Adaptive Batch Update in TCAM: How Collective Optimization Beats Individual Ones

Ying Wan[1], Haoyu Song[2], Yang Xu[3], Chuwen Zhang[1], Yi Wang[4, 5*], Bin Liu[1, 5*]
[1] Tsinghua University, China [2] Futurewei Technologies, USA [3] Fudan University, China
[4] Southern University of Science and Technology, China [5] Peng Cheng Laboratory, China
wany16@mails.tsinghua.edu.cn, wy@ieee.org, lmyujie@gmail.com

IEEE INFOCOM 2021 - IEEE Conference on Computer Communications | 978-1-6654-0325-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/INFOCOM42981.2021.9488758

***Abstract*—Rule update in TCAM has long been identified as a key technical challenge due to the rule order constraint. Existing algorithms take each rule update as an independent task. However, emerging applications produce batch rule update requests. Processing the updates individually causes high aggregated cost which can strain the processor and/or incur excessive TCAM lookup interrupts. This paper presents the first true batch update algorithm, ABUT. Unlike the other alleged batch update algorithms, ABUT collectively evaluates and optimizes the TCAM placement for whole batches throughout. By applying the topology grouping and maintaining the group order invariance in TCAM, ABUT achieves substantial computing time reduction yet still yields the best-in-class placement cost. Our evaluations show that ABUT is ideal for low-latency and high-throughput batch TCAM updates in modern high-performance switches.**

I. Introduction

As an indispensable component in modern network switches, Ternary Content Addressable Memory (TCAM) [1] is used for rule tables supporting functions like packet forwarding [2], access control [3], traffic inspection [4], and flow filtering [5]. The capability of parallel search on ternary encoded rules on the one hand sustains a peerless lookup throughput, but on the other hand makes the rule update, especially the new rule insertion, difficult. This is because ternary rules can overlap. A lookup key dropped in the overlapping region of some rules matches all of them. TCAM only returns the index of the first matching rule. Therefore, rules must be stored in TCAM in order of decreasing priority. At runtime when any new rule needs to be added to TCAM, it must be placed in an empty entry without violating the priority relationship with existing ones. In the absence of such an entry, some incumbent rule needs to be relocated to make one, which may lead to further rule moves. During this process, TCAM lookup is suspended.

It is crucial to make TCAM updates low cost to meet application requirements. The cost of a TCAM update process comprises two parts: the computing cost t^c and the placement cost t^p (*i.e.*, the number of TCAM operations required to finish the update). The former determines the CPU resource consumption and the latter determines the TCAM lookup bandwidth usage. A high computing cost strains the CPU and impairs its capability to handle other device control and management tasks. An excessive placement cost can either cause packet drops [6] or make the system fail to sustain the update requests [7], [8].

In reality, applications impose strict update delay and throughput requirements [9]. Fast failure recovery leaves no more than 10ms for a routing table update [10]; traffic engineering grants only a 20ms budget to activate a new policy [11], [12]; Software-Defined Networking (SDN) [13] introduces a high policy churn rate [14]–[20]. In prospect, the throughput of high-end switches already reaches the level of 12.8Tbps per chip [21] and the increase is relentless, but TCAM's bandwidth does not scale as fast, which means the bandwidth ratio allocated for updates shrinks [22]. Meanwhile, the emerging Intent-Driven Networking (IDN) [23] and autonomous networks will apply faster rule updates in realtime through closed control loops without human intervention. In consequence, we expect the TCAM update rate to keep increasing beyond the status quo.

No wonder the TCAM update problem was and remains a hot research topic. However, most of the algorithms proposed in recent years are designed to handle individual rule update with the optimization preference for either t^c or t^p. Even if multiple pending updates are present, these algorithms can only process them independently, resulting in additive t^c and t^p. A few algorithms allege to support batch updates [15], [24], but in fact they still rely on individual update techniques, and only seek opportunities to reduce the rule moves in the final TCAM placement, given the moving plan of the batch. While their t^p is moderately improved, their t^c remains additive and dominates the total cost.

Meanwhile, many TCAM applications indeed present a batch or bursty update pattern [16]. First, TCAM can be used as a cache for hot rules on switch dataplane fast path [7], [25]. Unlike the other types of cache, a TCAM cache usually replaces a number of rules together each time [26]–[28]. For instance, wildcard rule update scheme CAB proposed in [27], [28] requires installing multiple rules within a bucket for each rule installation request to guarantee the semantic correctness of wildcard rule caching. Previous works mainly consider the policies for rule replacement, but ignore the cost associated with the updates.

Second, in SDN and IDN, the TCAM rules are converted from high-level policies [29]–[35]. A policy with d W-

This work is supported by National Key R&D Program of China (2019YFB1802600), "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019)", NSFC (62032013, 61872213, 61432009), NSFC-RGC (62061160489), and Shanghai Pujiang Program (2020PJD005). Corresponding Author: Yi Wang, Bin Liu.

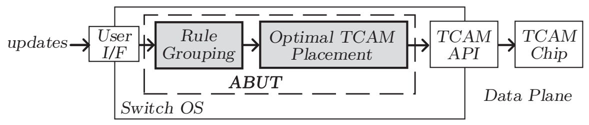


Fig. 1. The architecture of ABUT.

bit range fields may be converted into up to W^d TCAM rules [32]. A well-known example is the port range to prefix translation [36]. It also happens when some other rule fields cannot be aggregated. These rules need to be bundled together and inserted into TCAM in one atomic transaction. Failing to do so may violate high-level policy semantics and introduce errors to network operation [37].

Third, traffic engineering [11] and fast rerouting algorithms [38] often require changing multiple forwarding rules at the same time due to network failure or traffic pattern shift. In such realtime applications, update inconsistency and high cost can cause extended micro-loops and black holes for traffic.

Fourth, to synchronize the rule table states among switches and between the controller and the switch data plane, an SDN controller needs to coordinate [18], [39], [40], schedule [41], [42], and compress the updates [41]. Also, during an update, which can last for hundreds of milliseconds, some commercial switches (*e.g.*, HP-5406zl and Pica8-3290) [43] are found to stop receiving further updates [44]. In these scenarios, individual updates can easily accumulate to form a batch.

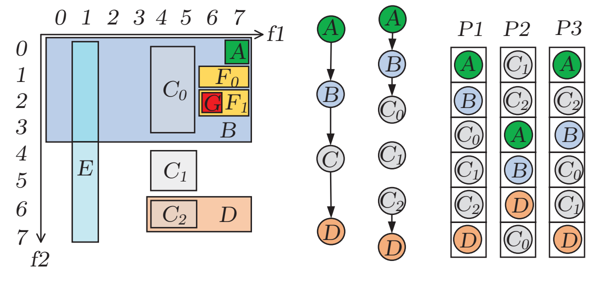
Last but not the least, we found no shortage of the batch update requirements in conventional applications, such as IP source guard [45], Bidirectional Forwarding Detection (BFD) for Pseudo Wire (PW) failover [46], and ACLs for the Remote Authentication Dial In User Service (RADIUS) [47]. A single change on network condition or configuration often induces many rule updates which need to take effect simultaneously.

Such TCAM applications challenge the existing algorithms, motivating us to rethink the problem from a holistic perspective. We consider a batch of updates as one single logical update, for which the overall update performance is concerned. An update is done only when the last rule in a batch is settled. Both the update throughput and delay are measured in terms of a batch. Such a perspective provides us a fresh opportunity to improve the update performance. The resulting adaptive batch update algorithm, ABUT, supports the aforementioned applications and shows a better aggregated performance than previous algorithms. That is, if a batch contains n rules to be inserted, ABUT achieves that $T_b^c < \Sigma_{i=1}^{n} t_i^c$ and $T_b^p < \Sigma_{i=1}^{n} t_i^p$, where T_b is the batch update cost induced by ABUT, and t_i is the individual rule update cost induced by previous algorithms.

