GreedyJump: A Fast TCAM Update Algorithm

Ying Wan, Haoyu Song, *Senior Member, IEEE*, and Bin Liu, *Senior Member, IEEE*

***Abstract*—TCAM is widely used for rule tables in network switches. An efficient update scheme, which requires as few rule moves as possible, is often time-consuming to compute, while the fast computation likely yields a mediocre update scheme. We studied the existing algorithms and found none is satisfactory to the emerging applications. In this letter, we present a simple greedy algorithm, *GreedyJump*, which achieves fast computation and efficient update scheme at the same time, both with the best-in-class performance. The evaluation qualifies *GreedyJump* as the default TCAM update algorithm ready for meeting the scalability challenges in various application scenarios.**

***Index Terms*—TCAM, update, greedy algorithm.**

I. Introduction

TCAM is ubiquitous in modern switches and plays a pivotal role in supporting line-speed policy-based packet filtering and forwarding [1]. However, as the single-chip switch bandwidth keeps increasing (the bandwidth of multi-terabits per second is on the horizon), the capacity and bandwidth of TCAM do not scale at the same pace. Meanwhile, the rule table size and the rule update frequency are both boosted by the highly dynamic applications. Therefore, it is important to improve the efficiency of TCAM usage. TCAM rule update—more specifically, new rule insertion—stands out as a fundamental problem with significant impact on switch performance [2], [3], [4].

A rule-insertion update involves two steps: (1) computing a TCAM rule moving scheme which preserves the rule's priority order, and (2) applying the scheme while temporarily halting the TCAM lookup process. The prevailing algorithms for computing the scheme are all based on the rule relation DAG [5]—after the new rule is inserted, the rules in TCAM must conform to a topological order.

The resulting schemes comprise a chain of moving steps: if a rule's target location is occupied, the incumbent rule needs to be relocated; the recursive step repeats until the evicted rule is settled in an empty entry. We name these algorithms the unidirectional chain-based algorithm, and categorize them into three types according to their optimization target.

- Single Chain (SC): In each recursive step, for the rule *r* to be inserted, the lowest TCAM address owned by *r*'s direct successors in the rule relation DAG (i.e., *r*'s nearest direct successor) is claimed. The evaluation process is simple and the computation time is linear to the DAG depth and the node fanout. However, the resulting scheme can be far from optimal. The representative algorithms of SC include Cao [6] and Γ_{down} [5].
- Range Chain (RC): Aiming to minimize the moving cost, in each recursive step, all the entries up to *r*'s nearest direct successors are considered for eviction, and the location that can lead to the fewest succeeding moving steps is taken. Such an algorithm uses a variation of dynamic programming and the computation time is proportional to the square of the rule table size. While the resulting scheme is superior, the computation cost can be beyond the affordable range. The representative algorithms of RC include Γ_{bh} [5] and RuleTris [7].
- Hybrid Chain (HC): As a trade-off, one can give up some moving cost gain in exchange of a smaller computation cost by mixing the ideas of RC and SC. For example, in the first step, all the entries up to *r*'s nearest direct successors are considered as in RC; in each subsequent step, only the nearest direct successor of the evicted rule is considered as in SC. Such an algorithm avoids leaning toward either extreme but neither its computation cost nor its moving cost is outstanding. The representative algorithm of HC is FastRule [8], [9].

TABLE I
Comparison of the Chain-Based Algorithms

Algorithm	SC	RC	HC	GJ
Computation Cost	low	high	moderate	low
Moving Cost	high	low	moderate	low

