the FW data set, one reordering case correction costs up to 1,801 movements, nearly 60 times more than the expected cost of a single update. Moreover, this solution generates empty space in low physical address, which reduces space utilization of the TCAM because not all rules can be inserted in that space.

B. An Efficient Solution for the Reordering Case in COLA

To improve the efficiency of solving reordering cases in TCAMs, we propose a new solution that reduces movement costs and avoids fragmentation. Instead of shifting $Sup(r_{ins})$ in one direction, we try to recursively shift $Inf(r_{ins})$ down and $Sup(r_{ins})$ up simultaneously. In each recursion, we define the original position of $Inf(r_{ins})$ and $Sup(r_{ins})$ as P_{Inf} and P_{Sup} respectively. To shift $Inf(r_{ins})$ down, we insert $Inf(r_{ins})$ into the position of $Inf(Inf(r_{ins}))$, and then insert $Inf(Inf(r_{ins}))$ recursively. This recursive insertion ends when the insertion rule r satisfies $Phy(Inf(r)) \leq P_{Sup}$, which means $Sup(r_{ins})$ is in the selection range of r and thus can be swapped out with r. Here we define that if $Inf(r)$/$Sup(r)$ does not exist, $Phy(Inf(r))$/$Phy(Sup(r))$ is minus infinity/infinity. Then we are going to shift $Sup(r_{ins})$ up. Similarly, we insert $Sup(r_{ins})$ into its supremum position and repeat this supremum insertion until the insertion rule r' satisfy $Phy(Sup(r')) \geq P_{Inf}$, and we insert r' into P_{Inf}, which is empty because we shift $Inf(r_{ins})$ down before. This process repeats until $Phy(Sup(r_{ins})) > Phy(Inf(r_{ins}))$. The entire solution is shown in Algorithm 3, and its correctness is shown in the following proof.

Algorithm 3: COLAReordering (r)

```
1  while $Phy(Sup(r)) < Phy(Inf(r))$ do
2      $r_{temp} \leftarrow Inf(r)$
3      $P_{Sup} \leftarrow Phy(Sup(r))$
4      $P_{Inf} \leftarrow Phy(Inf(r))$
5      while $Phy(Inf(r_{temp})) > P_{Sup}$ do
6          $WriteTCAM(Phy(Inf(r_{temp})), r_{temp})$
7          $r_{temp} \leftarrow Inf(r_{temp})$
8      end
9      $WriteTCAM(P_{Sup}, r_{temp})$
10     $r_{temp} \leftarrow Sup(r)$
11     while $Phy(Sup(r_{temp})) < P_{Inf}$ do
12         $WriteTCAM(Phy(Sup(r_{temp})), r_{temp})$
13         $r_{temp} \leftarrow Sup(r_{temp})$
14     end
15     $WriteTCAM(P_{Inf}, r_{temp})$
16 end
```

Proof: When we shift down a rule r by swapping out $Inf(r)$ recursively, $Phy(r)$ keeps decreasing in each recursion because $Phy(Inf(r)) < Phy(r)$ by our definition. Therefore, we can find a rule r without $Inf(r)$ or satisfying $Phy(Inf(r)) < Phy(Inf(r_{ins}))$ after finite recursions. Similarly, the shift-up process can be proven to be correct in the same way. During this process, $Phy(Inf(r_{ins})) - Phy(Sup(r_{ins}))$ decreases, and we can reverse their position after several iterations. ■

The upper bound of the cost to resolve one reordering case is $O(\eta^2)$, where η is the number of TCAM entries between $Phy(Sup(r_{ins}))$ and $Phy(Inf(r_{ins}))$. This is because we reduce the distance between $Sup(r_{ins})$ and $Inf(r_{ins})$ by at least 2 entries in one loop and for each loop we move at most η entries. For COLA, this process is conducted in the virtual TCAM and thus does not cause consistency problems. Moreover, as the modified entries during this process are only those between $Sup(r_{ins})$ and $Inf(r_{ins})$, COLA can further reduce this cost to $O(\eta)$.

