ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/371174201

FlowBench: A Flexible Flow Table Benchmark for Comprehensive Algorithm Evaluation

Conference Paper · May 2023

CITATIONS
0

READS
37

5 authors, including:

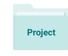
Haoyu Song
Futurewei Technologies
64 PUBLICATIONS **2,532** CITATIONS

SEE PROFILE

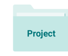
Bin Liu
Beijing Technology and Business University
346 PUBLICATIONS **6,510** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Dynamic Network Telemetry and Analytics View project



Low Latency Network View project

All content following this page was uploaded by Haoyu Song on 01 June 2023.

The user has requested enhancement of the downloaded file.

FlowBench: A Flexible Flow Table Benchmark for Comprehensive Algorithm Evaluation

Zhikang Chen*, Ying Wan*, Ting Zhang*, Haoyu Song†, Bin Liu*
*Tsinghua University, China †Futurewei Technologies, USA

***Abstract*—Flow table is a fundamental and critical component in network data plane. Numerous algorithms and architectures have been devised for efficient flow table construction, lookup, and update. The diversity of flow tables and the difficulty to acquire real data sets make it challenging to give a fair and confident evaluation to a design. In the past, researchers rely on ClassBench and its improvements to synthesize flow tables, which become inadequate for today's networks. In this paper, we present a new flow table benchmark tool, FlowBench. Based on a novel design methodology, FlowBench can generate large-scale flow tables with arbitrary combination of matching types and fields in a short time, and yet keep accurate characteristics to reveal the real performance of the algorithms under evaluation. The open-source tool facilitates researchers to evaluate both existing and future algorithms with unprecedented flexibility.**

I. Introduction

Flow table [1] is a fundamental and critical component in network data plane. Conventional routers and switches all have forwarding tables and Access Control List (ACL) as the core of the packet processing engine. In recent years, Software-Defined Networking (SDN) [2] abstracts the data plane as a flow table match-action pipeline. According to table size, matching type, system constraint, and performance requirement, different types of memory are engaged (*e.g.*, SRAM, DRAM, and TCAM [3]) and different algorithms are developed to address the potential performance bottleneck on table storage [4]–[9], lookup [10]–[16], and update [17]–[23]. While the increasing network scale keeps boosting the flow table sizes, SDN and the emerging autonomous networks significantly diversify the flow table types. For example, OpenFlow 1.0 [24] supports just 12 matching fields but OpenFlow 1.5 [25] already supports up to 45 fields. With the advent of protocol-independent data plane (*e.g.*, P4 [26]), a flow table can be customized from arbitrary packet headers and metadata combinations, even from protocols nonexistent today.

It is crucial to evaluate whether an algorithm can live up to its expectations or whether an old algorithm can adapt to a new situation. However, this is often a challenging task. The scales and characteristics of flow tables have different degrees of influence on an algorithm's performance. Therefore, a thorough evaluation requires a set of flow tables with varied scales and fine-tuned characteristics. Due to privacy and security concerns, it is difficult to acquire real flow tables from production networks, let alone enough such tables with ideal characteristics. As an alternative, researchers usually resort to software tools (*e.g.*, ClassBench [27]) to synthesize artificial flow tables. However, these tools are becoming inadequate to meet today's requirements for various reasons. Some are designed for conventional applications only (*e.g.*, IP lookup and 5-tuple IP packet classification), and some fail to provide accurate table characteristics.

This paper is supported by NSFC (61872213, 62032013, 62272258), NSFC-RGC (62061160489), and Guangdong Basic and Applied Basic Research Foundation under Grant 2019B1515120031.