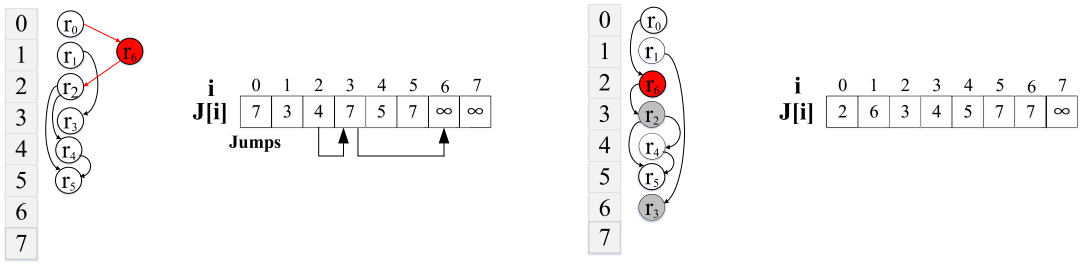
The qualitative comparison of these algorithms is summarized in Table I. The state of the art, while only able to excel in one of the two performance indicators, can hardly meet the requirements of emerging applications which deal with large and complex rule tables and demand frequent and bursty updates [10], [11], [12]. An algorithm with high computation cost consumes network control processor's resource, hinders the other critical switch control and management processes, and increases the rule deployment latency; an algorithm with high moving cost seizes too much TCAM bandwidth and impairs the normal lookup and forwarding performance.

Manuscript received February 9, 2021; revised March 30, 2021; accepted April 14, 2021. Date of publication April 20, 2021; date of current version March 3, 2022. The work of Ying Wan and Bin Liu was supported in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2019B1515120031; in part by the National Natural Science Foundation of China under Grant 61872213, Grant 62032013, and Grant 61432009; and in part by "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" under Grant LZC0019. The associate editor coordinating the review of this article and approving it for publication was M. Yuksel. *(Corresponding author: Bin Liu.)*

Ying Wan and Bin Liu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: wany16@mails.tsinghua.edu.cn; liub@mail.tsinghua.edu.cn).

Haoyu Song is with the Network Technology Lab, Futurewei Technologies, Santa Clara, CA 95050 USA (e-mail: haoyu.song@futurewei.com).

Digital Object Identifier 10.1109/LNET.2021.3074148



(a) Initial placement (b) Traverse J[:] (c) Final placement (d) Update J[:]

Fig. 1. Rule insertion by using GreedyJump.

TABLE II
PARAMETER DEFINITION

Symbol	Description
m	The number of TCAM entries
T[i], $0 \le i < m$	The i-th TCAM entry. T[0] is the TCAM top
T[i]→T[j]	Move the rule in T[i] to T[j]
r.ndp	The nearest direct predecessor of r
r.nds	The nearest direct successor of r
r.addr	The address of the TCAM entry in which r is placed
J[i]	The address of the furthest TCAM entry to which the rule r in T[i] can be moved, *i.e.*, J[i] = r.nds.addr

A pressing question is therefore: can we get the best of both worlds (i.e., achieve RC's moving cost with SC's computation cost)? We answer this question by presenting a new algorithm, GreedyJump (GJ). GreedyJump is based on a simple greedy algorithm. We prove that it achieves the identical cost (but not necessarily the exact same moving scheme) as RC; meanwhile, the experimental evaluations show that the computation cost is in line with SC and sometimes better.

An obvious optimization applicable to these algorithms is to evaluate the moving chain in both downward and upward directions and pick a better one. With this, the computation cost is doubled but the moving cost is only marginally improved. For simplicity, we only consider the downward moving direction for our algorithm and the comparison with others.

Related Work: The earlier priority-based update algorithms [6], [13] generate poorer schemes than SC so they have given way to the aforementioned chain-based algorithms. Recent batch update algorithms [14], [15] jointly consider the placement schemes for a batch of updates to reduce the moving cost. They still rely on the chain-based algorithms to compute each rule's moving scheme where GJ can play a role.

II. GREEDY JUMP

A. Algorithm Description

We use the example in Fig. 1 and the notations in Table II to explain the GreedyJump insertion update algorithm. Fig. 1(a) shows the rule graph and the initial rule placement in TCAM. Originally there are six rules, $\{r_0 \sim r_5\}$. When a new rule is to be inserted, any location between the entries occupied by its nearest direct predecessor (noninclusive) and its nearest direct successor (inclusive) is an eligible candidate. In our example, a new rule r_6, which overlaps with r_0 (its nearest direct predecessor) and r_2 (its nearest direct successor), needs to be inserted into the TCAM, and T[1] and T[2] are eligible.

