Partial Order Theory for Fast TCAM Updates

Peng He, Wenyuan Zhang, Hongtao Guan, Kavé Salamatian, and Gaogang Xie

***Abstract*— Ternary content addressable memories (TCAMs) are frequently used for fast matching of packets against a given ruleset. While TCAMs can achieve fast matching, they are plagued by high update costs that can make them unusable in a high churn rate environment. We present, in this paper, a systematic and in-depth analysis of the TCAM update problem. We apply partial order theory to derive fundamental constraints on any rule ordering on TCAMs, which ensures correct checking against a given ruleset. This theoretical insight enables us to fully explore the TCAM update algorithms design space, to derive the optimal TCAM update algorithm (though it might not be suitable to be used in practice), and to obtain upper and lower bounds on the performance of practical update algorithms. Having lower bounds, we checked if the smallest update costs are compatible with the churn rate observed in practice, and we observed that this is not always the case. We therefore developed a heuristic based on ruleset splitting, with more than a single TCAM chip, that achieves significant update cost reductions (1.05∼11.3×) compared with state-of-the-art techniques.**

***Index Terms*— Ternary content addressable memory (TCAM) update, software-defined networking, partial order theory.**

I. Introduction

SOFTWARE-DEFINED Networking (SDN) has renewed interest in packet classification algorithms. Generally, packet classification consists of matching a flow of incoming packets against a set of pre-defined rules. The data plane of a general SDN can be abstracted as a general packet classification engine that matches rules defined over different packet headers and acts correspondingly to the matched rules. Designing a flexible and efficient packet classification engine is therefore a key challenge for SDNs and, more generally, for any packet processing system.

There are two types of packet classification approaches: ternary content addressable memory (TCAM) and RAM-based solutions. RAM-based solutions, also known as algorithmic solution, are generally implemented as decision trees stored in RAM memory that do multi-dimensional packet classification. These solutions have to deal with a fundamental trade-off between the size of decision tree data structures and the look-up speed. This trade-off is hard to control for real rulesets, resulting for practical rulesets in very large memory footprints to achieve high-speed lookup [4], [15]. In contrast, TCAMs use a native hardware parallel search that can guarantee low deterministic look-up time and high performance. So, despite being power-hungry, expensive and taking up quite a bit of silicon space, TCAMs are still the *de facto* standard hardware solution used in practice for high-speed packet processing. This motivates largely the study of updates in TCAM-based switches.

There is already a large corpus of the literature dealing with TCAM-based packet classification. Most existing works aim at improving the space utilization [9], [13] or managing the high power consumption [11], [23]. Nonetheless, the major issue plaguing TCAMs and limiting their usage are the high cost of any ruleset updates (inserting or updating rule) in terms of TCAM operations, *i.e.*, inserting a single rule can need up to 1000 TCAM operations [18]. During the update operations, incoming packets can not be processed or matched correctly. The update problem is not a major issue when rulesets, *e.g.*, forwarding tables, are almost static or when packet arrival rates are much smaller than TCAM clocks.

The TCAM update problem is not a new problem and has been studied extensively in the past 15 years [17], [18], [21]. In practice, rulesets are updated by prepending "dropping rules" on the TCAM where the ruleset resides. The dropping rules will override the updated rule until update finishes. However as the update duration might be relatively long, this solution could result in packet drops during the update duration.

Moreover, the churn rates in new scenarios are increasing. Examples of such scenarios are OpenFlow switches where the fast flow cache (TCAM) changes dynamically [5], [6]; vCRIB [14] where rules are migrated between software and hardware switches to reduce the packet processing CPU load; multi-tenancy data centers, where the network configuration has to be frequently updated to adapt to fast varying requirements. All these examples provide scenarios where pausing the forwarding activity for a TCAM update will not be acceptable in real high-speed networks. More generally, full implementation (over the 11 matching fields defined in the standard [12]) of OpenFlow for high-speed switches will need to overcome the TCAM update problem. These scenarios give motivations about reducing further TCAM update delays and cost.

Rather than proposing "*yet another*" new heuristics, this paper provides a systematic and in-depth analysis of the update problem in TCAM used for multi-dimensional packet classifi-

Manuscript received October 25, 2016; revised June 10, 2017, August 11, 2017, and October 14, 2017; accepted November 2, 2017; approved by IEEE/ACM Transactions on Networking Editor A. Bremler-Barr. Date of publication December 13, 2017; date of current version February 14, 2018. This work was supported in part by the National Nature Science Foundation of China under Grant 61502458 and Grant 61725206, and in part by the Natural Science Foundation of Beijing under Grant 4162057. *(Corresponding author: Gaogang Xie.)*

P. He and H. Guan are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: hepeng@ict.ac.cn; guanhongtao@ict.ac.cn).

W. Zhang is with the Department of Computer Science, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: zhangwenyuan@ict.ac.cn).

K. Salamatian is with the University of Savoie, 74016 Annecy-le-Vieux, France (e-mail: kave.salamatian@univ-savoie.fr).

G. Xie is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the Department of Computer Science, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: xie@ict.ac.cn).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors.

Digital Object Identifier 10.1109/TNET.2017.2776565

cation. The contributions of this paper can be summarized as follows:

1) We apply partial order theory to uncover fundamental order constraints (Section II) that any TCAM-updating algorithm based on rule-swapping should comply. These order constraints lead to defining a generic TCAM update process (Section III) that will help in obtaining the lowest cost update algorithm, *i.e.*, the optimal update algorithm. More specifically, we extend a well-known FIB update algorithm (Chain Ancestor Ordering, CAO) [17], which was considered up to now to be optimal, to multi-dimensional packet classification and reveal that there exists a new algorithm that outperforms it. We have implemented and fully evaluated these algorithms both for update cost and processing time (Section IV). The evaluation shows that the optimal algorithm outperform existing ones, both in average ($1.6 \sim 5.7\times$) and worst update cases ($1.4 \sim 2.4\times$). Nonetheless, we show that the processing time for deriving such an algorithm is unacceptable for large rulesets.
2) As a side finding, we reveal a special update case leading to very high update cost (Section III-C), named as the "reordering update" case, which has not been detailed in previous publications. Using the Hasse diagram resulting from partial orders, we provide an upper and lower bound in the normal update case (Section III-D) for practical algorithms. We also derive a method to estimate the cost in the reordering case. This results in a pragmatic estimation of update cost for realistic algorithms.
3) We derive conditions under which the TCAM ordering with lowest average update cost exists and present an $\mathcal{O}(|V|^2)$ algorithm to obtain the order.
4) We observe that the lowest update cost might still be too large for practical scenarios with relative high churn rates. We therefore propose a heuristic for splitting a ruleset into a small number of low update cost sub-rulesets (Section V). Each sub-ruleset will be matched independently with one TCAM chip. In more detail, we leverage frequently observed features in real rulesets to propose a rule splitting heuristic, Black-White Split. Experimental evaluation (Section VI) shows that the proposed Black-White Split can reduce the average update cost by a coefficient 1.05 to 11.3, with consuming up to $10\times$ smaller TCAM space, compared with the state of the art method, TreeCAM [21].

We present a summary of related works in Section VII and finally conclude in Section VIII.

II. TCAM Updating and Partial Orders

A packet classification ruleset R can be defined as an ordered *sequence* of N rules $\{r_1, r_2, r_3, \ldots, r_N\}$. Each rule consists of a set of *ranges* defined over K fields, *i.e.*, it defines a hyper-rectangle in a K-dimension feature space. Two rules overlap if and only if the ranges of these two rules overlap on all K fields. As rules overlap, a given packet might match several rules. In such cases, the rule with the lowest order, eq. of highest priority, should be applied. The order where rules appear in R is called the "*ruleset order*".

TCAM memory can store bits in three states, 0, 1 or *, *i.e.*, the bitmask representing each (sub)rule is stored in a position of the TCAM memory. A TCAM query returns the position of the first position/bit mask that matches a given entry. It is therefore mandatory to store higher-priority rules in the ruleset before lower-priority ones. However, inserting a new rule or updating a rule might change the priorities and entails a change in the storing order of the rules in the TCAM. This fact results in the "TCAM updating problem".

The rules are stored in the TCAM in the *TCAM order*. A TCAM order is *equivalent* to the ruleset order if for any given input packet, the TCAM outputs the same lookup results as a ruleset order. A TCAM-updating algorithm Γ takes as input a rule r to insert and its position in the ruleset order and rearranges the rules on TCAM such that the resulting TCAM order after the update is equivalent to the ruleset order. One trivial TCAM ordering is precisely to follow the ruleset order. In this case, to insert a rule in k^{th} ruleset position, this specific ordering needs $n - k$ TCAM recopy operations. This cost is too large, especially for large rulesets. Fortunately there exist multiple other TCAM orders that are equivalent to the ruleset order and need fewer numbers of rearrangements. We will leverage the ruleset's *semantic* to define some order constraints that, when followed, ensure that a TCAM ordering is compatible with the ruleset's order.

The overlapping relationship among rules and their position in the ruleset order define a partial order $\mathcal{P}_R = \{R, \prec\}$ over the ruleset R. Let's denote the index of a rule r in ruleset order as $P(r) \in \{1, 2, \ldots, N\}$ and its position in the TCAM order as $T(r)$. We will say $r \prec r'$, if two rules r and r' overlap and $P(r) < P(r')$. The partial order $\mathcal{P}_R$ is the transitive closure of the previously defined relation. The partial order is a feature of the ruleset alone. A TCAM order $T(.)$ is compatible with the partial order $\mathcal{P}(R)$ when: $\forall r, r' \in R,\ r \prec r' \Rightarrow T(r) < T(r')$. The theorem below relates the partial order $\mathcal{P}_{\mathcal{R}}$, the semantic of the ruleset with a TCAM order.

Theorem 1: Only TCAM orders that are compatible with the partial order $\mathcal{P}_{\mathcal{R}}$ can implement the *semantic* of the ruleset.