An ideal flow table benchmark tool should meet the following requirements: 1) **Flexibility**. The tool should be able to generate flow tables with arbitrary sizes, number of fields, matching types, and other intrinsic characteristics. 2) **Accuracy**. All the characteristics of the generated flow tables should conform to the input parameters and be verifiable. 3) **Repeatability**. With the same set of input parameters, the flow tables generated from different platforms or at different time should be identical, allowing users to reproduce the experiments and make comparisons independently. 4) **Diversity**. Yet with different random seeds, different flow tables with the same characteristics can be generated, allowing users to take the average of multiple experiments and test the stability of their algorithms. 5) **Integrity**. Affiliated packet traces can be generated for each flow table to test the lookup performance. 6) **High Performance**. The tool should be reasonably fast, especially for large flow tables and packet traces.

Unfortunately, to the best of our knowledge, there is no such tool that can meet all these requirements. Therefore, we develop FlowBench, a new flow table benchmark, to fill the vacancy. FlowBench takes a different design principle. Instead of extracting the characteristics from some real-world "seed" flow tables and trying to retain them in the synthesized ones, we start from some intrinsic characteristics that have a direct impact on algorithm performance, and generate flow tables that can accurately match the configurations of the characteristics. Such a decision is due to the several drawbacks of the existing approaches: 1) it is unwarranted that the speculative large tables would still retain the same characteristics of the small seed tables, so the test results on such tables cannot guarantee an algorithm's real performance in reality; 2) the synthesized flow tables only vary in size and in a few preset classes, which are insufficient for thorough algorithm evaluation; 3) the synthesized tables can significantly deviate from the intended target characteristics, especially for large table size, and the synthesizing time increases exponentially when attempting to reduce the deviation; 4) the available seed data are only for conventional use cases (*e.g.*, IPv4 ACL and Firewall) and outdated, making the generated flow tables unsuitable for

today's and future use cases. In contrast, although the flow tables generated by FlowBench may not look "real", they can better reveal the true performance of an algorithm with different levels of fine-tuned stress tests, and still allow fair algorithm comparisons as long as the same benchmark is used.

The methodology of FlowBench is based on the Directed Acyclic Graph (DAG) abstraction. Each rule in the flow table corresponds to a node in the DAG. Therefore, the number of nodes in the DAG (*i.e.*, the DAG order) is equal to the flow table size, the edge between two nodes indicates the overlapping relationship of the two corresponding rules, the topological order of the nodes reflects the rule priority, the in and out-degree of a node signals the rule overlapping degree, and each directed path forms a rule dependency chain. Clearly, DAG grasps the essence of a flow table. The basic approach of FlowBench is to construct a DAG satisfying the given characteristics and derive a flow table from it. FlowBench can generate large flow tables in a very short time (*e.g.*, 10^6 rules in two seconds). Each rule supports arbitrary number of fields and each field can be of a matching type of Longest Prefix Match (LPM), Range Match (RM), or Exact Match (EM).

The rest of the paper is organized as follows: we first analyze the issues of existing benchmark tools (Section II), and then present the principle, functions (Section III), and algorithm (Section IV) of FlowBench. Next we evaluate FlowBench and demonstrate its usability through experiments (Section V). Finally, we review the related work (Section VI) and conclude the paper (Section VII). FlowBench is open source and available at *https://github.com/Flow-Bench/Flow-Bench*.

II. Motivation

ClassBench, proposed in 2007, is so far the *de facto* standard flow table benchmark tool. Limited by its seeds, ClassBench can only generate IPv4 5-tuple rules. In order to extend the support for IPv6 and OpenFlow, some improved versions of ClassBench, such as ClassBench-ng [28], are developed. These tools all follow the same design principle and basic approach. Taking ClassBench as the representative, it analyzes a few samples of real flow tables for ACL, Firewall (FW), and IP Chain (IPC), and extracts their characteristics as the seed for flow table generation. The seed contains a set of conditional probabilities derived from frequency statistics. Users can pick one of the 10 seeds provided by ClassBench to generate flow tables with different target sizes. By maintaining the statistical similarity between the generated tables and the seed tables, ClassBench claims that the generated flow tables can simulate the real-world situations. However, the aging ClassBench can no longer sustain its promise. Here are some details further backing up the assertion given in Section I.