As shown in Fig. 1, the high-level architecture of ABUT comprises two core algorithm components. ABUT integrates the TCAM management and the rule table management, and guarantees the update consistency. ABUT can also be used for general TCAM applications. In case the new rule's effectiveness is without urgency, one can collect a set of deletion and insertion updates during a proper time window and process them as a batch, aiming to minimize the update impact to the system performance. Even for single-rule updates, ABUT can still be appreciated for its simplicity and high performance.

TABLE I
A RULE TABLE WITH TWO MATCHING FIELDS.

Rule	pri	spec		act	Rule	pri	spec		act
		f1	f2				f1	f2	
A	9	111	000	a	D	0	1**	110	d
B	6	***	0**	b	E	2	001	***	e
C_0	4	10*	0**	c	F_0	7	11*	001	f
C_1	4	10*	10*	c	F_1	7	11*	010	f
C_2	4	10*	110	c	G	8	110	010	g



(a) Overlapping relationship (b) DAGs (c) 3 placements

Fig. 2. Rules in space, rule/policy graph, and placements in TCAM.

The remainder of the paper is organized as follows. Section II provides the background. Section III discusses the related works. Section IV and V describe the two algorithm components of ABUT. Section VI presents the implementation and evaluation. Finally, Section VII concludes the work.

II. Background

A high-level policy can be converted to one or more ternary-encoded rules suitable for TCAM. Each policy is assigned a unique priority value, which is inherited by the converted rules. A larger value means a higher priority. Some algorithms keep the cognate rules together in consecutive TCAM entries, but it is unnecessary. Later we will show that considering them as independent rules in a batch promises a better update performance. Table I gives a rule table example used throughout the paper. A rule $r = (pri, sp, act)$ is represented by three attributes: priority, field specification, and action. In the table, $\{C_0, C_1, C_2\}$ and $\{F_0, F_1\}$ are originated from the same high-level policy C and F, respectively. Specified by two fields $f1$ and $f2$, each rule is embodied as a box on a 2D plane as shown in Fig. 2(a).

Rule Relation Graph: The rule relation graph is a Directed Acyclic Graph (DAG), in which each vertex denotes a rule. A directed edge $r_i \rightarrow r_j$ indicates that r_i and r_j overlap and r_i has a higher priority than r_j. The topology order of the vertices on the same path reflects the relative priorities of the corresponding rules and determines their mandatory order in TCAM. If two vertices are not on the same path, the order of their corresponding rules in TCAM does not matter, regardless of their assigned priority values. This is a crucial foundation for many TCAM update algorithms including ours.

Fig. 2(b) shows the DAGs for the policies and rules A to D, respectively. Fig. 2(c) shows three possible TCAM placements, P1, P2, and P3, which obey the topology order of the rules. If $\{C_0, C_1, C_2\}$ are bound together, only the first placement is possible. If instead we consider them as independent rules, the placement is more flexible, and as a consequence, the update performance can also benefit.

Per Rule Priority: Most existing algorithms take the assigned priority value as a rule's true priority. The original rule table is placed in TCAM in the order of decreasing priority values. A new rule r needs to be inserted above all the lower priority rules and below all the higher priority rules that overlap with r. If no empty entry is available, in theory, a range of incumbent rules can be considered as candidates for relocation. Different algorithms diverge in the criterion that the candidates are selected and in the direction that the rules are allowed to move. Given a search space to explore, the recursive process aims to find a scheme with low t^p, while taking t^c as a trade-off.

Such algorithms can eventually make the rules in TCAM out of the original priority value order, although the placement still obeys a topology order of the DAG. This may raise a troublesome "reorder" problem [48] when the introduction of a new rule r breaks the current topology order. For example, assume two rules r_i and r_j are not on the same DAG path, and due to the previous updates, r_i is located above r_j despite r_i has a lower priority value. If the new rule r happens to overlap with both r_i and r_j, and it has a higher priority than r_i but a lower priority than r_j, now r_i and r_j's relative order must be switched before the insertion of r. This cannot be done by just swapping r_i and r_j, because of their own topology order constraints. Instead, either r_i or r_j needs to be reinserted, which is equivalent to another insertion update. The reorder problem, while neglected by some previous algorithms [49], [50], happens at a stark probability of up to 28.79% [24].

Per Group Priority: Some other algorithms reduce the number of priority values by grouping rules. All the rules in a same group share the same priority and their relative order can be arbitrary [37], [51]. One only needs to make sure each rule is assigned to a correct group and maintain the group order. The reorder problem will never happen in this case. A notable example is using TCAM for IP address Longest Prefix Match (LPM) tables in which prefixes are grouped and prioritized based on their length. The number of priorities is determined by the number of prefix groups but not the rule table size. A simple algorithm supports the rule insertion with the moving cost bounded by the group number [52]. Although its t^c is attractive, the t^p can be far from optimal.

Topology-Order Priority Grouping: The number of priority groups can be further compressed with the help of the rule relation DAG. As long as two rules are not on the same path, they are possible to be grouped together. Specifically, we use the following procedure to group the rules. All the vertices without any successor are grouped together and assigned the priority value '0'. For any other vertex, after all its direct successors have their priority values assigned, it will be assigned a priority value that is one plus the largest priority value of its direct successors. All the vertices (*i.e.*, rules) with the same priority value are grouped together, using the priority value as the group ID (GID). It is easy to see that the number of groups is the same as the length of the longest DAG path.

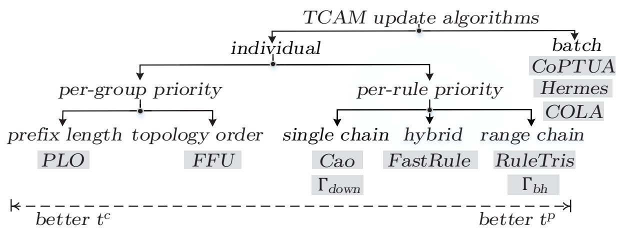


Fig. 3. Classification of TCAM update algorithms.

When applying an insertion update based on per group priority, the t^p is bounded by the group number. For the LPM tables, topology-order priority grouping results in much fewer groups than prefix-length grouping. For the general multi-dimension rule tables, the number of groups may not be small enough to make individual rule updates impressive, but the batch updates can amortize the T^p, so the t^p per rule stands low. Needless to say that the T^c, already superior even for single rule update, is also shared by all the rules in a batch.

Update in Batch: Driven by the batch update requirements, ABUT is the first algorithm that follows the above principle and seeks the collective optimization opportunities. Specifically, ABUT is built in light of the following high-level ideas.

First, the holistic viewpoint of batch update allows multiple rules to be evaluated at the same time, making it possible to achieve a lower TCAM placement cost than the additive cost for updating each rule individually. The first observation is that the rule insertion order can affect the total placement cost [41], [44]. Considering the rules together allows us to find the best order within a batch. The second observation is that collectively producing the final TCAM placement for a batch can avoid redundant rule moves. For example, if a rule needs to be moved twice due to two individual updates, now we can place it in its final target entry in just one move.

Second, previous algorithms have little control on the empty entry distribution. An update algorithm can benefit from the even empty entry distribution, but actively dispersing empty entries in TCAM is undesirable due to the extra moving cost. While rule deletion can naturally generate empty entries, it has no guarantee on the distribution. Because of the abundant placement choices, the batch update algorithm is possible to adapt to any initial placement and casually achieve even distribution of empty entries without extra cost.