Proof: Assume that a TCAM order $T(.)$ implementing the ruleset semantic violates the partial order, *i.e.*, two rules s and t exists such that $s \prec t$ and $T(t) < T(s)$. The order property $s \prec t$ happens in two cases: s and t are overlapping on all ranges, or there exists a sequence of N intermediate consecutively overlapping rules $\{r^0 = s, r^1, \ldots, r^N = t\}$, *i.e.* r^i and r^{i+1} are overlapping, such that $s = r^0 \prec r^1 \prec \ldots \prec t = r^N$.

The first case leads immediately to a contradiction; if the fields of an incoming packet are in the overlap space of s and t, it will be matched by the TCAM to t, the rule first appearing, while according to the ruleset semantic, it should match s.

The second case implies that in the sequence $\{r^i\}$, there exist at least two consecutive rules $r^i \rightarrow r^{i+1}$, with $r^i \prec r^{i+1}$ such that $T(r^i) > T(r^{i+1})$. This results in the same contradiction as above. ■

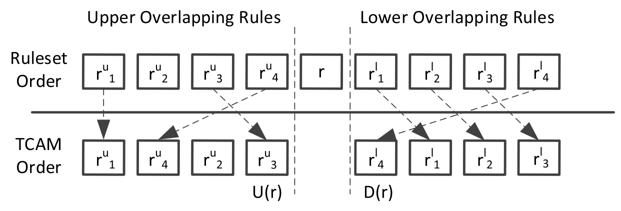


Fig. 1. Upper and lower overlapping rules.

For any rule $r \in R$, the set of rules overlapping with r can be divided into two subsets: upper overlapping rules $R^u(r) = \{r^u_1, r^u_2, \ldots, r^u_n\}$ that are positioned before r in the ruleset order and lower overlapping rules $R^l(r) = \{r^l_1, r^l_2, \ldots, r^l_m\}$ positioned after r in the ruleset order, *i.e.*,

$$r^u_1 \prec r, \ldots, r^u_n \prec r, \quad r \prec r^l_1, \ldots, r \prec r^l_m. \qquad (1)$$

Figure 1 shows the upper and lower overlapping rules in both ruleset and the TCAM order.

Let's assume one wishes to insert a new rule r_{ins} into the ruleset. This will induce a set of new order constraints:

$$\forall r \in R^u(r_{ins}), \quad r \prec r_{ins}, \; \forall r \in R^l(r_{ins}), \; r_{ins} \prec r \qquad (2)$$

that have to be added, along with their *transitive closure*, to the original partial order. We call this new set of order constraints as the augmented partial order. The theorem below gives the conditions to ensure that the TCAM order after inserting a new rule r_{ins} remains compatible with the augmented partial order. Let $U(r)$ be the rule in $R^u(r)$ with the largest TCAM index $U(r) = \max_{r \in R^u(r)} T(r)$ and $D(r)$ be the rule in $R^l(r)$ with the smallest TCAM index in the lower overlapping ruleset, $D(r) = \min_{r \in R^u(r)} T(r)$.

Theorem 2: In order to ensure that the TCAM order after inserting a new rule r_{ins} is compatible with the augmented partial order, it is necessary and sufficient to ensure that $T(U(r_{ins})) < T(r_{ins}) < T(D(r_{ins}))$ without violating the same constraints of other rules. In other words, for any rules r including r_{ins}, we have to ensure $T(U(r)) < T(r) < T(D(r))$ after the insertion of r.

Proof: The necessity is a direct consequence of the definition of the compatibility. The sufficiency proof proceeds in two steps. In the first step, proven in Lemma 1, we have to show that validating the set of order constraints in Eq. 2 is sufficient to ensure that all other transitive constraints are validated. In the second step, we need to ensure that $T(U(r_{ins})) < T(r_{ins}) < T(D(r_{ins}))$ is sufficient to guarantee all order constraints in Eq .2.

Lemma 1: If an inserted rule validates the set of order constraints in Eq. 2, it will comply with all other transitive order constraints.

For any rule r_{ins}, as long as the TCAM order is compatible with $r_{ins} \prec D(r_{ins})$ and $U(r_{ins}) \prec r_{ins}$, the rest of the partial orders shown in Eq 2 are naturally satisfied, as in the TCAM order the other rules in $R^u(r_{ins})$ are already positioned before $U(r_{ins})$ and the other rules in $R^l(r_{ins})$ are positioned after $D(r_{ins})$ (see Figure 1). Therefore, when inserting a new rule r_{ins}, any TCAM-updating algorithm needs to ensure

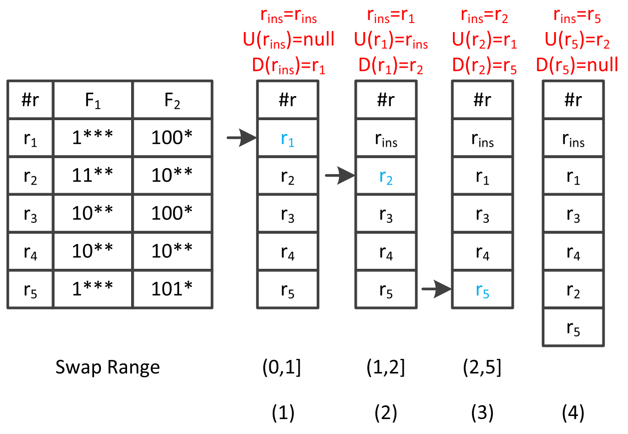

r_ins=r_ins U(r_ins)=null D(r_ins)=r_1
r_ins=r_1 U(r_1)=r_ins D(r_1)=r_2
r_ins=r_2 U(r_2)=r_1 D(r_2)=r_5
r_ins=r_5 U(r_5)=r_2 D(r_5)=null
#r F1 F2
r1 1*** 100*
r2 11** 10**
r3 10** 100*
r4 10** 10**
r5 1*** 101*
Swap Range
(0,1]
(1,2]
(2,5]
(1)
(2)
(3)
(4)

Fig. 2. An example of inserting a rule into the toy ruleset.

$T(U(r_{ins})) < T(r_{ins}) < T(D(r_{ins}))$ without violating the same order constraint of other rules. ■

We will leverage the properties above in the forthcoming to design TCAM update algorithms.

III. Design Space of TCAM Update Algorithms

In this section, we will explore the design space of TCAM update algorithms. We will leverage the partial order constraints described above to develop a generic TCAM updating process and look at different updating heuristics seeking strategies with low update cost and complexity.

A. Generic Update Process

Deleting a rule on a TCAM is trivial, *i.e.*, a rule can be removed by simply disabling the corresponding TCAM entry. However, inserting a new rule is more complex and will need moving rules using several TCAM operations. Updating a rule can be done by deleting it and inserting its updated version. We will therefore focus in the forthcoming on rule insertion.

To insert a rule r_{ins} at the k^{th} position of the ruleset, a TCAM-updating algorithm Γ first scans the ruleset to find $D(r_{ins})$ and $U(r_{ins})$ and chooses a rule in $(T(U(r_{ins})), T(D(r_{ins}))]$ to swap with the new one. In the forthcoming, we will denote the range wherein a TCAM-updating algorithm chooses rules to swap as the *swap range*.[1] If there are empty TCAM slots in this range, we can directly insert r_{ins} into it; if not, we replace a rule r' in the swap range, and we reduce the problem of inserting r_{ins} into inserting the rule r' at a correct position. We continue this process until we find an empty slot in the TCAM or until the repositioned rule r' satisfies $R^l(r') = \emptyset$; in the latter case, this rule can be directly appended into the tail of the TCAM.

Figure 2 shows an example of this generic updating process when we insert r_{ins} at the first position of the ruleset. Let's assume that the rules placed on TCAM are exactly the same as in the ruleset and r_{ins} overlaps with r_1, so its $D(r_{ins}) = r_1$ and its swap range is $(0, 1]$. We need to reposition r_1 after

[1]The swap range is a feature for every rule; it is not only related to rule insertion. It indicates the range of TCAM entry indexes that *every* rule can be positioned at, *e.g.* we can swap existing rules without inserting new ones.

swapping it with r_{ins}. Now $D(r_1) = r_2$, $U(r_1) = r_{ins}$, and the swap range is $(1, 2]$, so we can only swap r_1 with r_2. For r_2, $D(r_2) = r_5$, $U(r_2) = r_1$, and the swap range is $(2, 5]$. We choose r_5 to swap, and as $R^l(r_5) = \emptyset$, we insert r_5 at the tail of the TCAM. We see that the repositioning during an insertion defines an *update chain*, *e.g.*, the update chain in the example is $r_1 \rightarrow r_2 \rightarrow r_5$, and its length is 3, *i.e.*, the insertion needs 3 rule swaps (TCAM operations).

We look here at the situation where there is no headroom free space on the TCAM, so that one has to swap *down* rules, *i.e.*, the last swapped rule can be eventually inserted at the empty slots at the tail of TCAM. However, one can easily change the swap range to $[T(U(r_{ins})), T(D(r_{ins})))$ to enable a swap *up* operation and to extend the description which is therefore generic.

In Fig. 2, we show the rule swaps relative to one update following the order of the rule sequence in the update chain. However, if writing TCAMs following this order, in each step, there will be one rule overridden, resulting in wrong matches if the TCAM is searched at the same time, *i.e.*, during the update, the match engine needs to be paused and incoming packets need to be buffered. However, this can be avoided by pushing the updates in the reverse order. For example, in Fig. 2, we first copy r_5 into a TCAM empty slot, and then copy r_2 to the original location of r_5. Thereafter, we copy r_1 to the position of r_2, and finally write r_{ins} to replace r_1. During the whole process, no rules are overridden, so updates and lookups can be operated simultaneously. In the forthcoming we will only describe the update process using the order shown in Fig. 2.[2]

B. Update Strategies

The generic update process described above does not specify how to choose the rule to swap. An ideal strategy not only needs to have a low number of rule swaps (TCAM operations), it should also ensure that the time/space complexity of choosing a rule to swap is low. According to the swapping range size, we define three categories of update strategies: Γ_{full}, Γ_{bh}, and Γ_{down}. Γ_{full} chooses the rule to swap from the full swap range, and Γ_{bh} chooses the rule to swap from the *bottom half* of the full swap range $[T(r_{ins})+1, T(D(r_{ins}))]$; Γ_{down} always chooses $D(r_{ins})$. We illustrate in Figure 2 the Γ_{down} strategy.