First, ClassBench is inefficient in generating large flow tables and difficult to achieve the expected table size. Each rule is generated independently according to the conditional probabilities, so the rule duplication is unavoidable. Deduplication and redress are time-consuming. For example, when using IPC1 as the seed and setting $smoothness$ = 32, $address_scope$ = 0, $application_scope$ = 0, and expected table size n = 10^4, only 8,524 rules (85%) are generated. When n = 10^6, only 278,516 rules (28%) are generated. The ratio of actual to expected size decreases for larger sizes, smaller $smoothness$, or seeds from Firewall. The discrepancy between the expected n and the actual size is troublesome.

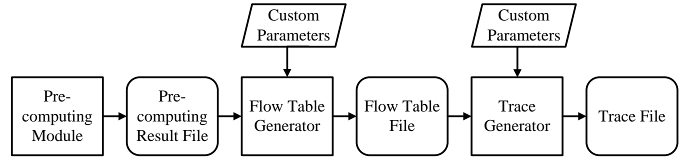


Fig. 1: FlowBench's workflow.

Second, the seeds collected in the 2000s can no longer reflect the current real network characteristics. New seeds need to be introduced to keep up, but such an effort increases the workload of researchers and the privacy issues may prohibit effective data sharing. Meanwhile, the rapid development of SDN and programmable networks make flow table rules customizable, volatile, and lacking a realistic statistics basis. As a result, ClassBench and such fall short of the needs of researchers today.

Third, ClassBench overlooks the fact that the performance of some algorithms is determined by the properties of the DAG corresponding to the flow table [17], [29]. Generating each rule independently, ClassBench has weak control over the rule dependencies. Although ClassBench provides some parameters such as $smoothness$, it is difficult to use them to control dependencies responsively (see Section V-D). Therefore, ClassBench cannot support straightforward performance evaluation under explicit and direct DAG characteristics.

In light of such observations, we design FlowBench. We focus on the flexibility rather than the authenticity of the artificial rules, as long as the algorithms can be faithfully evaluated and compared. According to the user needs, FlowBench complements the ClassBench-like tools and supports better controlled and more comprehensive evaluations for both existing and future algorithms.

III. FlowBench Overview

The workflow of FlowBench is shown in Fig. 1. FlowBench generates flow tables based on DAG. Some common graph-related computations are executed prior to any flow table generation. To improve the efficiency, we execute such computations once and save the result to a file as the data preparation step. The flow table generator reads the file and uses user-provided parameters to generate a flow table. After the flow table is saved in a file, the user can use FlowBench's trace generator to generate the corresponding packet trace.

As shown in Table I, FlowBench offers a rich set of parameters for accurate control of flow table characteristics. FlowBench supports the size of a flow table, n, to be as large as tens of millions, and it guarantees to generate exact n rules as specified by users in one run. Users can also specify the number of fields for a rule, and the bit width and match type

TABLE I: Parameters in Flow Table Generator

Parameter	Symbol	Type	Typical Value
Size of Flow Table	n	Integer	4096
Field Number	k	Integer	5
Field Bit Width	$b_{1\sim k}$	Integer	32,32,16,16,8
Field Match Type	$t_{1\sim k}$	LPM/RM/EM	LPM,LPM,RM,RM,EM
Field Weight	$w_{1\sim k}$	Double	32,32,16,16,8
Random Seed	r	Integer	-
Depth Parameter	D/D_r	Integer/Double	0∼1 (for D_r)
Edge Parameter	E/E_r	Integer/Double	0∼1 (for E_r)
Optional Flags	f	Flags	-

TABLE II: Parameters in Trace Generator

Parameter	Symbol	Type	Typical Value
Number of Packets	n	Integer	4096
Flow Table File	f_{rt}	String	256.txt
Rule Locality (Pareto Distribution)	a_r, b_r	Double	1,0
Flow Locality (Pareto Distribution)	a_f, b_f	Double	1,1

of each field. In addition, FlowBench allows users to assign a weight to each field. The greater the weight, the more likely FlowBench would establish dependencies on this field when generating rules. The default weight value is the same as the bit width of a field. A random value or the current time can be used as seed r to vary the resulting flow tables. FlowBench uses the Mersenne Twister Algorithm [30] to generate random numbers to ensure consistency across platforms.