III. Related Work

The existing TCAM update algorithms can be classified based on their technical features as shown in Fig. 3. Most algorithms are designed for individual updates. Among these algorithms, the earlier works adopt the priority grouping method featuring a small t^c. PLO [52] groups LPM rules based on the prefix length. FFU [51] groups general rules based on their topology order. The worst-case t^p subjects to the number of groups for these algorithms.

To improve t^p, later algorithms start to use each rule's priority value and its position in the rule relation DAG in lieu of the priority grouping. A clear trade-off on selecting the candidate rules to move distinguishes these algorithms. Cao [52] and Γ_{down} [48] only consider a single candidate (*i.e.*, the rule's closest successor or predecessor) in each recursive step. The evaluated rules form a single chain. On the other extreme, Γ_{bh} [48] and RuleTris [49] evaluate all the feasible candidates at each step, leading to a near-optimal t^p at a high t^c. FastRule [50] gives up some t^p gain in exchange of a smaller t^c. Only for the new rule, are all the candidates considered for FastRule; in each recursive step, FastRule regresses to the single chain approach as in Γ_{down} and Cao.

A few works tackle the batch update problem. CoPTUA [6] focuses on maintaining table consistency and lookup throughput during batch updates. The method actually increases the t^p. The complex table management and batch processing also increase the update latency. Hermes [15] uses a small logical shadow table to process batches of updates and migrates the changes to TCAM periodically. As a system architecture, Hermes lacks an underlying batch update algorithm. COLA [24] relies on the priority-based individual update algorithms to calculate the moving scheme for each rule in a batch first and then jointly considers the final TCAM placement. While its T^p is improved, its T^c is still additive and subjects to the poor t^c of the individual updates.

Due to the scalability issue of TCAM [53], in recent years it becomes popular to use TCAM as a cache rather than for a complete lookup table. Most works focus on improving the hit rate of the TCAM cache [7], [27], [54]. Less attention is paid on the algorithm supporting efficient and consistent batch updates. The aforementioned individual and semi-batch update algorithms are ill-suited for TCAM cache because of their high t^c, t^p, or long deployment delay.

IV. Incremental Topology Order Rule Grouping

A. Preliminary

ABUT is based on the topology order priority grouping, which is optimal in terms of the number of resulting groups [37], [51]. For any r, its GID $r.gid$ is assigned as:

$$r.gid = \begin{cases} 0 & r\ has\ no\ successor \\ max\{r'.gid\}+1 & r' \in successors\ of\ r \end{cases} \tag{1}$$

$r.gid$ is in essence the minimum height of its corresponding vertex in the DAG. While we can group the initial rules using topology sorting with a time complexity of $O(V+E)$, in which V is the number of vertices and E is the number of edges in the DAG, it is too expensive to run the regrouping algorithms once for each update. In fact, rule insertions or deletions can only affect the GIDs of their predecessors. A strawman approach is therefore to collect and work on only their predecessor rules. Unfortunately, this approach still involves too many rules to warrant a good performance. Therefore, we develop a fast incremental regrouping algorithm as the first key component of ABUT.

Both rule insertion and deletion may change some rules' GIDs. However, rule deletion does not change the topology order of the remaining rules in the DAG. Other than nullifying the corresponding TCAM entries for the deleted rules, we only mark their direct predecessors to be cared for, and postpone the regrouping until rule insertions are needed.

The newly inserted rules also need to get their GIDs calculated, so these rules are also marked in the DAG, and their GIDs are initialized to be '-1'. At this point, the GIDs of all the marked rules need to be calculated. The change of a rule's GID can in turn cause other rules to change their GIDs, making it important to maintain a proper order to avoid redundant and invalid calculations.

Our incremental regrouping algorithm is based on the following observations (assume r' is r's direct predecessor). (1) The change of r's GID can only affect the GIDs of its predecessors. If we evaluate and update the GID in the increasing priority value order, the GIDs of the visited rules will not change any more and the regrouping is guaranteed to finish in one pass. (2) In such an evaluation order, if $r.gid$ is changed, we only need to mark those r' whose GIDs are possibly affected, but not others. If the change propagates further and more rule's GIDs are affected, the evaluation order guarantees to attend to them eventually. (3) According to the above marking criterion, when r is deleted, r' should be marked only if $r'.gid == r.gid+1$. Any other value of $r'.gid$ means it is not directly derived from r. (4) Similarly, after a marked rule r's GID is calculated, if $r.gid$ is increased from x to y, r' should be marked only if $r'.gid \leq y$; if $r.gid$ is decreased from x to y, r' should be marked only if $r'.gid == x+1$. r cannot directly affect r' with other GID values. (5) If a marked rule's GID turns out unchanged after evaluation, no further rule marking is needed.

B. Algorithm Description

Starting from the DAG and the GID assignment for the initial rule set, we sketch the incremental regrouping algorithm.

To maintain the order for GID calculation, we augment each vertex in the DAG another pointer which is used to link all the rules in the order of increasing priority values. To keep track of the marked rules, we augment each vertex a flag. A "True" flag indicates that the GID of the corresponding rule needs to be evaluated.

At first, only the inserted rules and some direct predecessors of the deleted rules are marked. The linked list is scanned from the head. The GID of each marked rule encountered is evaluated. If its GID is changed, some of its direct predecessors that meet the marking criterion are marked. The flag is cleared once a marked rule is processed. The regrouping is done after the linked list is scanned.

Algorithm 1 lists the pseudo code for the incremental regrouping algorithm. The performance gain of the algorithm comes from two aspects. First, it reduces the time complexity to $O(V+e)$ in which e is a small subset of edges in the DAG. Second, the linked list traversal is much faster than the recursive depth-first search in topology sorting.

Example: Fig. 4 illustrates an example of the incremental regrouping process based on rules in Table I. Assume the

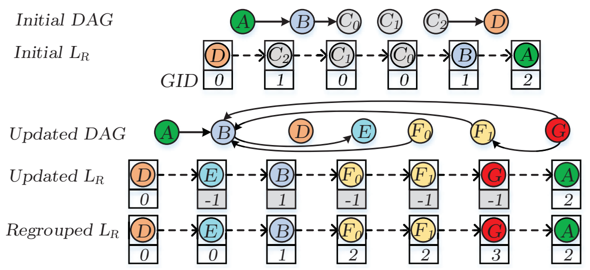


Fig. 4. An example of the fast incremental regrouping.

Algorithm 1: Incremental Topology Order Rule Grouping

```
Input: rule graph G=(V, E); the linked list $L_R$; set of
       {existing, inserted, deleted} rules {$R_e$, $R_i$, $R_d$}.
Output: GIDs of rules in $R_e$+$R_i$-$R_d$ are assigned correctly
for each deleted rule $r$ in $R_d$ do
    remove $r$ from $L_R$
    for each $r$'s direct predecessor $r'$ do
        if $r'$.gid == $r$.gid+1 then $r'$.flag = true
for each inserted rule $r$ in $R_i$ do
    add $r$ into $L_R$, $r$.flag = true, $r$.gid = $-\infty$
for ($r$ = $L_R$.head(); $r \neq L_R$.end(); $r$ = $L_R$.next()) do
    if $r$.flag == true then
        r_pre_gid = $r$.gid
        if $r$ has at least one successor then
            $r$.gid=max{$r'$.gid | $r' \in$direct successors of $r$}+1
        else $r$.gid = 0
        if r_pre_gid $\neq$ $r$.gid then
            for each $r$'s direct predecessor $r'$ do
                if $r$.gid+1 > $r'$.gid then $r'$.flag=true
                if r_pre_gid+1 == $r'$.gid then $r'$.flag=true
        $r$.flag = false
```

original table only contains rules A to D. The batch update requests to remove rules C_0 to C_2, and insert new rules E to G. The original DAG and linked list are shown in the top portion of the figure. In the linked list we label each rule with its GID. The updated DAG and liked list after the rule removal and the new rule insertion are shown in the middle portion of the figure. The initial set of marked rules are highlighted. In addition to the new rules, rule B is marked because of the deletion of C_0.