In each recursive step, for the evicted rule, the candidate locations are from its original location (noninclusive) up to the entry occupied by its nearest direct successor (inclusive). For example, assume r_6 decides to take r_2's location T[2]. Since r_4 is the nearest among r_2's direct successors r_4 and r_5, r_2's candidate locations are T[3] and T[4].

Differing from the other chain-based algorithms, GreedyJump uses a greedy method to choose a rule's target location among the eligible candidates in each step. Simply put, an eligible candidate's location is taken if the evicted rule has the potential to move down the furthest (i.e., its nearest direct successor is the furthest). Later we will prove this near-sighted single-step method can indeed achieve an optimal chain-based moving scheme.

1) Building J[:]: GreedyJump maintains an array J[:] with the same number of entries as TCAM. Each entry J[i] records the TCAM address of the nearest direct successor of the rule in T[i]. Particularly, if the rule in T[i] has no successor, J[i] is set to the address of the last TCAM entry; if T[i] is empty, J[i] is set to ∞. It is easy to construct J[:] from the TCAM placement and the rule relation DAG. J[:] for the original rule set in Fig. 1(a) is shown in Fig. 1(b).

2) Traversing J[:]: The moving scheme is derived by traversing J[:] once. For example, for the new rule r_6, we have known that it can be inserted into either T[1] or T[2]. Since J[1] $<$ J[2], according to our greedy method, T[2] is selected. Now for the evicted rule r_2, we have known that it can be inserted into either T[3] or T[4]. Since J[3] $>$ J[4], T[3] is selected to insert r_2, and r_3 is evicted. Since J[3] indicates that r_3 can be moved to up to the end of TCAM, and J[6] $=$ J[7] $= \infty$, T[6] or T[7] can be selected to place r_3. The update scheme (r_6 →T[2]→T[3]→T[6]) is shown in Fig. 1(c). Algorithm 1 lists the pseudo code of the algorithm. Since the greedy method only traverses J[:] once to calculate the update scheme, its time and space complexities are both $O(m)$.

3) Updating J[:]: The TCAM placement change after each rule update requires J[:] to be updated accordingly. Instead of rebuilding J[:] from scratch, we apply an incremental update procedure.

We consider the changed entries first. For the entry T[i] that the new rule r_u is placed, we find the r_u's nearest direct successor r and set J[i] to r.addr. If a rule is moved from T[p]

Algorithm 1: TraverseJumpArray (r_u, J[:])

Input: r_u is the rule to be inserted; J[:] is an array.
Output: $\mathbb{S}$ records the moving sequence to insert r_u.

```
curPos = r_u.ndp.addr+1, endPos = r_u.nds.addr
jumpSrc = null, jumpDst = − 1
while (J[curPos] ≠ ∞) do
    if (J[curPos] > jumpDst) then
        jumpSrc = curPos, jumpDst = J[curPos]
    if (curPos == endPos) then
        endPos = jumpDst, S.push_back(jumpSrc)
    curPos++
S.push_back(curPos)
```

Algorithm 2: UpdateJumpArray (r_u, $\mathbb{S}$, J[:])

```
for (i = S.size()-1; i > 0; i−−) do
    J[S[i]] = J[S[i-1]]
for (r : r is the direct predecessor of r_u) do
    if J[r.addr] > r_u.addr then J[r.addr] = r_u.addr;
J[r_u.addr] = ∞
for (r : r is the direct successor of r_u) do
    if J[r_u.addr] > r.addr then J[r_u.addr] = r.addr
for (i = 1; i < S.size(); i++) do
    r : the rule currently placed in T[S[i]]
    for (r' : r' is the direct predecessor of r) do
        if J[r'.addr] == S[i-1] then
            for (r'' : r'' is the direct successor of r') do
                J[r'.addr] = ∞
                if J[r'.addr] > r''.addr then
                    J[r'.addr] = r''.addr
```

to T[q], J[q] is set to J[p], regardless of the influence of the other rule moves.