V. Evaluation

To show that COLA is able to efficiently update TCAM while maintaining forwarding consistency, we evaluate COLA's performance in comparison to different existing update algorithms. We consider three scenarios:

TABLE I
NUMBER OF TCAM OPERATIONS TO INSERT RULE

Type	Size	Rules	COLA	SS		GS		Hermes	
			Average	Average	Worst	Average	Worst	Average	Worst
ACL	1K	941	2.34	2.72	35	2.69	42	2.89	62
	2K	1832	3.34	4.99	117	5.91	109	4.62	162
	4K	3595	2.91	3.74	112	5.08	139	3.62	192
	8K	7460	4.12	4.98	155	6.92	173	4.96	201
	10K	9439	5.46	6.42	191	7.76	194	6.36	323
	20K	18798	6.97	8.10	276	9.24	249	7.94	661
FW	1K	844	11.31	19.74	124	15.92	82	11.76	521
	2K	1734	13.86	22.77	144	18.69	128	16.44	830
	4K	3709	13.81	17.13	171	15.79	154	15.26	1177
	8K	7479	17.32	19.80	227	17.06	207	19.45	1390
	10K	9367	15.99	18.10	231	16.75	205	18.06	1447
	20K	18647	18.13	20.08	318	17.07	248	19.26	1556
IPC	1K	990	1.95	2.55	42	2.42	37	2.62	44
	2K	1928	2.81	2.96	47	3.30	58	3.21	63
	4K	3826	3.46	4.61	92	5.88	91	4.47	193
	8K	7590	3.70	4.82	114	7.10	128	4.65	373
	10K	9549	4.22	5.55	146	8.18	147	5.37	447
	20K	19457	5.93	7.20	182	10.84	173	7.07	505

1) We evaluate the general TCAM update cost (not considering the reordering case) of COLA in comparison to Hermes [19], τ_{down} [15], RuleTris [17] and FastRule [18]. COLA and Hermes both update rules in batches. RuleTris and FastRule both follow the greedy strategy (GS) but differ in calculation efficiency (the movement cost of both is the same). The τ_{down} algorithm follows the supremum strategy (SS).
2) We evaluate the reordering case solution of COLA and compare to the state-of-the-art reordering solution [15], as discussed in Section IV.
3) We execute rule update in consideration of reordering cases over different algorithms to observe the overall update performance when handling such exceptions.

A. Data Set and Evaluation Setup

We implement the TCAM update algorithms and solutions for reordering cases in C++. Similar to previous TCAM-related research, we use ClassBench [21] to generate synthetic rulesets. To confirm that our method is sufficiently robust and scalable, we use various types of rulesets: Access Control List (ACL), Firewall (FW), and IP Chain (IPC). For each type of ruleset, we vary TCAM sizes from 1K, 2K, 4K, 8K, 10K, to 20K (as TCAM are power-hungry, their size is usually limited to this range).

According to our experiments, the number of rule dependencies in FW is larger than in ACL and IPC and more extra entry movements occur when updating. This diversity in rulesets and corresponding DAG structures ensures the generality in performance.

B. Evaluation of General TCAM Update Cost

We present the update cost (i.e., number of operations in TCAM) of different algorithms to compare COLA to other state-of-the-art algorithms. Initially, the TCAM was empty, and the corresponding number of rules are sequentially inserted into TCAM using different algorithms. Table I shows the average number of TCAM movements to insert a rule using COLA, SS, GS, and Hermes for our 18 data sets.

We can see in Table I that algorithms based on GS do not always perform better than SS. Specifically, GS wins out when the ruleset exhibits more dependencies and underperforms when there are fewer dependency constraints. This is because GS tends to distribute rules tightly in the lower physical addresses and if there is any large-field rule with a relatively high physical address, numerous extra movements occur. In the ruleset with simpler DAG structures, GS's advantage of a larger selection range is not noticeable, and thus GS is more likely to underperform.

COLA and Hermes both update rules in batches, but these two frameworks apply different update strategies and differ in the logical TCAM structure. COLA uses much less overhead to represent the effective information of incoming rules. In addition, the MEF strategy of COLA matches batch updates better, thus reducing the entry write times when writing back.

COLA outperforms all other algorithms, except for GS for large FW scenarios. When comparing the average update costs, COLA outperforms the best of other algorithms by 19% in ACL, 23% in FW, and 19% in IPC on average. In total, COLA outperforms SS by 21%, GS by 27%, and Hermes by 16%. For some data sets, COLA achieves up to 48% improvement over current state-of-the-art algorithms.

To evaluate the longest inconsistency time of different algorithms, we also show the worst-case update costs. For COLA, the write-back process does not cause inconsistency and thus the duration of any forwarding inconsistency is zero (not shown in table). We can observe that the worst case grows with the size of TCAM and can reach several hundred TCAM operations. These cases may cause not only forwarding errors of substantial duration but may also violate delay requirements for updates, leading to performance bottlenecks.

C. Evaluation of Solution to Reordering Case

As described above, the reordering case is an exceptional situation that can cause correctness problems during recursive insertion. To compare our solution in COLA with the state-of-the-art (naïve) reordering solution, we conduct our experiments in two scenarios. We use the COLA reordering regulator (COLA-RR) and the naïve solution in conjunction with the two typical update algorithms, SS and GS. For both COLA-RR and the state-of-the-art solution, we evaluate the average movement cost per rule to solve reordering cases and the frequency at which reordering cases occur.