The linked list is scanned from rule D. Rule E is the first marked rule. Since it has no successor, its GID is set to 0. Rule B is E's only direct predecessor. Since $B.gid == E.gid+1$, there is no need to mark B, although in this case B is already marked due to the deletion of rule C_0.

Now the scan advances to the next marked rule B. After calculation, B's GID is not changed, so there is no need to examine B's direct predecessors. This process continues until all the linked list nodes are visited. The grouping result is shown in the lower portion of Fig. 4.

V. Optimal TCAM Placement

A. *Preliminary*

TCAM update is fundamentally a TCAM placement problem for the updated rule set, with the objective for low incremental cost. Given that any topology order placement of the rule set is legitimate [51], to reduce the search space, ABUT only examines the subset of the placements that the rules follow the decreasing GID order after the topology grouping, but allows arbitrary rule orders within the same group.

Assume n rules are placed in m TCAM entries T[0:$m-1$], and the rules are distributed in k groups with the size of N_i for the i-th group. Under such conditions, the number of legitimate placements N_L is still as large as:

$$N_L = (\prod_{i=0}^{k-1} N_i!) * \binom{n}{m} = (\prod_{i=0}^{k-1} N_i!) * \frac{m!}{n!(m-n)!} \qquad (2)$$

Clearly it is infeasible to conduct brute-force search in such a large space. ABUT manages to find the optimal TCAM placement in this space with a time complexity of $O(m*n)$. Before delving into the algorithm, we first explain how the placement cost is evaluated.

Updating a TCAM involves only two types of operation: (1) Nullify an existing rule (*i.e.*, empty an entry), and (2) Write a rule into an entry, which can concomitantly nullify the original rule if the entry is not empty. To evaluate the update performance more precisely, we count the total number of write operations (c_w) and nullify operations (c_n) to achieve a placement as the placement cost (*i.e.*, $t^p = c_w + c_n$).

Assume the original TCAM placement is L, and after update it becomes L'. The calculation of t^p needs to consider the following five cases. (1) T[i] was empty in L but is occupied in L'. This increases c_w by 1. (2) T[i] was occupied in L but is empty in L'. This increases c_n by 1. (3) T[i] is occupied in both L and L'. Assume in L, after regrouping, the GID of the rule in T[i] is x, and in L', the GID of the rule in T[i] is y. If $x \neq y$, we know the previous rule and the current rule are from two different groups, so they must be different. In this case, a write operation is required, which increases c_w by 1. (4) The condition is similar to the above case, but $x == y$, which means that the previous rule and the current rule are from the same group. Since we allow arbitrary rule orders in the same group, we can keep the previous rule in situ from L to L'. This case incurs no cost. (5) T[i] is empty in both L and L', which incurs no cost as well.

When transforming from L to L', in order to achieve the optimal placement cost, we should minimize the occurrence of the first three cases and maximize that of the last two.

B. *Algorithm Description*

Finding the min cost: We calculate the optimal placement cost through dynamic programming. Assume m is the total number of TCAM entries and n is the total number of rules to be placed (including the inserted rules but excluding the deleted ones). We use a 2D matrix C[0:m][0:n] with $m+$

Algorithm 2: Optimal TCAM Placement

```
Input: initial placement T[0:m-1]; set of {existing, inserted,
       deleted} rules {$R_e$, $R_i$, $R_d$}; the linked list $L_R$
Output: The operations for the Optimal TCAM placement
1  IncrementalTopologyOrderRuleGrouping($L_{R_e}$,$R_i$,$R_d$)
2  $R=R_e+R_i-R_d$, $n$=R.size()
3  sort the rules in R in GID decreasing order
4  C[0][0]=0 // INIT for finding the minimal placement cost
5  for i in range (1, m+1) do
6      C[i][0] = C[i-1][0] + (T[i-1].gid$\neq$-1), A[i][0] = "$\downarrow$"
7      for j in range (1, n+1) do
8          $C_1$ = C[i-1][j] + (T[i-1].gid $\neq$ -1)
9          $C_2$ = C[i-1][j-1] + (T[i-1].gid $\neq$ R[j-1].gid)
10         if $C_1 < C_2$ then C[i][j] = $C_1$, A[i][j] = "$\downarrow$"
11         else if $C_1 > C_2$ then C[i][j] = $C_2$, A[i][j] = "$\searrow$"
12         else if (i-1)-(j-1) > (m-n)/m*(j-1) then
13             C[i][j] = $C_1$, A[i][j] = "$\downarrow$"
14         else C[i][j] = $C_2$, A[i][j] = "$\searrow$"
15 i = m, j = n, $T_o$[0:m-1] = -1 //INIT for finding the placement
   while i $\geq$ 1 do
16     if A[i][j] == "$\searrow$" then
17         $T_o$[i-1].gid = R[j-1].gid, i = i-1, j = j-1
18     else i = i-1
19 $S_w$.clear(), $S_n$.clear(), $R_w = R_{ins}$ //INIT for finding operations
20 for j in range (0, m) do
21     if $T_o$[j].gid == T[j].gid then continue
22     if $T_o$[j].gid == -1 then $S_n$.add(j)
23     if $T_o$[j].gid $\neq$ -1 then $S_w$.add(j)
24     if T[j].gid $\geq$ 0 then $R_w$.add(original rule in T[j])
25 for i in range (0, $S_n$.size()) do
26     nullify the TCAM entry T[$S_n$[i]].
27 sort the rules to be written in $R_w$ in decreasing GID order
28 sort the entries for writing in $S_w$ in increasing address order
29 for j in range (0, $R_w$.size()) do
30     write the rule $R_w$[j] into the TCAM entry T[$S_w$[i]].
```

1 rows and $n+1$ columns to keep the intermediate results. C[i][j] holds the minimum placement cost if the first i TCAM entries T[0:i-1] are used to place the first j rules R[0:j-1]. C[m][n] is exactly the final result we look for. In initialization, C[0][0] is set to 0; since C[i][j] for $j > i$ is impossible, the value of these elements is set to $+\infty$.