One can use different heuristics to choose the swapped rule. Given a ruleset R and a TCAM order Ω, let's define the update cost $L(r, \Omega)$ of any rule as the length of the update chain starting at r. If we can precompute $L(r, \Omega)$ for each rule r, we can choose, for example, to swap a rule with the rule in its swap range that has the smallest update cost, resulting in a very low update cost strategy.

1) Γ_{full}: Among the three strategies, Γ_{full} has the largest flexibility in choosing swapped rules. However, for this strategy, it is impossible to precompute the update chain for each rule and therefore to predict the *a priori* update cost. This is because in Γ_{full}, swapping a rule might enlarge or narrow

[2]When performing simultaneous reads and writes on the same TCAM entry, an extra blocking delay will arise. However the delay time is in the order of some nanoseconds and negligible.

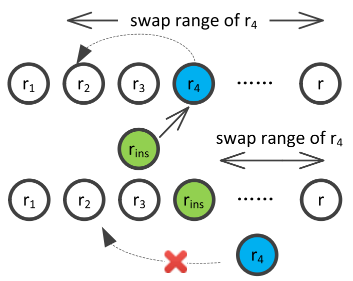


Fig. 3. Suppose that Γ_{full} would choose r_4 to swap with r_{ins} and the update chain of r_4 is $r_4 \rightarrow r_2 \rightarrow \ldots$. Assuming that r_{ins} overlaps with r_4, after swapping with r_4, the swap range of r_4 has been narrowed and r_4 now cannot swap with r_2 because it should be positioned after r_{ins}. In this case, the update chain starting from r_4 changes as rule order changes. Thus, in Γ_{full} the update chains of rules cannot be precomputed.

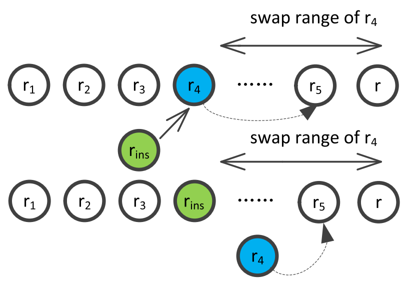


Fig. 4. The swap range of r_4 will not change after swap r_4 with r_{ins}.

the swap range of the next rule to swap. This can eventually change the update chain. We show an example of this in Figure 3.

2) Γ_{bh} and Γ_{down}: In contrast, for Γ_{bh} and Γ_{down}, the swapping rule does not change the swap range of the next rule to swap (see Figure 4). We can therefore compute *a priori* the update cost of each rule. The update cost $L(r, \Omega)$ can be easily calculated through a dynamic programming algorithm. This algorithm begins by assigning an update cost of 1 to the rules with no $D(r)$, as these rules need only one swap, and for any other rule r, the update cost $L(r, \Omega)$ can be calculated recursively using the update cost of rules $r_{s1}, r_{s2}, \ldots$positioned within r's swap range, *i.e.*,

$$L(r, \Omega) = min\{L(r_{s1}, \Omega), L(r_{s2}, \Omega), \ldots\} + 1 \qquad (3)$$

Chain Ancestor Ordering (CAO) [17] is in fact a single dimension-specific variant of Γ_{down} where overlap can only occur over one field. Interestingly, CAO was believed to be an optimal update algorithm [21], while we show here that even for a single dimension, Γ_{bh} can have better update performance than Γ_{down} and thus CAO, because of a larger swap range.

3) Comparing the Three Strategies: It is noteworthy that the swap range in Γ_{full} is containing the swap range of other strategies, and therefore, Γ_{full} might outperform competitors. Nonetheless, we observed that Γ_{full} has unpredictable performance and therefore, it can not be used in practice where deterministic update costs are mandatory. We will therefore focalize on predictable strategies and put aside Γ_{full}.

Both Γ_{bh} and Γ_{down} have predictable update performance and are usable in practice. Because of its larger swap range,

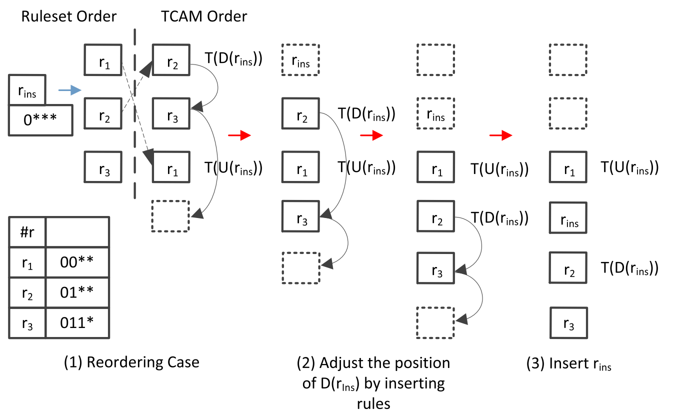


Fig. 5. An example of the reordering update case.

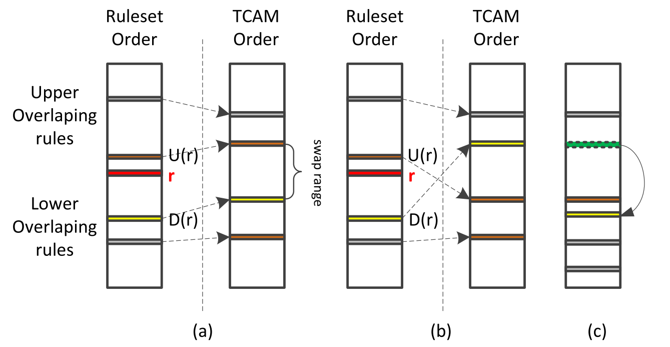


Fig. 6. Two update cases. (a) Normal case. (b) Reordering case. (c) Adjusting $D(r)$.

it is expected that Γ_{bh} would outperform Γ_{down}. However, in Γ_{bh}, the cost of calculating the next position for each rule is quite large. While Γ_{down} only swaps a rule r with its $D(r)$, Γ_{bh} has to choose to swap r with a rule r' in its swapping range with the smallest cost $L(r', \Omega)$. However, as repositioning each rule might change its update cost (because its swap range has changed after reposition), in Γ_{bh}, we have to recalculate the cost $L(r, \Omega)$ of these repositioned rules before the next insertion. Moreover, since, in Γ_{bh}, a rule's cost also determines the update cost of other rules as long as their swap ranges contain this rule, this means we still have to re-check all these rule update costs. This re-check is general and is required for any implementation. In our experiments, we observed that a single insertion using Γ_{bh} usually results in the worst case, *i.e.*, having to recalculate $L(r, \Omega)$ for all rules in the ruleset.

In terms of the worst case complexity, Γ_{bh} has to recalculate for each insertion, the cost $L(r, \Omega)$ for all rules in the ruleset. Calculating the update cost of a single rule needs to find its $D(.)$ in order to determine its swap range. The complexity of this operation is $\mathcal{O}(N)$, and the overall complexity of calculation $L(r, \Omega)$ for all rules in the ruleset is therefore $\mathcal{O}(N^2)$. Nevertheless, Γ_{down} is less complex, as it only finds $D(.)$ for all re-positioned rules during an update process. The cost of Γ_{down} is therefore $\mathcal{O}(N)$. This means that even if Γ_{bh} might achieve the optimal cost among all predictable update strategies, it might be too costly for practical usage, as its processing delay reduces the achievable churn rate. We will evaluate this point in Section IV.

C. The Reordering Update Case

The general update process we described above is based on the assumption that $T(U(r_{ins})) < T(D(r_{ins}))$. However, this assumption might not always be correct. When the current TCAM order is incompatible with the new order constraints resulting from inserting r_{ins}, we will have $T(U(r_{ins})) > T(D(r_{ins}))$, and we need extra operations to make the TCAM order compatible. Figure 5 illustrates such an update case. Let's assume a ruleset $\{r_1, r_2, r_3\}$, where the rule r_1 is not comparable with any other rules. Therefore, on TCAM, r_1 can be positioned anywhere, *e.g.*, the TCAM order $\{r_2, r_3, r_1\}$ shown in Figure 5 is a compatible order. Now, let's assume we insert a rule r_{ins} overlapping with both r_1 and r_2. This results in two new partial orders, $r_1 \prec r_{ins}$ and $r_{ins} \prec r_2$, to be added and the two non-comparable rules (r_1, r_2) becoming comparable, *i.e.*, $r_1 \prec r_2$. However, as $T(r_1) > T(r_2)$, we have to reorder the positions of r_1 and r_2 before inserting r_{ins}. This reordering is very costly, as it entails extra rule repositioning. We call such a situation the *reordering case*, as it requires to reorder the existing TCAM order before inserting the new rule. It is noteworthy that the case described above of reordering has not been fully analyzed in the literature. We will later evaluate the additional cost resulting from it. Figure 6 compares the normal and the reordering case. We see in Figure 6 that the reordering update case happens if $T(U(r_{ins})) > T(D(r_{ins}))$.

A key observation is that the new order constraints between existing rules are all generated through the "link" of r_{ins}. So, as long as we ensure that r_{ins} sits in front of all the lower overlapping rules and after all the upper overlapping rules, *i.e.*, $T(U(r_{ins})) < T(r_{ins}) < T(D(r_{ins}))$, all partial orders constraints are satisfied. This means that the order constraints introduced in Section II remain valid and that the generic update is still applicable, even in this special update case.