Next, we analyze the cases where J[p] for the other entries needs to change. For the new rule r_u in T[p] and each of its direct predecessors r in T[q], if r_u becomes the nearest direct successor of r (i.e., J[q] $> p$), J[q] is set to p; otherwise, if J[q] $< p$, J[q] does not change. For the rule r moved from T[p] to T[q] and its direct predecessor r' in T[k], if J[k] $== p$, it indicates that r was the nearest direct successor of r'. Since r is moved, we need to traverse all the direct successors of r' to find r''s nearest direct successor r'' and set J[k] to r''.addr; otherwise, if J[k] $< p$, indicating that r was not the nearest direct successor of r', the move of r will not change the nearest direct successor of r', so J[k] does not change.

The updated J[:] after inserting r_6 is shown in Fig. 1(d). Algorithm 2 lists the pseudo-code for incremental J[:] update. Although in the worst case it takes $O(m^2)$ time to update each element of J[:], which is equivalent to rebuilding J[:] from scratch, in practice the worst case is unlikely. Usually only a few rules are moved in the greedy method, and in turn only a few elements of J[:] need to be updated.

Rule Deletion: A rule deletion update, which simply invalidates a TCAM entry, also needs to update J[:] by using the same but simplified Algorithm 2.

B. Proof of Optimality

We prove by contradiction that GreedyJump achieves the minimum number of rule moves for the unidirectional chain-based algorithm.

Given an optimal moving chain S_O with p moves:

$$S_O : r_u \rightarrow T[a_0] \rightarrow T[a_1] \rightarrow \cdots \rightarrow T[a_{p-1}] \rightarrow T[e]$$

and the scheme S_G with q moves computed by GreeyJump:

$$S_G : r_u \rightarrow T[b_0] \rightarrow T[b_1] \rightarrow \cdots \rightarrow T[b_{q-1}] \rightarrow T[e]$$

T[e] is the first empty entry below r_u.

Assume that S_G requires more moves than S_O (i.e., $p < q$) and the k-th move of S_O (T[a_{k-1}]→T[a_k]→T[a_{k+1}]) is the first move that does not follow the greedy method used in S_G. That is,

$$\begin{cases} \text{(i) } \forall\ 0 \le i < k,\ a_i == b_i \\ \text{(ii) } a_k \neq b_k \\ \text{(iii) } J[a_k] < J[b_k] \end{cases} \quad (1)$$

TABLE III
RULE TABLE SIZE

Type	ACL										
$S(k)$	4	5	6	7	8	9	10	11	12	13	14
$S'(k)$	6	8	9	11	12	13	15	16	18	19	21

Type	FW										
$S(k)$	4	5	6	7	8	9	10	11	12	13	14
$S'(k)$	9	11	13	16	17	19	20	23	25	27	29

Now we show that it is legitimate to change the k-th rule move of S_O to be T[a_{k-1}]→T[b_k]→T[a_{k+1}]. First, since T[b_{k-1}]→T[b_k] in S_G is legitimate and $a_{k-1} == b_{k-1}$, T[a_{k-1}]→T[b_k] is also legitimate. As for T[b_k]→T[a_{k+1}], we prove that $b_k < a_{k+1}$: if $b_k \ge a_{k+1}$, since T[a_{k-1}]→T[b_k] has been proved to be legitimate, T[a_{k-1}]→T[a_{k+1}] must also be legitimate. In other words, the move to a_k in S_O is redundant, which is contradict with the optimality of S_O.

Now, since T[a_k]→T[a_{k+1}] is reasonable and J[a_k] $<$ J[b_k], the rule in T[b_k] can be directly moved to T[a_{k+1}]. Therefore, we can change the k-th rule move in S_O to be the same with that in S_G. Since the change does not introduce any extra rule move, the optimality of S_O is maintained.

In the same manner, we can modify every step in S_O that does not comply with the greedy method. Since the modifications do not incur extra rule moves, the final S_O still needs p moves, which contradicts with the assumption; hence we can infer $p == q$.

III. IMPLEMENTATION AND EVALUATION

A. Experiment Setup

We compare GJ with SC, RC, and HC. All the algorithms are implemented using C++ and run on a programmable switch EdgeCore Wedge100BF-32X [16], which has an Intel Xeon D-1517 CPU with the Open Network Linux operating system.