We use T[i].*gid* to denote the GID of rule placed in T[i] and R[$\cdot$].*gid* to denote the decreasing group order that must be enforced. Specifically, T[i].*gid*$=-1$ if T[i] is empty, and T[i].*gid*$=-2$ if the old rule in T[i] is one of the deleted rules. With this notation, we have two ways to place the first j rules in the first i entries. (1) Assume we have found the optimal way to place the first j rules in the first $i-1$ entries, then we just need to make the i-th entry empty. (2) Assume we have found the optimal way to place the first $j-1$ rules in the first $i-1$ entries, then we just need to place the j-th rule in the i-th entry. The optimal way is the one that results in a lower cost, which can be described by the following recursive step:

$$\text{C}[i][j]=min\begin{cases}\text{C}[i\text{-}1][j] + (\text{T}[i\text{-}1].gid \neq \text{-}1)\\ \text{C}[i\text{-}1][j\text{-}1] + (\text{T}[i\text{-}1].gid \neq \text{R}[j\text{-}1].gid)\end{cases} \quad (3)$$

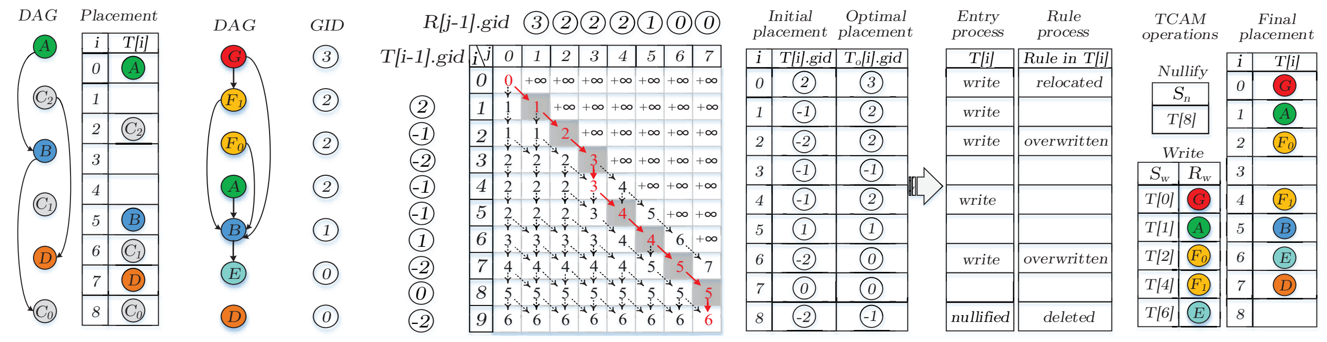
In the first way, the i-th entry must be emptied. So if it is currently occupied (*i.e.*, T[i-1].*gid*$\neq$-1) by r, no matter r is one of the deleted rules or it should be relocated, a nullify operation is needed, increasing the total cost by 1. In the second way, if the j-th rule's GID, R[j-1].*gid*, is different from T[i-1].*gid* (as a result of three possibilities: (1) T[i-1] is empty, (2) It currently holds one of the deleted rules, or (3) It holds a rule with different GID), a write operation to T[i] is needed, increasing the total cost by 1. Otherwise, if T[i-1].*gid*==R[j-1].*gid*, based on the earlier discussion on Case 4, we can keep the old rule in T[i] in situ to avoid increasing the cost.

Example: We use our previous example to illustrate the dynamic programming process. The original rule graph and the TCAM placement are shown in Fig. 5(a). The updated rule graph (now containing 7 rules) and the regrouping result are shown in Fig. 5(b). The new placement needs to place 7 rules in 9 entries. In Fig. 5(c), T[$\cdot$].*gid* reflects the TCAM layout we start with. Since the rules C_0, C_1, and C_2 are deleted, the GIDs of their corresponding entries T[2], T[6], and T[8] are set to -2. The GIDs of the empty entries T[1], T[3], and T[4] are set to -1. The remaining entries T[0], T[5], and T[7] are occupied by the original rules A, B and D. Their GIDs are set to the updated GID of the rule in it accordingly. The dynamic programming process starts from C[0][0] and progresses towards C[9][7]. In this example, the optimal placement cost is 6, implying the minimum number of write and nullify operations is 6.

Finding the placement: Only knowing the optimal cost is not enough. We need to obtain the actual placement and the required operations. We augment the dynamic programming process with a little extra data that helps us keep track of the progression paths. When calculating C[i][j], we also record from which way the value is derived. In case of a tie, either way can be taken—later we will discuss how we can take advantage of this fact to achieve automatic and arbitrary empty entry distribution. The arrows in Fig. 5(c) indicates the progression directions. Following the arrows, any path from C[0][0] to C[m][n] represents an optimal placement.

At any grid (i, j) on such a path, whenever the value increases by 1, either a nullify or a write operation on T[i-1] is needed. These entries are collected in two sets, S_n and S_w, respectively. For such an entry, if C[i][j] is derived from the first way (*i.e.*, by a vertical arrow), then T[i-1] should be empty, so T[i-1] belongs to S_n. Otherwise, if C[i][j] is derived from the second way (*i.e.*, by an oblique arrow), then T[i-1] should be occupied, so T[i-1] belongs to S_w. In this case, if T[i-1] was occupied by r and r is not one of the deleted rules, we add r in the set of new rules because it needs to be reinserted somewhere else.

The update plan is therefore as simple as (1) Nullifying the entries in S_n, and (2) In any decreasing GID order, writing the set of new rules into the entries in S_w. In some applications, the TCAM operations are interleaved with the lookup operation. In order to maintain the lookup consistency, in the second step, we simply place the relocated rules first



(a) Before update (b) Rule regrouping (c) Calculation of optimal placement (d) Finding the placement operations (e) After update

Fig. 5. An example of dynamic programming for optimal TCAM placement.

before placing the other new rules.

Example: There are 9 equal-cost optimal placements possible for our example. Fig. 5(c) highlights one of the progression path, and Fig. 5(d) illustrates the corresponding placement and required operations. This specific placement requires 5 write operations—one of them for relocating A—and 1 nullify operation. The operations and the final TCAM placement are shown in Fig. 5(e).

Adaptive empty entry distribution: The distribution of the empty entries in TCAM can affect the placement cost. Almost all algorithms benefit from the empty entries closing to the rule insertion locations. Since one cannot predict the locations for future insertions, the best strategy is to evenly distribute the empty entries throughout the TCAM space. However, existing algorithms have no control on the empty entry distribution. They either passively take advantage of the empty entries left by rule deletions, or manually move the empty entries to some specific locations at extra cost.

In contrast, ABUT supports arbitrary empty entry distribution as a concomitant process without extra cost. During the dynamic programming process, whenever the two ways in Eq. 3 give the same cost, we are free to choose either one. The first way (*i.e.*, keeping the TCAM entry empty) is equivalent to pushing the empty entry down, and the second way (*i.e.*, keeping the TCAM entry occupied) is equivalent to pushing the empty entry up. If adhering to the first (second) way, eventually all the empty entries will concentrate at the lower (upper) part of the TCAM. This effect enables us to design an adaptive algorithm for even empty entry distribution, keeping TCAM in an ideal state for future updates.

Since we have $m{-}n$ empty entries in total, if we would like to evenly distribute these empty entries, we would expect to have one empty entry every $m/(m{-}n)$ entries. Or inversely, during the dynamic programming, for the first i entries, we should try to maintain the expected empty entry density $(m{-}n)/m$. That is, whenever a way selection opportunity emerges at grid (i, j), if $(i{-}j)/(i{-}1){>}(m{-}n)/m$, the first way is chosen to reduce the empty entry density by pushing the empty entry down; otherwise the second way is chosen.

Summary: Algorithm 2 lists the pseudo code of the optimal TCAM placement algorithm. Composed of the topology grouping and the optimal placement, ABUT has a overall time complexity of $O(m * n)$ regardless of the batch size. In our example, the batch update contains 3 deletions and 4 insertions, but the placement cost is only 6, which means ABUT needs less than one TCAM operation per rule update, beating the best case for all individual TCAM update algorithms.

In the actual implementation, we manage to only store the Boolean type variables in the 2D matrix to indicate the progression direction, while maintaining an array for the necessary cost values. For a 4K-entry TCAM, this optimization reduces the required memory for computation from 80MB to 16MB. This may not be a big deal for a server, but is worthwhile when the algorithm runs on the embedded or on-board management CPU in switches.