Therefore, the solution to the reordering issue is to shift down $D(r_{ins})$ in such a way that $T(U(r_{ins})) < T(D(r_{ins}))$. Shifting down $D(r_{ins})$ is implemented by applying the Γ_{down} strategy to insert r_{ins} at position $T(D(r_{ins}))$. If after this insertion, $T(U(r_{ins})) < T(D(r_{ins}))$, we are set. If not, we have to erase the position where we inserted r_{ins} and reinsert r_{ins} using Γ_{down} at the new position $T(D(r_{ins}))$, as with each insertion, $D(r_{ins})$ changes as the TCAM rule order has changed. We continue to do this until the position of r_{ins} in TCAM satisfies $T(U(r_{ins})) < T(r_{ins})$. Since, for each insertion, $D(r_{ins})$ has the smallest TCAM index rule, the procedure above ensures that $\forall r \in R^l(r_{ins}), T(U(r_{ins})) < T(r_{ins}) < T(r)$, resulting in the obtained TCAM order becoming compatible with the augmented partial order after adding r_{ins}. An example of this process is shown in Figure 5. To note, after each rule shift-down, we have to erase r_{ins}, since it is only used as an auxiliary.

It is noteworthy that both the reordering case and rule deletion, generate empty TCAM entries. These empty entries can be reused for inserting new rules: if an empty TCAM entry exists in the swap range of an inserting rule, we can simply put the inserted rule into this position and terminates the update process.

Algorithm 1 ShiftRuleDown(r_{ins}, R)

```
$r \leftarrow r_{ins}$
$r^D \leftarrow$ FindDRule($r$, $R$)
**while** $r^D \neq$ nil **do**
    WriteRule($r$, $T(r^D)$)
    let $r \leftarrow r^D$
    $r^D \leftarrow$ FindDRule($r$, $R$)
**end while**
AppendRuleInAnEmptySlotAtTail($r$)
```

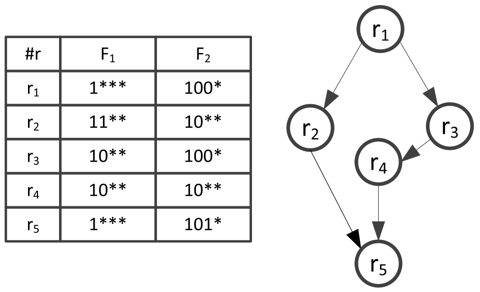
Algorithm 2 InsertRuleBH(r_{ins}, R)

```
let $r \leftarrow r_{ins}$
let $L \leftarrow []$
$r^U \leftarrow$ FindURules($r$, $R$)
$r^D \leftarrow$ FindDRules($r$, $R$)
**while** $T(r^U) > T(r^D)$ **do**
    ShiftRuleDown($r$, $R$)
    Del the inserted $r$
    $r^U \leftarrow$ FindURules($r$, $R$)
    $r^D \leftarrow$ FindDRules($r$, $R$)
**end while**
push $r_{ins}$, $L$
Calculate weight table $L(r, \Omega)$
Find smallest weight rule $r_{swap}$ in $(T(r^U), T(r^D)]$;
WriteRule($r$, $T(r_{swap})$)
push $r_{swap}$, $L$
let $r \leftarrow r_{swap}$
$r^D \leftarrow$ FindDRules($r$, $R$)
**while** $r^D \neq$ null **do**
    Find smallest weight rule $r_{swap}$ in $[T(r) + 1, T(r^D))]$
    WriteRule($r$, $T(r_{swap})$)
    push $r_{swap}$, $L$
    let $r \leftarrow r_{swap}$
    $r^D \leftarrow$ FindDRules($r$, $R$)
**end while**
AppendRuleInAnEmptySlotAtTail($r$)
RecalCost($L$, $R$)
```

D. Putting It All Together

Now, we summarize the complete Γ_{bh} algorithm in Algorithm 2. The function `ShiftRuleDown` adjusts the positions of $U(r_{ins})$ and $D(r_{ins})$ before r_{ins} is inserted; the functions `FindURule` and `FindDRules`, respectively, find for a given rule r the corresponding $U(r)$ and $D(r)$ according to the priority $P(r)$; the function `T` returns the TCAM slot index for a given rule.

In Algorithm 2, we track all the repositioned rules using a list L. The update cost of these rules is recalculated in `RecalCost` in reverse order. Also, we scan each rule in R in reverse order to check whether its swap range contains any rule in L, and if so, we also update its update cost and add it into L. For simplicity, we omit the details of `RecalCost` and the process of searching empty slots in the swap range that will terminate earlier the update process.



#r	F1	F2
r1	1***	100*
r2	11**	10**
r3	10**	100*
r4	10**	10**
r5	1***	101*

Fig. 7. The Hasse diagram of the partial orders.

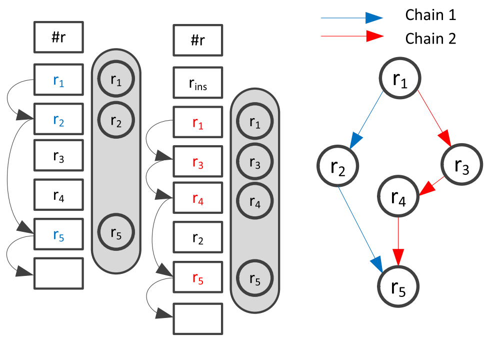


Fig. 8. Rule shifting will follow different chains.

E. Hasse Diagram and Γ_{down}

Partial orders can be represented by different types of graphs or diagrams. In particular, Hasse diagrams are frequently used, as they provide minimal representation of a partial order. Figure 7 illustrates an example of the Hasse diagram of the toy ruleset shown in Figure 2. Each node of the Hasse diagram corresponds to a rule, and there is a directed edge connecting two nodes $r, r' \in R$ if and only if $r \prec r'$ and there is no r'' such that $r \prec r'' \prec r'$. As can be seen, we are using a variation of Hasse diagrams that use directed links, following $\prec$ relations, whereas classical diagrams do not use directed links and represent the relation direction by the higher or lower vertical position of nodes.

An important feature of a Hasse diagram is that no node's children are comparable. We can use this feature for TCAM ordering. In Figure 7, as the children of r_1, r_2, and r_3 are not comparable, either of them can be positioned on TCAM as the first overlapping rule after r_1. Recall that Γ_{down} swaps $D(r)$, *i.e.*, the first overlapping rule, with r. Therefore, the candidates to swap with (to become $D(r)$) for all compatible TCAM orderings are the children of r in the Hasse diagram. Therefore, a path in the Hasse diagram starting from a node and terminating at a node without egress is a possible update chain for an insertion. This path length represents the number of swaps (TCAM operations) in this insertion process, *i.e.*, the cost of this insertion.

There are several possible paths beginning at any node, and depending on the TCAM order ($T(r_2) < T(r_3)$ or $T(r_3) < T(r_2)$), one of these alternatives will be the update chain. We illustrate this in Figure 8 with two alternative paths in blue and red. The specific TCAM order is therefore impacting the complexity of the update operation. As updates might change the TCAM order, it can change the Γ_{down} cost. For example, in Figure 7, adding r_{ins} changes the TCAM order, and this shifts the update path of r_1 from the blue path into the red one. In the forthcoming, we provide an analysis of update cost dynamics through a refined analysis of the update cost of Γ_{down}.

F. Average, Weighted, and Max-Min Update Cost

We defined before $L(r_k, \Omega)$ as the length of an update chain beginning at r_k when the TCAM order is Ω. We can derive the average update cost of Γ_{down} algorithm as:

$$UC(\Omega, \mathbf{w}) = \Sigma_{k=1}^{N} w_k L(r_k, \Omega) \tag{4}$$

The average update cost depends on the probability w_k that for a newly inserted rule r, $D(r) = r_k$, *i.e.* the update chain begins at r_k, and the update cost is $L(r_k, \Omega)$. As large range rules are more likely to overlap with a newly inserted rule r, they are more likely to become the $D(r)$, *i.e.*, w_k is larger for large range rules than for small range ones. However, without *a priori* knowledge of the inserted rules, one can not derive the probabilities w_k and compute precisely the average update cost.

In order to obtain an indicative average cost, and in the absence of a given distribution of incoming updates, we will assume that the incoming update stream results in a uniform choice of update chain, *e.g.*, a new inserted rule has equal probability of overlapping with any rule in the ruleset; we will define the *average uniform update cost* UC of a ruleset with a TCAM order Ω as:

$$UC(\Omega) = \frac{\Sigma_{k=1}^{N} L(r_k, \Omega)}{N} \tag{5}$$

For each rule r_k, we define the cost of the longest update chain beginning at r_k as $L_{max}(r_k)$ and the cost of the shortest update chain as $L_{min}(r_k)$. We define the maximal cost TCAM order $\Omega_{\max}$ as the order (if it exists) where all nodes will follow their maximal update chain. The average cost of this maximal cost order, named the maximal update cost R, is denoted as $UC(\Omega_{\max}) = UC_{\max}(R)$. Similarly, the minimal cost order $\Omega_{\min}$ is the order (if it exists) where all nodes are following their minimal update chain. The minimal update cost of the ruleset R is denoted as $UC(\Omega_{\min}) = UC_{\min}(R)$. By definition, for all TCAM orders Ω, $UC_{\min}(R) \leq UC(\Omega) \leq UC_{\max}(R)$.

The minimal and maximal update cost of a given rule r_k under the Γ_{down} algorithm can be calculated by a dynamic programming algorithm. We begin the dynamic programming from nodes without out-going edges. For these nodes, we assign $L_{\min} = L_{\max} = 1$. Thereafter, for any node r_k, we can calculate the max-min update cost of this node using the maximal and minimal update cost of its children r_{c1}, …, r_{cn}:

$$L_{\max}(r_k) = \max\{L_{\max}(r_{c1}), \ldots, L_{\max}(r_{cn})\} + 1 \tag{6}$$

$$L_{\min}(r_k) = \min\{L_{\min}(r_{c1}), \ldots, L_{\min}(r_{cn})\} + 1 \tag{7}$$

In order to obtain the shortest/maximal path for one node, we have to follow all its edges and check all its children.

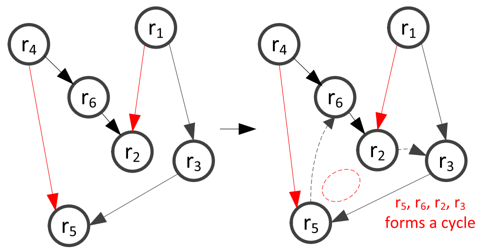


Fig. 9. Example of scenario where the augmented Hasse diagram has a loop and the minimal update cost of r_1 and r_4 can not be achieved simultaneously.