FlowBench uses two parameters, D and E, to control the form of the DAG, where D controls the node depth and E controls the edge number. FlowBench allows users to specify the exact value of the edge number E. If users decline the direct control of E, FlowBench allows the use of a relative value E_r between 0 and 1 instead. E_r = 0 means E = 0, and E_r = 1 means the maximum value of E that FlowBench can generate. FlowBench converts the relative value to an exact value in subsequent calculations. In a DAG, the depth of a node is the maximum depth of its direct predecessors plus one, or 0 if it has no predecessor. The average node depth is an important DAG property. To be consistent with E, FlowBench accepts the parameter D as the sum of the depths of all nodes. Users can also provide a relative value D_r, similar to the concept of E_r. Because the average depth and the edge number are correlated, D and E or their corresponding relative values D_r and E_r are not allowed to be specified at the same time. The last parameter, f, is used to support the optional features of FlowBench, which will be described in detail in Section IV-F.

Table II shows the parameters used in FlowBench trace generator. Similar to ClassBench, we use the Pareto Distribution to describe the traffic locality. We consider two types of localities: the first type is for the flow level locality (*i.e.*, a small number of flows in the trace are elephant flows while the most flows are mice flows); the second type is for the rule level locality (*i.e.*, a small number of rules are hit by a majority of flows while the most rules are hit by a few flows). By default, the

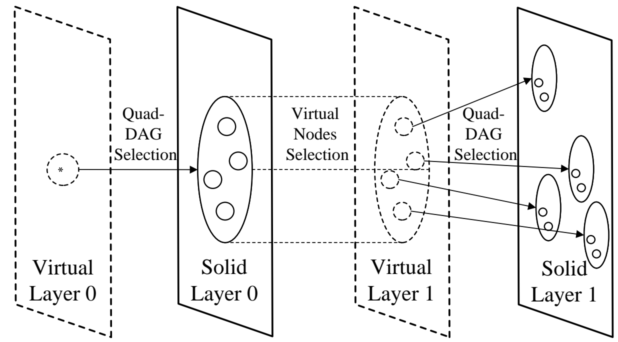


Fig. 2: A layer-based search example for $n = 12$.

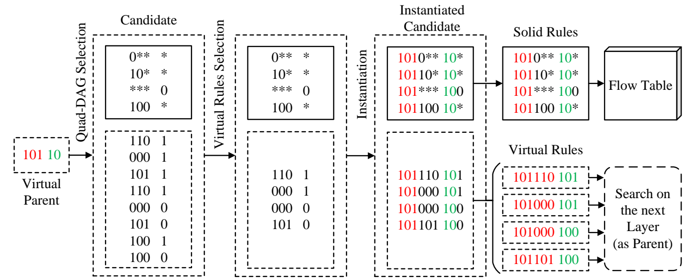


Fig. 3: Generating four solid and four virtual rules.

Pareto parameter b of rule locality is set to 0 to exclude the rule level locality, similar to ClassBench-like tools.

IV. Algorithm Design

FlowBench uses a layer-based search algorithm to generate a DAG as well as the corresponding flow table. As shown in Fig. 2, for efficient DAG control, we split the search process into interlaced solid layers and virtual layers. The solid layers produce solid rules which are output to the resulting flow table, while the virtual layers contain virtual rules as intermediate variables for inter-layer edge extension.