VI. Implementation and Evaluation

A. Experiment Setup

We choose Single Chain (SC) and Range Chain (RC) as the representatives for the individual TCAM update algorithms. For fair and sound comparisons, our implementations of SC and RC fix some flaws in the existing algorithm embodiments. Specifically, we detect and solve the reorder problem which was ignored by some previous algorithms, and allow rules to move bidirectionally so empty entries in TCAM can be fully utilized. We also compare ABUT with COLA [24], the semi-batch update algorithm. COLA uses either SC or RC as the underlying individual update algorithms. The two variations are labeled as COLA_SC and COLA_RC, respectively. All the algorithms are implemented in C++.

We use 26 rule tables summarized in Table II for algorithm evaluation. Due to the privacy and security concerns, we can only access the real LPM tables from CAIDA [55] and the Stanford backbone network [56]. The multi-field rule tables are acquired, as a convention, from the synthesis tools ClassBench [57] and ClassBench-ng [58]. ClassBench-ng can generate rules with more than 5 fields.

Unless otherwise mentioned, t^c is measured by the real computing time per updated rule and t^p is measured by the number of TCAM operations per updated rule. To test the algorithms' scalability and their sensitivity to different parameters, we vary the rule table type, batch size, TCAM capacity, and TCAM fill rate, respectively. Note that the modern switches predominantly use only on-chip TCAM due

TABLE II
RULE TABLES USED FOR PERFORMANCE EVALUATION

Type	Name	Source	Feature	Field #	Size
LPM	cd1 - cd10	CAIDA	real	1	∼1M
LPM	sf1, sf2	Stanford	real	1	∼90K
ACL	acl1 - acl5	ClassBench	synthetic	5	10K
Firewall	fw1 - fw5	ClassBench	synthetic	5	10K
IP Chain	ipc1, ipc2	ClassBench	synthetic	5	10K
Openflow	of1, of2	ClassBench-ng	synthetic	9	10K

to ultra-high throughput and I/O limitation, and the TCAM capacity needs to be partitioned to support multiple logical tables [59]–[61], so a single TCAM table contains at most a few thousand entries. However, in the TCAM scalability test, we still extend the TCAM capacity to 16K.

B. ABUT Examination

We first examine several key factors that contribute to ABUT's performance.

Rule Grouping. Fig. 6 shows the rule table size versus the number of topology order rule groups. The actual sizes of *ipc*, *acl*, and *fw* are larger than that of their corresponding synthesized tables because of the conversion from range to prefix on the port fields [62]. The number of groups ranges from a few to a few dozens. The small number paves the foundation for the two algorithm components of ABUT.

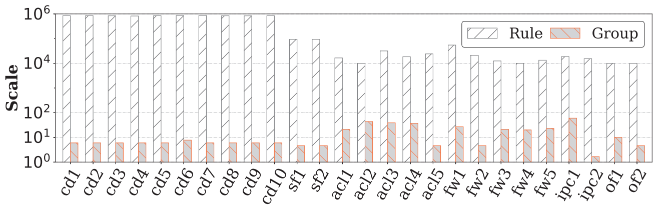


Fig. 6. The number of rules vs. the number of groups.

Regrouping Performance. Fig. 7 shows the running time of ABUT's incremental regrouping algorithm versus that of the naive approach that runs topology sorting on each batch, under the combinations of different batch size θ and rule table size m. The incremental regrouping is 20 to 40 times faster. While the running time has a linear relationship with the table size, it is not sensitive to the batch size. The time per rule will decrease quickly as the batch size increases.

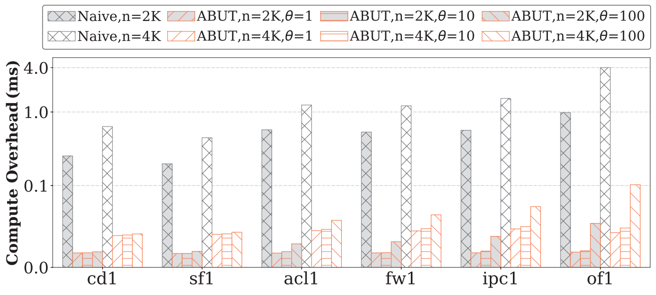


Fig. 7. Naive regrouping vs. ABUT's incremental regrouping.

Effect of Empty Entry Distribution. We examine the consequence of six different empty entry distribution strategies by modifying the ABUT implementation. The empty entries can be pushed to the bottom, the top, the middle, or both ends

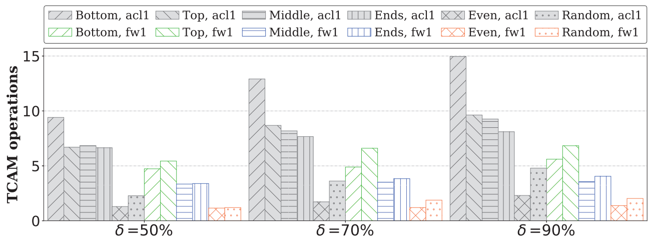


Fig. 8. Empty entry distribution strategies under m=4K and θ=50.

of the TCAM space. It is also easy to make the distribution random or even (as described in Section V). Under different TCAM fill rate δ, we measure the t^p for randomly inserting 100 new rules. As shown in Fig. 8, the experiments confirm that different strategies have a significant impact on t^p and the even distribution of empty entries is the best, so it is used as the default strategy in ABUT.

C. Algorithm Comparisons

Performance on LPM tables. We first compare the update performance on the single-field LPM table *cd1*. Fig. 9 shows the t^c and t^p for ABUT and the other algorithms. In this case, TCAM is typically used as a cache (*i.e.*, the fill rate δ is 100%). The rule caching and replacement decisions are calculated by CacheFlow [7], using an hour-long real packet trace [63]. The cache refreshing period is set to 1 minute. Under such a configuration, the batch size is 304, 485, and 522 on average when m is 2K, 4K, and 8K, respectively. The average results on 60 refresh cycles are shown in Fig. 9. ABUT is the only one algorithm that requires less than 2 operations per rule update, meanwhile with the least t^c.

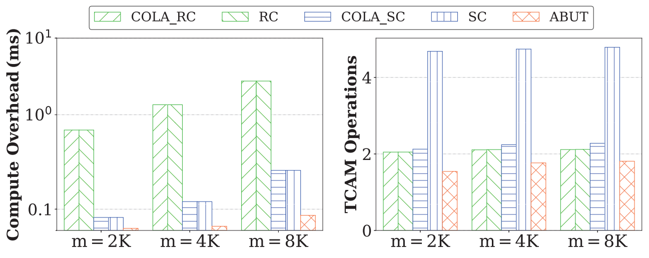


Fig. 9. Update performance on LPM tables.

Performance on multi-field rule tables. We compare the update performance on several typical multi-field rule tables. Fig. 10 shows the results when m, θ, and δ are set to 4K, 50 and 80%, respectively. The rules in each batch are chosen randomly and the result is the average of 40 runs. Again, ABUT's t^c is up to 600∼1200x and 2∼4x shorter than RC/COLA_RC and SC/COLA_SC, respectively. ABUT's t^p is also the best.

Scalability on batch size θ. Fig. 11 shows the algorithm performance on two typical rule tables, *acl1* and *fw1*, by varying the batch size. The rules in each batch are chosen randomly and the result is the average of 40 runs. ABUT's t^c becomes the best when θ is greater than 10, while ABUT's t^p remains the best for all the batch sizes. As θ increases, all the batch update algorithms including COLA show a better t^p.