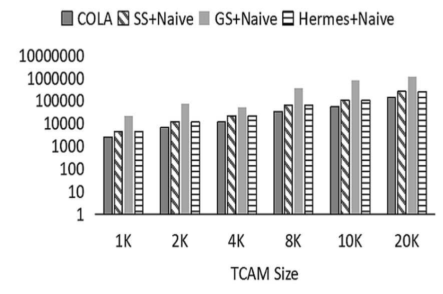
The evaluation results are shown in Table II. We can see that the reordering case occurs frequently and may incur significant additional costs of movement to resolve. COLA solves reordering cases with much fewer movements than the naïve solution: only 24% in ACL, 5% in FW, and 28% in IPC for SS; 9% in ACL, 7% in FW, and 1% in IPC for GS. COLA also reduces the frequency of occurrence of such cases by 70% on average.

D. Evaluation of Total Update Cost

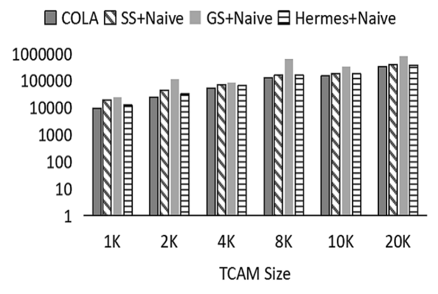
The total update cost consists of two parts: time to execute the update algorithm and time to write the physical TCAM. He et al. [15] have shown that update algorithms with time complexities of no more than $O(N)$ (N is the occupancy

TABLE II
COMPARISON OF REORDERING CASE SOLUTION PERFORMANCE BETWEEN COLA REORDERING REGULATOR AND THE NAÏVE ALGORITHM

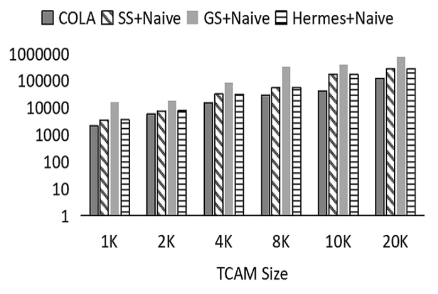
Type	Size	Rules	SS+COLA-RR		SS+Naïve		GS+COLA-RR		GS+Naïve	
			Movement	Frequency	Movement	Frequency	Movement	Frequency	Movement	Frequency
ACL	1K	941	0.41	7.33%	1.73	18.81%	0.28	5.63%	18.77	38.79%
	2K	1832	0.50	6.39%	1.54	12.66%	0.22	4.15%	31.65	12.01%
	4K	3595	0.73	6.18%	2.08	14.10%	0.19	3.98%	8.63	22.84%
	8K	7460	0.95	5.48%	3.43	11.18%	0.20	2.96%	39.26	16.22%
	10K	9439	0.66	3.74%	4.10	9.15%	0.20	2.47%	74.73	9.74%
	20K	18798	0.47	2.55%	5.17	7.79%	0.15	1.68%	48.74	8.62%
FW	1K	844	0.23	4.86%	3.10	9.48%	0.26	6.04%	12.37	15.88%
	2K	1734	0.23	3.63%	2.71	9.86%	0.30	4.27%	45.33	11.19%
	4K	3709	0.20	2.78%	2.40	10.68%	0.06	2.02%	6.67	10.35%
	8K	7479	0.03	1.00%	1.84	5.58%	0.08	1.32%	65.79	10.12%
	10K	9367	0.03	0.94%	1.27	5.89%	0.07	1.37%	18.68	9.03%
	20K	18647	0.01	0.44%	1.10	5.50%	0.03	0.63%	24.72	6.82%
IPC	1K	990	0.35	6.06%	1.01	11.72%	0.48	7.27%	14.05	28.79%
	2K	1928	0.41	5.34%	0.96	10.37%	0.14	3.16%	6.23	24.27%
	4K	3826	0.76	6.06%	3.69	12.26%	0.16	3.06%	15.54	18.48%
	8K	7590	1.05	5.74%	2.55	12.57%	0.14	2.49%	36.23	18.63%
	10K	9549	1.21	6.17%	12.56	13.02%	0.19	2.61%	34.17	25.68%
	20K	19457	1.53	5.05%	6.93	10.34%	0.11	1.73%	27.66	11.51%



(a) ACL



(b) FW



(c) IPC

Fig. 7. Total TCAM update cost of algorithms considering reordering cases. Cost (the number of movements) is shown for processing entire ruleset.

of TCAM) will not be the bottleneck of updates in current TCAMs. All update algorithms mentioned above can be executed in $O(N)$ or nearly $O(N)$ (the time complexity of GS is higher). Therefore, we only evaluate the write times of the different algorithms in the following experiments.