Therefore the overall complexity of the algorithm is $\mathcal{O}(|V| + |E|)$, where $|V| = N$ is the number of the rules and $|E|$ the number of edges of the Hasse diagram.

G. Achievability of Minimal Update Cost

Knowing if a minimal or maximal update cost is achievable for the Γ_{down} algorithm, *i.e.*, if there exists an order that will results in all nodes achieving simultaneously their minimal or maximal cost, is indeed an important question. One way to enforce that all rules in a ruleset follow the minimal update path consists of adding additional order constraints forbidding to follow non-minimal update paths. Let $A(r_k) = \{r_{k1}, \ldots, r_{kn}\}$ be the set of child nodes of a node r_k and $B(r_k) \subset A(r_k)$ be the subset of children leading to a minimal update path with length $L_{\min}(r_k)$. To ensure that only the minimal update path can be followed, we add for each rule in $r \in A(r_k) - B(r_k)$ a spurious order constraint $r' \prec r$, where $r' \in B(r_k)$. We will call the new Hasse diagram with additional order constraints the “augmented Hasse diagram” and the new partial order the “augmented partial order”.

The additional partial order constraints in the augmented partial order do not change the semantic of the ruleset. They just make it stricter. In other terms, if a TCAM order is compatible with the augmented partial order, it will also be compatible with the initial order (as validating the larger set of constraints entails validating the smaller subset). However, as the additional partial orders constraints are artificial, there is no guarantee that there exists a TCAM order that is compatible with the augmented partial order. In particular, if adding the new partial order constraints results in a loop $r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow r_l \rightarrow r_1$ in the augmented Hasse diagram, this entails $T(r_1) < T(r_2) < \ldots < T(r_l) < T(r_1)$, which is impossible. In this case, the TCAM order that achieves minimal cost does not exist.

Figure 9 illustrates an example where the following rule of r_1 should be r_2 to achieve its minimal update cost. Similarly, r_4 should have r_6 as its following rule. Now, if we have to add two new partial orders $r_5 \prec r_6$ and $r_2 \prec r_3$, this will results in a loop $r_6 \rightarrow r_5 \rightarrow r_2 \rightarrow r_3$, *i.e.,* no orders can be compatible with such a graph. If the augmented Hasse diagram has no loop, the TCAM ordering achieving the minimal update cost exists and can be derived by a **topological sort** of the augmented Hasse diagram.

Theorem 3: Checking if a minimal update cost order for the Γ_{down} algorithm exists and obtaining it, is possible by an algorithm of complexity $\mathcal{O}(|V|^2)$, where the $|V|$ is the number of rules in the ruleset.

Proof: Following the above description, obtaining such a TCAM ordering entails three steps: One should obtain for each rule the shortest update path on the Hasse diagram; then edges are added to generated the augmented Hasse diagram; finally a topological sort is implemented on the augmented Hasse diagram. The topological sort detects if there is a cycle in a the graph and if so the minimal update cost cannot be achieved.

The complexity of the first step is $\mathcal{O}(|V| + |E|)$ [2]. Assuming the augmented Hasse diagram contains $E' > E$ edges, the complexity of topological sort is $\mathcal{O}(|V| + |E'|)$. As the largest Hasse diagram with $|V|$ nodes has not more than $|V|(|V| - 1)/2$ edges, the overall complexity of finding a minimal cost TCAM order is at most $\mathcal{O}(|V|^2)$. ■

It is noteworthy that the optimal TCAM order derived above is not stable, *i.e.*, additional insertions result in rule shifts that makes the order not optimal anymore. This means that one can use an optimal order when a new ruleset is initially inserted into a TCAM but it will not remain optimal. Another noteworthy comment is that the presented analysis does not depend directly on the number of dimensions in the rules space. It only depends on the rule overlaps in the ruleset.

H. Reordering Case Update Cost

The description above ignores the update cost for a reordering case. While this update case rarely happens in practice, it is still mandatory to study its cost, as it impact considerably the update cost once it happens. As explained, the update cost for a reordering case consists of two components: the cost of reordering the current rule positions in order to make the TCAM order compatible with the ruleset order and the cost of inserting the new one. The cost of reordering the current TCAM order is hard to predict in advance, since it depends on the particular TCAM order and on the inserted rule. Nonetheless, we can derive some lower bound for the worse case reordering cost.

For this purpose, we search for each rule $r \in R$, the incomparable rule $I_c(r)$ with the largest TCAM index. The reordering update cost is exactly equal to the cost of repositioning r to the TCAM position $T(I_c(r))$. This cost can be obtained by simulating the update process described in III-C and counting the number of repositioned rules. However, this is very time consuming, especially for a large ruleset as the whole process needs to be applied for each $r \in R$.

We can use a heuristic to derive a lower bound on the cost. Since all rules in the range $(T(r), T(I_c(r))]$ that are comparable with r definitely have to be repositioned, a simple heuristic consists of finding these rules and observing that the final update cost will be larger than the number of these rules, as some rules positioned after $I_c(r)$ have to be repositioned. This leads to a lower bound on the cost of the worst case reordering: the number of rules in the range $(T(r), T(I_c(r))]$. This number can be easily derived from the Hasse graph by finding that, among rules comparable with r, *i.e.*, the rules that can be reached from r, the number of rules that are in the range $(T(r), T(I_c(r))]$. This lower bound is very valuable, as it enables the developer to have ideas about the lowest cost of updates and decide if it is compatible with the targeted churn rate or not.

TABLE I
UPDATE COSTS OF Γ_{down} AND Γ_{bh}

Type	Size	#Rules	Γ_{down}				Γ_{bh}		Reordering Case LB
			UC_{min}	UC_{max}	$UC(\Omega_0)$	Worst Case	$UC(\Omega_0)$	Worst Case	
ACL	1K	953	5.45	5.45	5.45	10	2.13	7	12
		955	15.40	32.2	29	54	9.4	31	466
		989	3.71	5.96	4.7	13	1.97	7	124
	10K	9792	7.02	10.3	8.63	20	2.66	14	88
		9449	13.69	55.66	47.91	109	13.6	53	3605
FW	1K	815	6.17	12.83	8.84	16	2.9	7	485
		964	8.86	17.95	17.95	29	11.16	21	305
		838	12.47	25.75	19.01	29	8.64	17	1827
	10K	9001	12.90	40.93	22.82	35	8.19	16	3495
		8909	23.68	65.65	44.95	68	14.11	32	10684
IPC	1K	963	13.64	15.82	5.56	27	2.53	11	200
		644	5.01	7.77	7.6	13	2.32	5	129
	10K	9515	13.79	53.93	39.6	79	6.87	43	2603
		10000	2.87	4.36	4.36	11	2.21	5	4971

IV. VALIDATION

We implemented Γ_{bh}, Γ_{down} update strategies and the update cost analysis in C++. Similar to [18] and [21], we use ClassBench [20] to generate synthetic rulesets. We have evaluated all available types of rulesets, including Access Control List (ACL), Firewall (FW), and IP Chain (IPC). Our rulesets contain 1K, 10K, and 20K rules. As TCAM needs prefix rules, we first transform rules into prefix ranges, eventually splitting a single non-prefix rule into several prefix rules, which we call *micro-rules*. This results in an inflation of the number of entries needed on the TCAM that is not caused by our proposed schemes, but is unavoidable. We observed in our evaluation that 20K rules were inflated to up to 68K micro-rules.

We have described in Sec. III-A the update methods without pausing the match engine. However, during the updating, packets are still matched with the old ruleset. It is therefore important to reduce the update time. Rewriting a single entry needs one TCAM cycle, *i.e.*, 2.5 nsec for a 400 MHz TCAM and the update duration is proportional to the number of needed rewrites. Therefore, the first performance criterion is the number of rules swaps. The second criterion is the TCAM space overhead, *i.e.*, the additional space needed by the TCAM update algorithm in excess of the space used to store the micro-rules. A third parameter is the time needed to execute the update algorithm that decides the position where to insert the new rule. This last metric controls the achievable churn rate, *i.e.*, the number of updates that can be managed per second.

A. Update Costs of Γ_{down} and Γ_{bh}

We first derive the update cost of both Γ_{down} and Γ_{bh} for 14 rulesets with 1K to 10K rules. We used as a reference Ω_0, *i.e.*, the TCAM ordering where rules are ordered following the ruleset order. We show in Table I the update cost when inserting a *micro-rule*.

TABLE II
SIMULATED UPDATE COSTS OF Γ_{down}

Type	Size	#Insert Rules	Meas. Max. Update Cost	Calc. Max Update Cost	Meas. Avg. Update Cost	*UC*	#Empty Entries	Empty Entries Overhead
ACL	1K	632	9	9	3.32	3.40	1	0.08%
	10K	6621	42	16	7.15	7.09	9	0.07%
	20K	13798	54	16	5.30	5.23	18	0.07%
FW	1K	1355	95	15	4.42	6.25	23	0.85%
	10K	15991	676	30	16.85	13.51	901	2.82%
	20K	34275	1801	34	15.73	16.95	2027	2.96%
IPC	1K	653	56	24	7.26	9.23	4	0.31%
	10K	6299	856	70	15.76	31.49	167	1.33%
	20K	13635	1557	69	16.85	28.14	235	0.86%

The real average performance observed over a ruleset will ultimately depend on the particular sequence of given updates. Nevertheless, if no reordering is needed, the performance will be in between $UC_{\min}$ and $UC_{\max}$. We see in Table I that 1K rulesets have different update costs, but all have $UC_{\min}$ and $UC_{\max}$ relatively small and close to each other, while 10K rulesets, with more complex rules overlapping relationships, exhibit a larger number of needed updates and a much larger spread between $UC_{\min}$ and $UC_{\max}$. We see in Table I that Γ_{bh} significantly outperforms Γ_{down} both in average ($1.6 \sim 5.7\times$) and worst update cost ($1.4 \sim 2.4\times$). We will evaluate later in sec. IV-A1 the average processing time to decide where to insert a rule using Γ_{down} and Γ_{bh}. Table I also shows that the reordering case cost could be high, even for 1K rulesets. We see that the lower bound on reordering cost could be on the order of several hundreds of rule swaps.