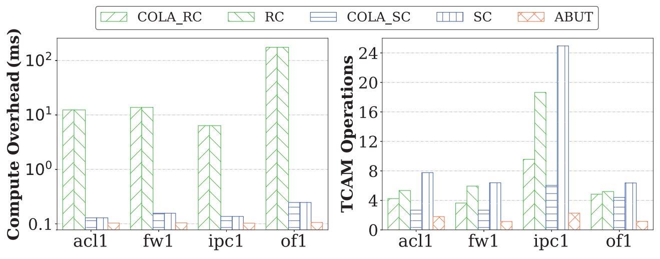


Fig. 10. Update performance on multi-field rule tables.

According to the finding in [26], [64], the typical batch size θ is often greater than 100, for which ABUT is much more preferable.

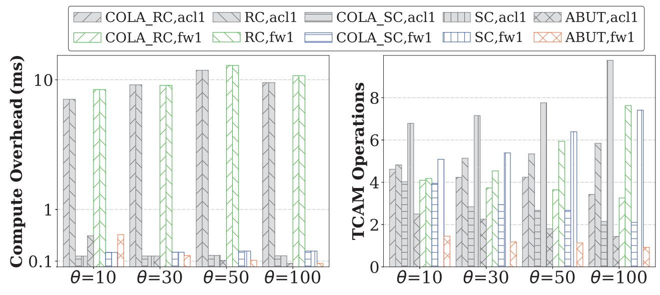


Fig. 11. Update performance with θ for $acl1$ and $fw1$, m=4K, δ=80%.

Scalability on TCAM capacity m. Fig. 12 shows the performance of the algorithms under different TCAM capacity. The t^c of ABUT is 40∼100x shorter than that of RC/COLA_RC and 1.1∼4x shorter than that of SC/COLA_SC. While the t^c advantage of ABUT over SC/COLA_SC decreases with the increase of m, ABUT's t^p is stable and consistently beats other algorithms with a large margin.

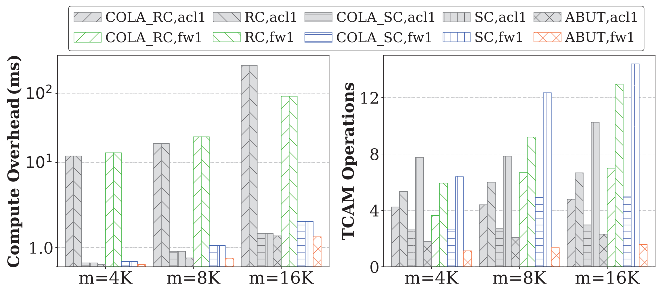


Fig. 12. Update performance with m for $acl1$ and $fw1$, θ=50, δ=80%.

Scalability on TCAM fill rate δ. Fig. 13 shows the performance of the algorithms under different TCAM fill rates. When δ is 100%, we randomly delete 50 rules first and then randomly insert 50 rules, to emulate the cache operations. All the algorithms are not very sensitive to the TCAM fill rate, but generally a higher fill rate does require higher t^c and t^p for updates. The t^c of RC and the t^p of SC show a clear increasing trend when δ is less than 100% and we only insert new rules to TCAM, because the occurrence of the reorder problem becomes more frequent as δ increases. ABUT consistently outperforms the other algorithms for any TCAM fill rate. ABUT's performance is also more stable because it is immune to the reorder problem.

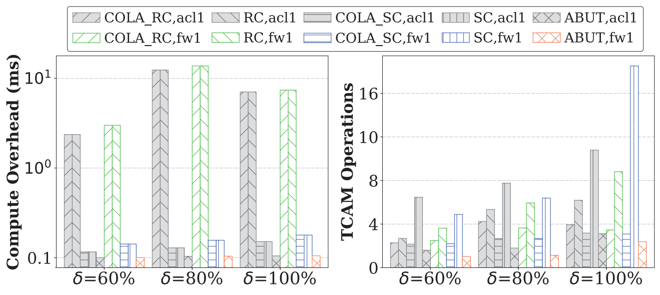


Fig. 13. Update performance with δ for $acl1$ and $fw1$, θ=50, m=4K.

Computing time vs. TCAM operations. The absolute time is more revealing in weighing the real impact of t^c and t^p to the system. The update latency for a batch is the sum of the two, and the update throughput is determined by the larger of the two. A TCAM operation can finish in less than a microsecond due to its high clock frequency, while the per-rule computing time for the existing algorithms is in the order of milliseconds. The huge discrepancy suggests that t^c is the dominant factor impacting the update performance, and therefore deserves more optimization efforts. In this regard, ABUT is the most balanced algorithm so far with the exceptional t^c and top-rank t^p. Fig. 14 shows the per-batch update latency comparisons. In the figure, the lower part of each bar filled with the star pattern represents the portion of latency due to t^p, and the rest represents the portion due to t^c, using the parameters suggested by Xilinx [65] (*i.e.*, TCAM frequency is 170MHz and one operation takes 33 clock cycles). ABUT shows stable and small update latency regardless of the batch size and supports much higher update throughput than others for moderate to large batch sizes. The simplicity of ABUT will become more important in high performance switches, because while terabit per second switch chips are commonplace, "the management CPU on most switches is relatively wimpy" [66].

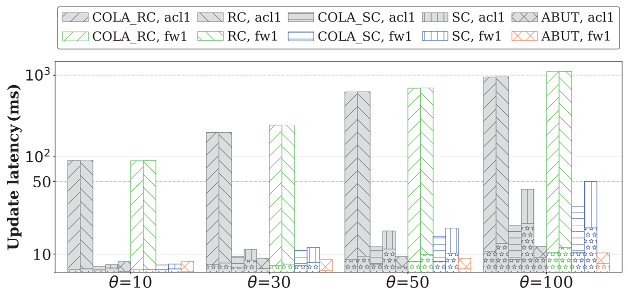


Fig. 14. Update latency per batch with θ for $acl1$ and $fw1$, m=4K.

VII. Conclusion

ABUT is the first true batch update algorithm tailored to the requirements for bursty and batch TCAM updates in high-performance switches. Through extensive evaluations, we showcase the collective optimization on batches, with predictable lower cost and higher scalability, can indeed beat the algorithms relying on individual updates. Thanks to its simplicity, ABUT is practical and ready to be integrated into the TCAM-based rule table management system supporting existing and emerging applications.