The search starts from the virtual layer 0, which contains only one virtual rule, the wildcard. We perform a solid rule selection to construct the solid layer 0. In this step, a sub-DAG of order 4 (referred to as Quad-DAG) is selected from a number of pre-computed candidates. The selected Quad-DAG determines the four solid rules in the solid layer 0. On the other hand, among the pre-computed candidates, each Quad-DAG has at least five corresponding candidate virtual rules (described in detail in Section IV-A). According to our design, while the virtual rules may depend on 0∼4 solid rules in the Quad-DAG, they are independent of each other, so they appear to be isolated nodes in the DAG. Four of them are chosen to construct the virtual layer 1.

The above process includes two steps: Quad-DAG selection and virtual-node selection. We take the wildcard rule on the virtual layer 0 as the parent, and generate four solid and four virtual rules in the two subsequent layers. An example of the procedure for generating two-field rules is shown in Fig. 3. An instantiation process is used to adjust the matching range of each solid or virtual rule to be a subset of the parent, under

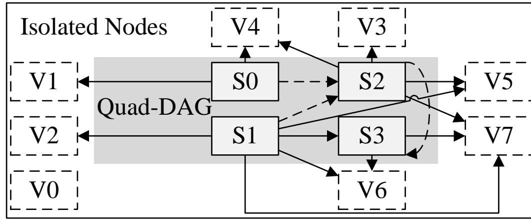


Fig. 4: A Quad-DAG example.

TABLE III: The pre-computed data of the example

Field Number		2			
		Field1	Field2	D_j or d	E_j or e
Maximum Prefix Length		3	1		
Solid Rules	S0	0**	*	0	0
	S1	10*	*	0	0
	S2	***	0	1	2
	S3	100	*	3	4
Virtual Rules	V0	110	1	0	0
	V1	000	1	1	1
	V2	101	1	1	1
	V3	110	0	2	1
	V4	000	0	2	2
	V5	101	0	2	2
	V6	100	1	3	2
	V7	100	0	3	3

the premise that the dependencies are unchanged. In Fig. 3 we simply concatenate the prefixes together, but the actual process is more complicated (see Section IV-E). After the instantiation, the solid rules are output to the flow table, and the virtual rules become the new parents for searching on the next layer. The remaining $n-4$ rules are divided into four equal parts and each part is to be generated by extending one of the four virtual rules in the recursive process. Fig. 2 shows an example for n = 12, in which each node on the virtual layer 1 is assigned to generate two solid rules. Therefore, we only need to pick two of the four solid rules in each Quad-DAG on the solid layer 1 without needing to construct the virtual layer 2.

In the rest of this section, we first discuss the information included in the pre-computed candidates (Section IV-A) and the maximum D and E that FlowBench supports under a given n (Section IV-B). Then we give the methods of Quad-DAG selection (Section IV-C) and isolate-node selection (Section IV-D), and prove that the user parameters can be satisfied. Next, we show how candidates are instantiated to enhance rule diversity and guarantee that their matching range is a subset of the parent (Section IV-E). Finally, we present the optional features supported by FlowBench (Section IV-F), and briefly describe the design of the trace generator (Section IV-G).

A. Compute Quad-DAG Candidates

The independent isolated nodes on a virtual layer (corresponding to virtual rules) lead to the independence of Quad-DAGs with different parents. Intra-layer node dependency exists only within a certain Quad-DAG and can be controlled in Quad-DAG selection.

Obviously, the larger the sub-DAG, the richer the diversity. However, as the sub-DAG order increases, the number of sub-DAG patterns increases rapidly. Let the order of a sub-DAG be s. Considering that any sub-DAG of an order-s directed complete graph may correspond to a candidate, we need to perform the computation for at least $2^{\frac{s(s-1)}{2}}$ cases. The number is 64 when s = 4, but becomes 32,768 when s = 6. Since we need to use a depth-first search algorithm with high time complexity to find the candidates, we choose only Quad-DAGs (*i.e.*, s = 4) as the candidates for the trade-off.