Due to the limited accessibility to real-world rule sets, as a common practice, synthetic rule sets bearing the characteristics of the real-world rule sets are used for algorithm evaluation. Various size rule tables for Access Control List (ACL) and

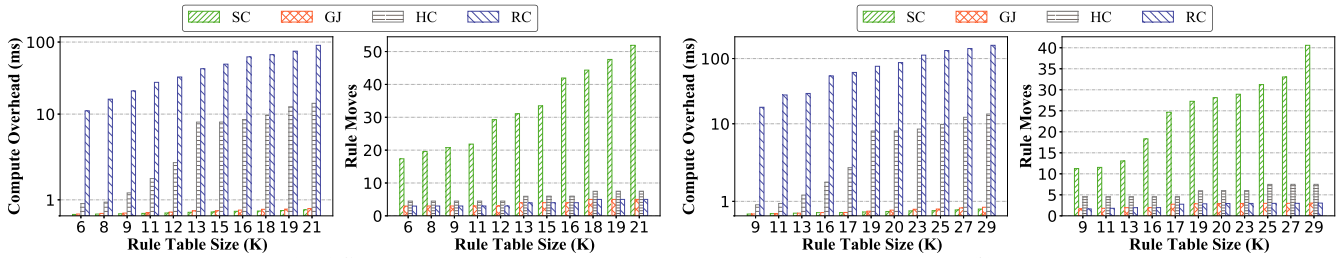


Fig. 2. Comparison of time consumption to insert a new rule to TCAM on two types of rule tables ACL and FW.

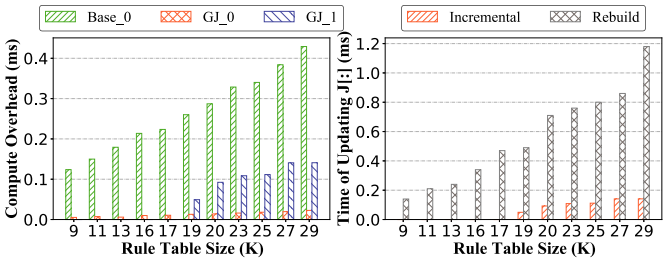


Fig. 3. Insight on GJ's computation time on FW tables.

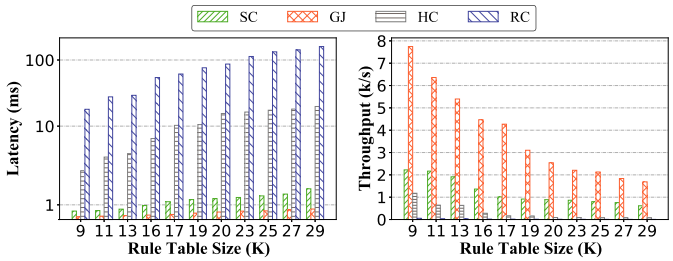


Fig. 4. Update latency and throughput on FW tables.

Firewall (FW), are generated by ClassBench-ng [17], [18]. The range rules are converted into prefix rules first to suit TCAM. The table sizes (before and after conversion) are summarized in Table III.

Given a rule table, we first insert 90% randomly chosen rules in TCAM, and then take the remaining 10% of the rules as updates. Since different algorithms may produce different moving schemes for a new rule and result in different TCAM placement layouts, to guarantee the identical comparison basis, for each update, we evaluate the computation time and rule moving scheme without actually inserting the rule into TCAM. In addition, we also evaluate the rule update latency and throughput which count in both the computation time and the moving scheme operation time (on EdgeCore Wedge100BF-32X, the time to perform a rule write in TCAM is 0.04 ms).

B. Experimental Results

Fig. 2 shows the algorithm performance under different rule types and different rule sizes. On the one hand, although all algorithms take more time to compute the TCAM update scheme as the rule table size increases, GJ and SC take less than 0.5ms for any table size, while HC and RC take more than 10ms and 100ms, respectively. On the other hand, GJ and RC demonstrate the same and the lowest moving cost, which is much better than that of SC. The moving cost of GJ and RC is also less sensitive to the rule table size.

The reason behind this is that GJ has the same complete search space as RC, so their results are both optimal; but the greedy algorithm adopted by GJ, which has a complexity similar to SC, is much more efficient than the dynamic programming adopted by RC.