REFERENCES

[1] B Salisbury. TCAMs and OpenFlow-what every SDN practitioner must know. *See http://tinyurl.com/kjy99uw*, 2012.
[2] Kirill Kogan, Sergey I Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. Exploiting order independence for scalable and expressive packet classification. *TON*, 2015.
[3] Haesung Hwang, Shingo Ata, Koji Yamamoto, Kazunari Inoue, and Masayuki Murata. A new TCAM architecture for managing ACL in routers. *IEICE Transactions*, 2010.
[4] S Jayashree and N Shivashankarappa. Deep packet inspection using ternary content addressable memory. In *I4C*, 2014.
[5] Zhijun Wang et al. A TCAM-based solution for integrated traffic anomaly detection and policy filtering. *Computer communications*, 2009.
[6] Zhijun Wang, Hao Che, et al. CoPTUA: Consistent policy table update algorithm for TCAM without locking. *TOC*, 2004.
[7] Naga Katta et al. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *SOSR*, 2016.
[8] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. vCRIB: virtualized rule management in the cloud. In *HotCloud*, 2012.
[9] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *SIGCOMM*, 2014.
[10] B Niven-Jenkins, D Brungard, M Betts, N Sprecher, and S Ueno. Requirements of an MPLS Transport Profile. *RFC5654*, 2009.
[11] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.
[12] Abhinav Pathak, Ming Zhang, Y Charlie Hu, Ratul Mahajan, et al. Latency inflation with MPLS-based traffic engineering. In *IMC*, 2011.
[13] Jean Tourrilhes, Puneet Sharma, Sujata Banerjee, and Justin Pettit. The evolution of SDN and OpenFlow: a standards perspective. *IEEE Computer Society*, 2014.
[14] Geng Li, Yichen Qian, Chenxingyu Zhao, Y Richard Yang, and Tong Yang. DDP: Distributed Network Updates in SDN. In *ICDCS*, 2018.
[15] Huan Chen and Theophilus Benson. Hermes: Providing tight control over high-performance SDN switches. In *CoNEXT*, 2017.
[16] Geng Li et al. Update Algebra: Toward Continuous, Non-Blocking Composition of Network Updates in SDN. In *INFOCOM*, 2019.
[17] Sushant Jain et al. B4: Experience with a globally-deployed software defined WAN. *SIGCOMM CCR*, 2013.
[18] Jiaqi Zheng, Hong Xu, Guihai Chen, and Haipeng Dai. Minimizing transient congestion during update in datacenters. In *ICNP*, 2015.
[19] Wei Zhang, Guyue Liu, Ali Mohammadkhan, Jinho Hwang, KK Ramakrishnan, and Timothy Wood. SDNFV:Flexible and dynamic software defined control of application-and flow-aware data plane. In *IMC*, 2016.
[20] Klaus-Tycho Foerster, Stefan Schmid, et al. Survey of consistent software-defined network updates. *IEEE Commun. Surv. Tutor.*, 2018.
[21] Broadcom. At a Glance: Tomahawk 3 is the first 12.8 Tb/s chip to achieve mass production. https://www.broadcom.com/blog.
[22] Mehdi Malboubi, Liyuan Wang, Chen-Nee Chuah, and Puneet Sharma. Intelligent SDN based traffic (de) aggregation and measurement paradigm (iSTAMP). In *INFOCOM*, 2014.
[23] Victor Heorhiadi, Sanjay Chandrasekaran, et al. Intent-driven composition of resource-management SDN applications. In *CoNEXT*, 2018.
[24] Baolan Zhao, Rui Li, Jin Zhao, and Tilman Wolf. Efficient and Consistent TCAM Updates. In *INFOCOM*, 2020.
[25] Kirill Kogan et al. Sax-pac (scalable and expressive packet classification). In *SIGCOMM*, 2014.
[26] Zixuan Ding et al. Update Cost-Aware Cache Replacement for Wildcard Rules in Software-Defined Networking. In *ISCC*, 2018.
[27] Bo Yan, Yang Xu, Hongya Xing, et al. CAB: A reactive wildcard rule caching system for software-defined networks. In *HotSDN*, 2014.
[28] Bo Yan, Yang Xu, and H Jonathan Chao. Adaptive wildcard rule cache management for software-defined networks. *TON*, 2018.
[29] Pavel Chuprikov, Kirill Kogan, and Sergey Nikolenko. How to implement complex policies on network infrastructure. In *SOSR*, 2018.
[30] Ori Rottenstreich et al. On finding an optimal TCAM encoding scheme for packet classification. In *INFOCOM*, 2013.
[31] Anat Bremler-Barr and Danny Hendler. Space-efficient TCAM-based classification using gray coding. *Transactions on Computers*, 2010.
[33] Alex X Liu, Chad R Meiners, and Eric Torng. TCAM Razor:A systematic approach towards minimizing packet classifiers in TCAMs. *TON*, 2009.
[32] Ori Rottenstreich and Isaac Keslassy. Worst-case TCAM rule expansion. In *INFOCOM*, 2010.
[34] Kirill Kogan, Sergey Nikolenko, William Culhane, Patrick Eugster, and Eddie Ruan. Towards efficient implementation of packet classifiers in SDN/OpenFlow. In *HotSDN*, 2013.
[35] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using graphs to express and automatically reconcile network policies. *SIGCOMM CCR*, 2015.
[36] Huan Liu. Efficient mapping of range classifier into ternary-CAM. In *Proceedings 10th Symposium on High Performance Interconnects*, 2002.
[37] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.
[38] Mike Shand and Stewart Bryant. IP fast reroute framework. Technical report, RFC 5714, January, 2010.
[39] Aggelos Lazaris et al. Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization. In *CoNEXT*, 2014.
[40] Jeffrey C Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. Corybantic: towards the modular composition of SDN control programs. In *HotNets*, 2013.
[41] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.
[42] Hongli Xu, Zhuolong Yu, Xiang Yang Li, Liusheng Huang, Chen Qian, and Taeho Jung. Joint Route Selection and Update Scheduling for Low-Latency Update in SDNs. *TON*, 2017.
[43] Maciej Kuźniar, Peter Perešíni, Dejan Kostić, and Marco Canini. Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches. *Computer Networks*, 2018.
[44] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about SDN flow tables. In *PAM*, 2015.
[45] F Baker. Cisco IP Version 4 Source Guard. *Work in Progress*, 2007.
[46] T Nadeau et al. Bidirectional Forwarding Detection for the Pseudowire Virtual Circuit Connectivity Verification. *RFC5885*, 2010.
[47] Carl Rigney et al. Remote authentication dial in user service, 2000.
[48] Peng He et al. Partial order theory for fast TCAM updates. *TON*, 2018.
[49] Xitao Wen, Bo Yang, Yan Chen, Li Erran Li, Kai Bu, Peng Zheng, Yang Yang, and Chengchen Hu. RuleTris: Minimizing rule update latency for TCAM-based SDN switches. In *ICDCS*, 2016.
[50] Kun Qiu et al. FastRule: Efficient Flow Entry Updates for TCAM-Based OpenFlow Switches. *JSAC*, 2019.
[51] Haoyu Song and Jonathan Turner. Nxg05-2: Fast filter updates for packet classification using TCAM. In *IEEE Globecom*, 2006.
[52] Devavrat Shah and Pankaj Gupta. Fast incremental updates on Ternary-CAMs for routing lookups and packet classification. In *HOTI*, 2000.
[53] Kirill Kogan et al. Fib efficiency in distributed platforms. In *ICNP*. IEEE, 2016.
[54] Wan Ying, Haoyu Song, Yang Xu, et al. T-cache: Dependency-free Ternary Rule Cache for Policy-based Forwarding. In *INFOCOM*, 2020.
[55] CAIDA. Data collection,curation and sharing. https://www.caida.org.
[56] Stanford backbone router configuration. http://tinyurl.com/o8glh5n.
[57] David E Taylor and Jonathan S Turner. Classbench: A packet classification benchmark. *TON*, 2007.
[58] Jiří Matoušek et al. Classbench-ng: Recasting classbench after a decade of network evolution. In *ANCS*, 2017.
[59] Nexus 9000 TCAM Carving. http://goo.gl/wXC0KY.
[60] NOVISWITCH. http://noviŒow.com/products/noviswitch/.
[61] Understanding and Configuring Switching Database Manager on Catalyst 3750 Series Switches. http://goo.gl/nLziyq.
[62] Alexander Kesselman, Kirill Kogan, et al. Space and speed tradeoffs in TCAM hierarchical packet classification. *JCSS*, 2013.
[63] The CAIDA UCSD Anonymized Internet Traces[20180315]. www.caida.org/data/passive/passive_dataset.xml. Accessed in Mar, 2018.
[64] Xitao Wen et al. Compiling minimum incremental update for modular SDN languages. In *HotSDN*, 2014.
[65] Xilinx. TCAM Search IP for SDNet. https://www.xilinx.com/support/documentation/ip_documentation/tcam/pg190-tcam.pdf.
[66] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.