1) Empirical Cost Evaluation: The values presented in Table I were computed using analytic models developed in the paper. In particular, re-ordering costs were only lower bounded. We now present here an empirical evaluation of the update cost, including reordering cost, by measuring the update performance over a synthetic update stream. As we already know from Table I the worst update cost without reordering, this measurement enables to evaluate the effect of reordering on the empirical cost.

We generated the update stream by extracting micro-rules with an even index in the ruleset index, deleting them from the TCAM and reinserting them. We show the results in Table II. Following the generation process for updates, we can infer the number of occupied TCAM entries to be two times the number of inserted rules given in column titled `#Insert Rules`. We show in the column titled `Meas. Max Update Cost` the empirical maximal update cost measured, accounting for re-ordering cases, and in `Calc. Max Update Cost` the theoretical derived maximal update cost. The column titled `Meas. Avg. Update Cost` shows the measured average update cost, while the `UC` column contains the theoretical average update cost.

As can be seen from the difference from Columns 4 and 5, the reordering case can strongly impact the maximal number of updates, *e.g.*, for a 20K FW rules, the cost increases sharply from an expected 34 operations to 1801! We also observe that the measured average update cost is frequently less than the theoretical UC. This is caused by the fact that UC is derived using a uniform distribution of updates, while the measured one is resulting from the particular update stream we used. To put it into perspective, the obtained worst case update performance, 1801 updates, results in a 4.5-μsec update delay that is equivalent to the arrival of 351 packets of 64 bytes on a 40 Gbps link. This update delay is becoming too large for operational deployment. Another noteworthy observation is in the last column of Table II, where we show the number of empty rules in the TCAM that results from the re-ordering cases. One can see that the proportion of empty rules is quite small for all cases, *e.g.*, the largest overhead is only 2.9% (see FW_20K data). It is noteworthy that this space is not really wasted, as the lazy scan approach described in Section III-C can reuse it for future updates.

We finally show in Table III the average processing time to decide the position where to insert a micro-rule for Γ_{down}. We use for this purpose a commodity 2.13 GHz x86 server with 16 GBytes of RAM, and we used a single core. It is noteworthy that during the time we decide where to insert a micro-rule, we do not need to block the data plane, as this can be done before beginning the updates. Therefore, the main impact of this delay is on churn rate. As can be seen, in most cases, the average processing time is less than 1 msec. The only case that Γ_{down} goes beyond 1 msec is the FW 20K ruleset. A more refined look using a profiler shows that a large part of this time is consumed by the linear search to find overlapping rules. This delay can be reduced by using a more efficient data structure, such as tries or decision trees for searching overlapping rules. Nonetheless, this update time is enough for a churn rate of more than 400 updates per second.

We already said that because of the complexity of $\mathcal{O}(N^2)$, Γ_{bh} is too time-costly to be used in practice. We have measured the update time per rule using Γ_{bh} to be up to 20 msec for 1K rules. It becomes prohibitively large for larger rulesets. This means that using Γ_{bh} for a 1K ruleset is still compatible for churn rates up to 50 updates per second. For a larger churn rate, Γ_{down} should be considered.

B. NetFlow Measurement on a Real Trace

Unfortunately, there is no available realistic ruleset for OpenFlow, and, as other works, we have resorted in this paper to synthetic rulesets. However, in order to show that the results are still valid on a real ruleset, we used a specific scenario, assuming that one wants to measure in real time flows going from a BGP level IP mask to another BGP level IP mask. We can do this by inserting into the TCAM any new BGP level flow and matching an incoming packet to it. Using this approach on a trace containing 1 million packets coming from a 10 Gbps backbone link, we generated 30K rules with more specific IP masks having higher priority than less specific ones. Thereafter, we look at the update performance over this ruleset. During the insertion of these 30K rules, we measured an average number of rule movements of 1.60 and a maximal movement cost of 40. It is noteworthy that 40 movements need 100 nsec of update time in the dataplane. Since over a 10 Gbps there is an average of one packet arriving each 68 nsec, all TCAM updates can be completed in 2 packets arrival time.

C. Discussion

The update cost analysis presented above provides indications about the expected minimal and maximal update costs, as well as, the lower bound of reordering costs that an update algorithm will incur. We can see that for some rulesets, the update performance and, in particular, the cost incurred by the reordering case (described in Section III-C) are still too large for TCAM use in high-performance and high churn rate environments. We have therefore to find solutions to get around this. One solution consists of duplicating the TCAM chips on board and having therefore one TCAM chip running when the second one is being updated. While this can solve the issue of overflow and packet losses incurred by update waiting time, it will not solve the issue with churn rate. Moreover, TCAM chips are costly and power-hungry. Duplicating them just to manage the update issues might be overkill. We then still need to find a way to reduce the update cost further and to avoid reordering cases. As explained above, if Γ_{bh} can not achieve a low enough update cost, no other deterministic algorithm will be able to achieve it, and we have therefore to look for other approaches. We will describe in the next section an approach that can achieve this by splitting the ruleset.

V. The Black-White Split Algorithm

In this section, we describe how to split the ruleset into sub-rulesets, each with lower update costs. This will enable us to overcome the update cost performance bottleneck of the unique ruleset approach. The idea consists of deploying each sub-ruleset on independent TCAMs that are searched in parallel for each incoming packet. The final result is set as the highest priority rule among all returned matches. Nevertheless, we still need to decide where to insert newly incoming rules. The idea of rule-splitting has already been proposed in the literature, *e.g.*, TreeCAM [21] used decision trees to split rules into small blocks of overlapping rules and bounded the update overhead.

Although spliting a ruleset into sub-rulesets can reduce the update cost, it has side effects. A packet processing board might not have enough space for accomodating mutiple TCAMs storing sub-ruleset. However, we can use modern TCAMs, that provide multiple TCAM banks to store each sub-ruleset, and multiple query ports to launch several queries in parallel. However using multiple sub-rulesets will slow down the TCAM's overall query throughput. For instance, a 400 Mhz TCAM chip with 4 query ports can achieve 4×400 MHz = 1.6G lookup/sec. However, since each port is used to search on different sub-rulesets, the overall lookup speed is reduced, *e.g.*, with 4 sub-rulesets, the throughput is reduced to 400M lookup/sec. Having more than 4 rulesets will even reduce further the dataplane throughput. It is therefore mandatory to reduce the number of sub-rulesets while maintaining small update cost.

#r	F_1	F_2
r_1	1***	100*
r_2	11**	10**
r_3	10**	100*
r_4	10**	10**
r_5	1***	101*

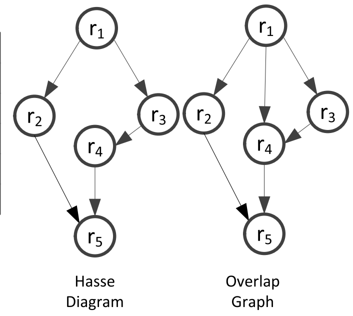


Fig. 10. Overlap Graph and Hasse diagram: Compared to the Hasse diagram, the overlap graph contains an extra edge $r_1 \rightarrow r_4$. This edge is removed in the Hasse diagram, as it can be induced by two edges $r_1 \rightarrow r_3$ and $r_3 \rightarrow r_4$.

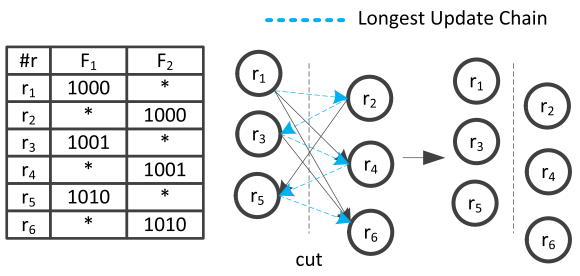
In the forthcoming, we will describe an efficient rule-splitting approach that achieves both high lookup performance and low update cost.

A. Rule Features and the Corresponding Overlap Graph

We will use a variant of the Hasse diagram introduced earlier, the *overlaph graph*. In the Hasse diagram, edges represent only direct overlapping relationships that can not be deduced through transitive completion. An overlap graph is a graph of the transitive closure of the overlap relation, *i.e.*, it contains all *overlapping relationships* between rules, and there is an edge connecting two rules $r, r' \in R$ if and only if r overlaps with r' and $P(r) < P(r')$. We illustrate the overlap graph in Figure 10 and compare it with the Hasse diagram.

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. Therefore, an intuitive approach to reduce update costs is to split the rules into subsets such that the number of order constraints, *i.e.*, edges, between them are as large as possible. Cutting these edges minimizes the number of constraints between the rules in each subset. This is equivalent to solving the *MAX-CUT Problem* for the overlap graph. This problem is known to be NP-hard for general graphs. We present here a heuristic split algorithm that leverages the ruleset properties and features described in [4] and [21].

Commonly, empirical rulesets contains rules with a wildcard $*$ on all k dimensions beside a single dimension, where the selected range is more precisely defined, *e.g.*, $r = \{100*, *, *, *\}$. These rules are generally not overlapping with other rules having specific (non-wildcard) ranges on the same dimension, *e.g.*, $r' = \{110*, *, *, *\}$ does not overlap with r. Nonetheless, such rules overlap with all rules having specific ranges on other dimensions, *e.g.*, $r'' = \{*, 1000, *, *\}$ overlaps with both r and r'. We will call such rules *orthogonal rules*. Let's assume that all orthogonal rules with a specific range on the same dimension are put together in a set S_i. No rule in such a set is overlapping another rule in the same set, but all rules in any set are overlapping with all rules in other sets. We will call the resulting overlap graph a "*k-partite overlap graph*", with each part being an independent set of orthogonal rules with specific ranges on a given dimension. We show in Figure 11



#r	F_1	F_2
r_1	1000	*
r_2	*	1000
r_3	1001	*
r_4	*	1001
r_5	1010	*
r_6	*	1010

Fig. 11. Orthogonal structure rules and the overlap graph resulting from these rules.

a ruleset containing such 2-dimensional orthogonal structure as well as the resulting bipartite overlap graph.