We compare the state-of-the-art algorithms with COLA in term of the total TCAM write costs, including insertion movements and reordering case solutions. For different update algorithms, the reordering case frequency differs and the reordering case solution redistributes rules in TCAM differently, which in turn affects the update algorithm efficiency. Therefore, it is important to assess the total update cost of algorithm combinations.

Fig. 7 shows that GS performs worst for almost all data sets due to the large cost of solving reordering cases with the existing, naïve approach. Overall, COLA outperforms SS by 46% in ACL, 29% in FW and 49% in IPC; GS by 88% in ACL, 65% in FW, 83% in IPC; Hermes by 45% in ACL, 23% in FW and 50% in IPC.

VI. CONCLUSION

In this paper, we present COLA, a consistency-preserving optimization framework for efficient TCAM updates that ensures forwarding correctness during the entire update process. COLA uses batch updates and applies a new rule selection strategy named MEF to speed up single insertions. Furthermore, COLA uses layered TCAM to avoid forwarding inconsistencies. Asynchronous batch update inserts temporary rules into logical sub-tables of the TCAM during rule accumulation and writes back rules in a specific order. To solve reordering cases, COLA applies a new solution that outperforms existing approaches in both space utilization and efficiency. Our evaluation results show that COLA requires up to 88% less overall movements in TCAM than state-of-the-art TCAM update algorithms when considering reordering cases.

ACKNOWLEDGMENTS

The work was supported by the National Natural Science Foundation of China under Grant No. 61972101. Jin Zhao is the corresponding author.

References

[1] S. Jain, A. Kumar, S. Mandal *et al.*, "B4: Experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM*, Aug. 2013, pp. 3–14.

[2] B. Niven-Jenkins, D. Brungard, M. Betts *et al.*, "Requirements of an mpls transport profile," IETF, RFC 5654, 2009. [Online]. Available: https://tools.ietf.org/html/rfc5654

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks." in *Proc. USENIX NSDI*, Apr. 2010, pp. 89–92.

[4] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. PAM, Springer Lecture Notes in Computer Science*, vol. 8995, 2015, pp. 347–359.

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

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

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

[8] S. Vissicchio and L. Cittadini, "Safe, efficient, and robust SDN updates by combining rule replacements and additions," *IEEE/ACM Trans. Netw*, vol. 25, no. 5, pp. 3102–3115, Feb. 2017.

[9] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proc. ICDCN*, Jan. 2013, pp. 439–444.

[10] K. He, J. Khalid, S. Das *et al.*, "Mazu: Taming latency in software defined networks," Univ. WisconsinMadison, Madison, WI, USA, Rep. TR1806, 2014. [Online]. Available: http://digital.library.wisc.edu/1793/68830

[11] A. Lazaris, D. Tahara, X. Huang *et al.*, "Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization," in *Proc. ACM CoNEXT*, Dec. 2014, pp. 199–212.

[12] P. Pereíni, M. Kuzniar, M. Canini *et al.*, "Espres: transparent sdn update scheduling," in *Proc. ACM HotSDN*, 2014, pp. 73–78.

[13] B. Zhao, J. Zhao, X. Wang *et al.*, "Ruletailor: Optimizing flow table updates in openflow switches with rule transformations," *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 4, pp. 1581–1594, Dec. 2019.

[14] B. Vamanan and T. Vijaykumar, "TreeCAM: decoupling updates and lookups in packet classification," in *Proc. ACM CoNEXT*, Dec. 2011, pp. 27:1–12.

[15] P. He, W. Zhang, H. Guan *et al.*, "Partial order theory for fast TCAM updates," *IEEE/ACM Trans. Netw*, vol. 26, no. 1, pp. 217–230, Feb. 2018.

[16] D. Shah and P. Gupta, "Fast updating algorithms for TCAM," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, Jan./Feb. 2001.

[17] X. Wen, B. Yang, Y. Chen *et al.*, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE ICDCS*, Jun. 2016, pp. 179–188.

[18] K. Qiu, J. Yuan, J. Zhao *et al.*, "FastRule: Efficient flow entry updates for TCAM-based OpenFlow switches," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 484–498, Mar. 2019.

[19] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance SDN switches," in *Proc. ACM CoNEXT*, Dec. 2017, pp. 283–295.

[20] J. M. Lucas and M. S. Saccucci, "Exponentially weighted moving average control schemes: properties and enhancements," *Technometrics*, vol. 32, no. 1, pp. 1–12, 1990.

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