We conduct an in-depth analysis of the computation time of GJ by breaking it down into three parts: (1) Base_0 represents the time to determine the overlapping relationship between the insertion rule and the existing rules, and update the rule graph; (2) CJ_0 represents the time to traverse J[:] to find the optimal update scheme; (3) CJ_1 represents the time to incrementally update J[:]. Base_0 is common for all the algorithms based on the rule graph; GJ_0 and GJ_1 are unique to GJ. Because of the limited space, we only show the experimental results on F W tables in Fig. 3.

We can see that Base_0 uses more than 80% of total time consumption and GJ_0 accounts for less than 2%. Less than 20% of the time is used for updating J[:]. To demonstrate the performance of the incremental update, we compare it with rebuilding J[:] from scratch, and show the results in Fig. 3. The incremental update is about 10X better than rebuilding from scratch.

Finally, we evaluate the actual rule update latency and throughput on the switch. The results are shown in Fig. 4. GJ demonstrates the shortest update latency and the highest update throughput among all the algorithms. It turns out that the computation cost is more critical in determining the actual system performance, which surprisingly makes SC preferable to RC and HC in reality. The pursuit of better moving schemes must strive to balance the computation cost.

IV. Conclusion

GreedyJump achieves fast computation and efficient update scheme at the same time, both with the best-in-class performance. The evaluation qualifies *GreedyJump* as the default TCAM update algorithm ready for meeting the scalability challenges in current and future application scenarios.

References

[1] B. Salisbury. *TCAMs and Openflow-What Every SDN Practitioner Must Know*. Accessed: Jul. 2012. [Online]. Available: http://www.sdncentral.com/technology/sdn-Openflowtcam-need-to-know/2012/07/

[2] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware Openflow switches," *Comput. Netw.*, vol. 136, pp. 22–36, May 2018.
[3] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Passive Active Meas. Conf. (PAM)*, 2015, pp. 347–359.
[4] Y. Wan *et al.*, "T-cache: Dependency-free ternary rule cache for policy-based forwarding," in *Proc. IEEE INFOCOM*, Jul. 2020, pp. 536–545.
[5] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast TCAM updates," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 217–230, Feb. 2018.
[6] S. Devavrat and G. Pankaj, "Fast incremental updates on ternary-CAMs for routing lookups and packet classification," in *Proc. Hot Interconnects*, Aug. 2000, pp. 145–153.
[7] X. Wen *et al.*, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE ICDCS*, Jun. 2016, pp. 179–188.
[8] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "Fast lookup is not enough: Towards efficient and scalable flow entry updates for TCAM-based OpenFlow switches," in *Proc. IEEE ICDCS*, 2018, pp. 918–928.
[9] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "Fastrule: Efficient flow entry updates for TCAM-based OpenFlow switches," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 484–498, Mar. 2019.
[10] G. Li, Y. R. Yang, F. Le, Y.-S. Lim, and J. Wang, "Update algebra: Toward continuous, non-blocking composition of network updates in SDN," in *Proc. IEEE INFOCOM*, Apr. 2019, pp. 1081–1089.
[11] H. Xu, Z. Yu, X.-Y. Li, L. Huang, C. Qian, and T. Jung, "Joint route selection and update scheduling for low-latency update in SDNs," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3073–3087, Aug. 2017.
[12] X. Jin *et al.*, "Dynamic scheduling of network updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, Oct. 2014.
[13] 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.
[14] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance SDN switches," in *Proc. ACM CoNEXT*, 2017, pp. 283–295.
[15] B. Zhao, R. Li, J. Zhao, and T. Wolf, "Efficient and consistent TCAM updates," in *Proc. IEEE INFOCOM*, Jul. 2020, pp. 1241–1250.
[16] *Edge-Core Wedge100BF Series Switches*. Accessed: Mar. 2021. [Online]. Available: https://www.edge-core.com/
[17] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.
[18] J. Matoušek, G. Antichi, A. Lučanskỳ, A. W. Moore, and J. Kořenek, "ClassBench-NG: Recasting classbench after a decade of network evolution," in *Proc. ACM/IEEE ANCS*, May 2017, pp. 204–216.