Because of the k-partite structure, inserting a rule in the l^{th} position using Γ_{down} into a set of N orthogonal rules may need up to l rule swaps, resulting in an average update cost of $\frac{N-1}{2}$. However, if we cut the k-partite overlap graph into k independent sets, as shown in Figure 11, all partial order constraints are cut, the overlap graph becomes disconnected, and the average update cost of each sub-ruleset reduces to 1.

The second type of rule leading to high update costs is the *large rules*, *i.e.*, rules having large ranges on all dimensions. These rules overlap with each other. A ruleset containing only N large rules will have a particular overlap graph, where a rule positioned at the k^{th} position in the ruleset has $k-1$ incoming connections from rules positioned before it and $N-k$ outgoing connections to rules positioned after it. We will call such a graph a "*complete overlap graph*". Using Γ_{down} to insert a new rule at the k^{th} position will therefore need to shift down k rules and do replacements. This results in an average update cost $\frac{N-1}{2}$. A good splitting algorithm will cut the large rule overlap graph above described into 2 sub-graphs with nearly the same number of nodes and with similar properties, *i.e.* each rule is connected to the rules following it. The average update cost is therefore reduced to $\frac{N-1}{4}$ for each subgraph.

In addition to the two types of rules above, *small rules* which have small range on all dimensions is a third type observed empirically in ruleset. These rules become sparse isolated nodes in the overlap graph that are only connected to orthogonal and large rules.

As real rulesets are mixtures of the types of rules described above, the overlap graph will therefore contain a mix of quasi-multi-partite, quasi-complete, and isolated graph components and we need a splitting algorithm able to correctly split the resulting structures. We propose the Black-White Splitting algorithm to split the overlap graph into 2 sub-graphs.

B. Black-White Split

The main idea of the splitting algorithm is to color nodes differently, and to split following the colors the graph into 2 sub-graphs . For this purpose, we first color all parentless nodes with a black color. We search thereafter for the first un-colored node in the ruleset order, and we look at its ingress neighborhood. We color the node in a color opposite to the majority of its parents, *i.e.*, if the number of white-colored parents of rule r is $W(r)$ and the number of its black-colored parents is $B(r)$, the node is colored white if $B(r) < W(r)$. We continue this until all nodes are colored. We illustrate the Black-White Split in Figure 12.

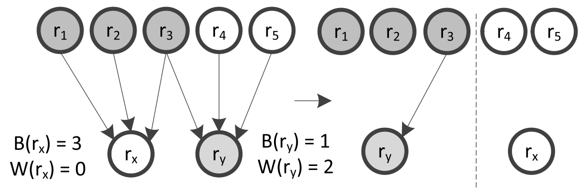


Fig. 12. Black-White split.

The Black-White Split has the properties described in the following theorems.

Theorem 4: Given an overlap graph $G = (V, E)$, one execution of the BW-split algorithm eliminates at least $|E|/2$ edges during the splitting into two sub-rulesets.

Proof: There are $B(r) + W(r)$ edges pointing to any node r. By splitting the rules by color, we eliminate $max(B(r), W(r))$ edges connecting r to its parents with different color. Since $max(B(r), W(r)) \geq \frac{B(r)+W(r)}{2}$, Black-White Split can eliminate at least half of the ingress edges of each node and therefore of the whole overlap graph. ■

Theorem 5: Running the BW-split algorithm $k-1$ times over a k-partite overlap graph $G = (V, E)$, resulting k-dimensions orthogonal rules, will split G into k independent sets.

Proof: See Appendix in online supplementary material. ■

Theorem 6: A single execution of the BW-split algorithm on a complete overlap graph $G = (V, E)$, will produce two sub-graph with $|V|/2$ nodes.

Proof: See Appendix. ■

The two theorems above show that Black-White Split works well on the specific subgraph that we expect to see in real rulesets. However, real rulesets are mixing these different structures, and one can expect that on a realistic overlap graph, our algorithm will split along the direction suggested by the theorems. In practice, we might need to do several iterations of Black-White Split to reach to sub-graphs with a low enough update cost. We have observed empirically that for most rulesets, splitting into 3 sub-rulesets is enough to ensure a small update cost.

VI. Validation of Black-White Split

In order to compare the performance of the proposed ruleset split method, we implemented one of the state-of-art splitting algorithm, TreeCAM [21]. In [21], the authors have compared TreeCAM with a large number of algorithms in the literature and shown that it works better. We therefore only compare our approach with TreeCAM. TreeCAM groups rules according to large/small ranges and provides bounded update cost using decision trees. We only considered in our evaluations of TreeCAM the *No Re-balancing* and *Local Re-balancing* case, as the *Global Re-balancing* case would result in a much higher update cost. This means that our comparison with TreeCAM is conservative and that the performance of TreeCAM can be worse than what we are showing in our evaluation.

TABLE III

AVERAGE PROCESSING TIME OF Γ_{down}

	ACL(μsec)	**FW**(μsec)	**IPC**(μsec)
1K	10	30	17
10K	145	900	430
20K	344	2300	981

TABLE IV

UPDATE COSTS OF SPLIT RULESETS

Type	**Size**	**#Rules**	**UC(Ω_0)**				**Reordering Case**			**Reduction**	
			R1	R2	R3	Worst Case	R1	R2	R3	UC(Ω0)	RC
ACL	1K	953	1.00	1	3.17	5	1	1	2	5	6
		955	1.20	1.2	6.3	10	88	31	61	24.2	5.3
		989	1.17	1.11	3.44	5	23	5	8	4.2	5.4
	10K	9792	1.00	1	5.7	9	24	2	8	8.62	3.7
		9449	1.20	2.4	17.7	30	1196	408	500	40	3
FW	1K	815	1.26	1.6	5.1	10	47	40	58	7	8.36
		964	1.00	4.8	1.9	6	1	5	2	17.95	61
		838	4.00	2.2	9.5	14	401	88	147	9	4.56
	10K	9001	2.02	5.27	1.36	11	841	120	564	16.8	4.16
		8909	4.80	6.1	8.4	19	2377	897	545	9.36	4.5
IPC	1K	963	1.10	1.7	3.2	6	49	6	25	12.4	4.08
		644	2.20	1.3	1.1	4	3	1	1	6.9	43
	10K	9515	1.70	2	10.5	19	722	334	99	23.3	3.6
		10000	1.00	1	1	1	0	0	0	4.36	-

TABLE V

UPDATE COSTS OF BLACK-WHITE SPLIT

Type	**Size**	**#Insert Rules**	**Γ_{down}**			**Black-White Split**			
			Meas. Max. Update Cost	**Avg. Update Cost**	**#Empty Entries**	**Meas. Max. Update Cost**	**Avg. Update Cost**	**#Empty Entries**	**Empty Entries Overhead**
ACL	1K	632	9	3.32	1	2	1.03	1	0.08%
	10K	6621	42	7.15	9	12	1.03	19	0.14%
	20K	13798	54	5.30	18	18	1.01	56	0.20%
FW	1K	1355	95	4.42	23	26	1.28	12	0.44%
	10K	15991	676	16.85	901	482	1.53	4516	14.12%
	20K	34275	1801	15.73	2027	747	1.66	15887	23.18%
IPC	1K	653	56	7.26	4	15	1.32	29	2.22%
	10K	6299	856	15.76	167	263	2.35	3033	24.08%
	20K	13635	1557	16.85	235	454	3.62	6573	24.10%

TABLE VI

AVERAGE PROCESSING TIME USING THE THREE SUB-RULESETS GENERATED BY BLACK-WHITE SPLIT

	ACL(μsec)	**FW**(μsec)	**IPC**(μsec)
1K	7	11	7
10K	57	215	70
20K	149	573	277

A. Update Cost of Split Rulesets

While the performances presented in the previous section are good enough for small rulesets, we saw that in several cases with more than 10K rules the number of updates could be still too large for a practical usage. We therefore have to use splitting in order to reduce the number of update-related operations. We will extend here the results to 100K rulesets, as they are only manageable with splitting the ruleset.

We show in Table IV the normal and reordering update performance of rulesets split by Black-White split into 3 subsets R_1, R_2 and R_3. For most cases, $UC_{\min}(R_i)$ and $UC_{\max}(R_i)$ are very close to each other, and even the simple TCAM ordering Ω_0 achieves best average update cost. We therefore show for each sub-rulesets only $UC(\Omega_0)$, along with the worst case, *i.e.*, the longest update path. We are assuming here that new incoming rules are inserted in the ruleset with the lowest average update cost such that the update performance of the splitted ruleset becomes the lowest average update cost among all sub-rulesets. We show the improvement ratio in the average update complexity in the last column of Table IV. As can be seen, splitting the ruleset using the Black-White Split results in a very significant improvement of the normal update cost, with a ratio of $3 \sim 40\times$ and of the reordering case, with $3 \sim 60\times$. Similar to the case without split, we have evaluated experimentally the update performance of the sub-rulesets resulting from Black-White Split over a synthetic stream of updates. We show the performance results with using Γ_{down} in Table V. We see that the measured maximum and average cost with reordering of the three sub-rulesets are consistently better than the non-splitted case, with improvement ratio $3 \sim 11\times$. However, using the sub-rulesets entails a larger number of empty entries, *e.g.*, for FW and IPC rules with more than 10K rules the overhead resulting from empty entries goes up to 24% . This can be explained by a larger number of needed reorderings, resulting from the applied update policy that inserts the new rules into the sub-ruleset with the smallest average update cost, ignoring the reordering. Nevertheless, the overhead is still much less than alternative approaches. We are leaving for future works the search for other update policies that will improve TCAM utilization and minimize the update costs.

We show in Table VI the average processing time in the control plane for inserting a new micro-rule into one of the three sub-rulesets resulting from Black-White Split. Since Black-White Split can significantly reduce the number of updates in reordering situations and Γ_{down} only needs to scan the sub-ruleset for each insert, the processing time is also reduced significantly compared to Table III. A churn rate of 1745 updates per second is therefore sustainable with a single thread running on a single core.

B. Comparison With TreeCAM

In this section, we compare the performance of two ruleset splitting algorithms: the Black-White Split algorithm with the TreeCAM proposed in [21]. It is noteworthy that [21] describes the reordering case (described in Section III-C) as "*updates that cause chains to merge*". However, the authors of [21] decided to not consider these cases, as it "*would require all the merged chains' rules to be reordered in the TCAM, which may move more rules than the longest update path*". In order to have a fair comparison, we will only compare the average update cost without considering the reordering cost. Moreover, we implement an optimistic version of TreeCAM, which does not consider the *Global Re-balancing* that is needed in some cases, *i.e.*, the obtained average and max update cost for TreeCAM are optimistic, and they are likely to be higher in real implementation.

For the evaluated rulesets, we observed that TreeCAM splitting frequently results in $7 \sim 9$ sub-rulesets, while we found in our experiments that by using Black-White Split, 3 sub-rulesets are usually enough for achieving a small cost. This means that since modern TCAMs have usually 4 parallel

TABLE VII
COMPARING BLACK-WHITE SPLIT WITH TREECAM

Type	Size	TreeCAM				Black-White Split			BW Split .vs. TreeCAM	
		Avg. Update Cost	**Max Update Cost**	**#sub-rulesets**	**Rule dup.**	**#sub-rulesets**	**Avg. Update Cost**	**Max Update Cost**	**Ratio of Avg.Update Cost**	**Ratio of Used TCAM space**
ACL	1K	6.26	49	5	1.58	3	1.2	10	0.19	63.14%
	10K	7.28	57	5	1.33	3	1.2	30	0.16	75.40%
	20K	4.51	57	5	1.31	3	1.3	24	0.29	76.55%
	100K	2.78	57	5	1.46	3	1.02	15	0.37	68.27%
FW	1K	10.98	41	9	1.41	3	3.97	14	0.36	70.85%
	10K	6.53	57	9	3.57	3	4.79	19	0.73	27.98%
	20K	5.31	57	7	10.6	3	5.03	24	0.95	9.39%
	100K	7.50	57	7	3.34	3	1.82	10	0.24	29.91%
IPC	1K	12.62	57	9	1.36	3	1.11	6	0.09	73.73%
	10K	9.64	57	9	1.08	3	1.73	19	0.18	92.30%
	20K	4.46	57	9	1.04	3	1.19	23	0.27	95.70%
	100K	7.34	57	9	1.89	3	1.74	60	0.24	52.97%

query ports, our scheme is able to achieve small update costs using only **one** TCAM chip. We show the comparisons in Table VII. It can be observed that in all scenarios, Black-White splits achieved better performance than TreeCAM in all performance criterion: max and average update cost and TCAM-used space. The improvement ratio of the average update cost ranges from 1.05 to 11.1. In the worst case seen in the table (for IPC 100K) the number of needed updates TCAM is 60, which is equivalent to 150 nsec of update time, the arrival time for 12 packets of 64 bytes over a 40 Gbps link. The needed TCAM space is lower by a coefficient ranging from 1.04 for the 20K IPC ruleset to 10.64 for the 20K FW ruleset. The difference in the needed TCAM space is caused by the rule duplication in TreeCAM splitting, while Black-White Splitting does not duplicate rules.

VII. RELATED WORKS

There is already very rich literature on TCAM packet classification algorithms. Most of the literature has dealt with packet classification performance [1], [3], [4], [7], [8], [16], [19], [22]. By revealing some fundamental algorithmic limits in dealing with large and complex rulesets, several recent works have proposed to split the ruleset into smaller subsets. In this direction, EffiCuts [22] leverages range “wideness” on each rule field to partition rulesets and build independent search trees on each subset. SAX-PAC [7], [8] also partitions rulesets by the rule fields, and uses TCAM and SRAM to perform lookups. SmartSplit [4] follows the same partioning idea and leverages ruleset characteristics to choose the most efficient search algorithm for each subset. The Black-White split algorithm in this paper follows the similar idea to split the ruleset to reduce update complexity.

There are relatively fewer works targeted at TCAM updating. We have already discussed Chain Ancestor Ordering (CAO) [17] in Section III-B2. Compared to CAO, Γ_{down} is able to manage multi-dimensional ruleset both for normal and reordering update cases.

In [18], Song and Turner present another update algorithm that groups non-overlapping rules and allocates for each group an index that follows rule priorities. When inserting a new rule, its index inside the group is inserted in the TCAM, along with the rule mask, and indices in other existing rules that are impacted are adjusted. Rules within the same group are in fact non-comparable. However, in the reordering update case, inserting a new rule may make several previously non-comparable rules comparable. In such a case, several group indices need to be changed, and the number of TCAM operations for changing these indices might become prohibitive. This explains the bad update performance reported by [18]: even for 1K rulesets, the update cost can become as high as 1000 TCAM operations. Moreover, as rules are stored in TCAM along with their group indices, we need for each incoming packet to run several binary searches to find in the TCAM the correct group index of the matched rule. This reduces search throughput. Nonetheless, even if it is not explicitly stated, the idea of grouping non-overlapping rules and giving them a separate index boils down to ruleset-splitting.

Another ruleset-splitting based approach is TreeCAM [21]. TreeCAM proposes to use decision trees to split rules into small blocks of overlapping rules, and bound the update overhead. TreeCAM is the highest performance method proposed before the Black-White splitting method presented in this paper. We have used TreeCAM as a comparison baseline to evaluate the performances of the proposed methods. On the application side, [10] proposed a hybrid architecture mixing TCAM and FPGA, that reduces TCAM's update cost by storing only non-overlapping rules in the TCAM, and pushing all overlapping rules into FPGA.

VIII. CONCLUSION

This paper studies systematically the TCAM update problem. We introduce partial orders over the ruleset to explain the fundamental properties of the TCAM update problem.

In particular, based on partial order theory, we were able to develop a generic update process and explore the design space of TCAM-updating algorithms. Three algorithms with different update strategies, named as Γ_{full}, Γ_{bh}, and Γ_{down}, are presented and discussed. While Γ_{bh} has the optimal predictable update cost, it requires high time complexity in calculating the position of rules, and therefore, we focus on Γ_{down} and perform an in-depth analysis on its update performance.

We used the Hasse diagram to derive precise maximal and minimal average update costs of Γ_{down} (without reordering), which enabled us to precisely evaluate the performance of Γ_{down} without knowing the update stream. In addition, we present a simple method that enables us to predict the lower bound of the worst case reordering update cost.

Looking at the ultimate achievable performance, we observed that for a relatively large set of practical rulesets, the minimal update cost is not yet compatible with operational constraints in terms of TCAM update blocking delays. We therefore proposed the Black-White Split algorithm, which leverages overlap graph properties and ruleset features to split the ruleset into 3 sub-rulesets. We evaluated both theoretically and empirically using a synthetic update stream, Γ_{down}, and the Black-White Split. The results showed very good update performance and excellent TCAM memory usage while confirming that the reordering case has a strong impact on the worst case update costs. We finally compared the Black-White Split with the state-of-the-art, TreeCAM. We showed that Black-White Split achieves in all scenarios better performance than TreeCAM, both in average update cost and in TCAM space usage.

References

[1] F. Baboescu and G. Varghese, "Scalable packet classification," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 199–210, 2001.
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[3] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. Hot Interconnects VII*, 1999, pp. 34–41.
[4] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *Proc. ICNP*, Oct. 2014, pp. 308–319.
[5] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite CacheFlow in software-defined networks," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 175–180.
[6] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "CacheFlow: Dependency-aware rule-caching for software-defined networks," in *Proc. ACM Symp. SDN Res. (SOSR)*, 2016, pp. 175–180.
[7] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and eXpressive PAcket classification)," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 15–26, 2014.
[8] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 1251–1264, Apr. 2016.
[9] A. X. Liu, E. Torng, and C. R. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *Proc. IEEE 27th Conf. Comput. Commun. (INFOCOM)*, 2008, pp. 176–180.
[10] L. Luo, G. Xie, Y. Xie, L. Mathy, and K. Salamatian, "A hybrid hardware architecture for high-speed IP lookups and fast route updates," *IEEE/ACM Trans. Netw.*, vol. 22, no. 3, pp. 957–969, Jun. 2014.
[11] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 335–346.
[12] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[13] C. R. Meiners, A. X. Liu, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," in *Proc. IEEE Int. Conf. Netw. Protocols (ICNP)*, 2007, pp. 266–275.
[14] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "vCRIB: Virtualized rule management in the cloud," in *Proc. NSDI*, 2013, pp. 1–6.
[15] M. H. Overmars and F. A. van der Stappen, "Range searching and point location among fat objects," *J. Algorithms*, vol. 21, no. 3, pp. 629–656, 1996.
[16] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2016.
[17] D. Shah and P. Gupta, "Fast updating algorithms for TCAM," *Micro, IEEE*, vol. 21, no. 1, pp. 36–47, Jan./Feb. 2001.
[18] H. Song and J. Turner, "Fast filter updates for packet classification using tcam," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Nov./Dec. 2006, pp. 1–5.
[19] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 135–146, 1999.
[20] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," in *Proc. IEEE 24th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 3. 2005, pp. 2068–2079.
[21] B. Vamanan and T. N. Vijaykumar, "TreeCAM: Decoupling updates and lookups in packet classification," in *Proc. 7th Conf. Emerg. Netw. Experim. Technol.*, 2011, p. 27.
[22] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 207–218, 2010.
[23] F. Zane, G. Narlikar, and A. Basu, "Coolcams: Power-efficient TCAMs for forwarding engines," in *Proc. INFOCOM*, vol. 1. Mar./Apr. 2003, pp. 42–52.

Peng He, photograph and biography not available at the time of publication.

Wenyuan Zhang, photograph and biography not available at the time of publication.

Hongtao Guan, photograph and biography not available at the time of publication.

Kavé Salamatian, photograph and biography not available at the time of publication.

Gaogang Xie, photograph and biography not available at the time of publication.