There are a total of 64 order-4 DAGs. On this basis, we distinguish the edges into two categories for diversity: an edge is categorized as *complete* if the matching range of one rule completely covers the other, or categorized as *partial* if the rules are only partially overlapping. Excluding some equivalent or unconstructable cases, 199 Quad-DAG candidates are left. Thanks to the pruning technology, the entire pre-computing process can be completed in a few seconds.

These candidates are saved in an output file for subsequent use. Fig. 4 shows one of the Quad-DAGs as an example, in which *complete* edges are represented by solid lines and *partial* edges are represented by dashed lines. In the output file, the profile of a candidate Quad-DAG covers the following aspects, as shown in Table III:

1) A set of four solid rules. These rules contain no more than four LPM fields, each using a bit width of no more than 4.
2) The field number k, and the maximum prefix length on each field.
3) The sum of the depths of the first j solid rules, D_j, for j = 1,2,3,4. This property is used to make the selected Quad-DAG satisfy the parameter D. Considering that the bottom solid layer may only select the first j rules in a Quad-DAG, each case of j must be considered here.
4) The number of edges in the sub-DAG containing the first j solid rules, E_j. Similarly, this property is used to make the selected Quad-DAG satisfy the parameter E.
5) A set of at least five virtual rules. The virtual rules use the same k fields as the solid rules, and the prefix length on each field does not exceed the maximum value. In order to control D and E in the subsequent recursion, each virtual rule also contains two properties: d and e. If a virtual rule is given the highest priority and added to the Quad-DAG, its depth is d and its degree is e.

The algorithm guarantees that for each solid rule r_s, there exists a corresponding virtual rule r_v, whose matching range is a subset of r_s and does not overlap with the range of the solid rules with a higher priority than r_s. This property makes sure every generated solid rule can be hit by some packet. Meanwhile, it guarantees that there exists a virtual rule r_{v0} whose range does not overlap with any solid ranges. In other words, we make it possible that some packets cannot hit any solid rules. Therefore, there are at least five virtual rules for each Quad-DAG. The extra virtual rules come from the overlapping ranges of the multiple solid rules.

B. Compute Maximum D or E

FlowBench supports a wide range of values for D and E (see Section V-D), but there exist the theoretical limits. In case a user gives a D or E beyond the limit, FlowBench reports an error. We denote the maximum parameters under a given n as $M_D(n)$ and $M_E(n)$.

First we discuss the case of $M_E(n)$. When $n{\le}4$, all rules are included in one Quad-DAG, so the case that maximizes E is an order-n complete graph, indicating $M_E(n){=}(n{-}1)!$. When $n{>}4$, we need to select a Quad-DAG first. For maximization, the order-4 complete graph is selected, which has four solid rules and six edges. The remaining $n{-}4$ rules should also depend on every rule in the Quad-DAG, resulting in $4(n-4)$ edges. Then we allocate the remaining $n-4$ rules to R_j ($j = 1,2,3,4; \sum_{j=1}^{4} R_j = n-4$) and assign them to four virtual nodes. The maximum in each part is $M_E(R_j)$. In summary, we get the following recursive formula:

$$M_E(n) = \begin{cases} (n-1)!, & n \le 4 \\ 4n - 10 + \sum_{j=1}^{4} M_E(R_j), & n > 4 \end{cases} \tag{1}$$

There is a significant correlation between the depth and in-degree of nodes. In fact, it can be obtained that $M_D(n)$ has exactly the same recursive formula as $M_E(n)$ by a derivation similar to the above. Therefore, we denote the maximum value for both parameter in a uniform form $M_P(n)$.

C. Quad-DAG Selection

Taking a certain virtual node as the parent, we need to select a Quad-DAG from the 199 candidates. Three parameters play a role in the selection process:

1) The number of rules to be generated, denoted as n.
2) The number of available fields, denoted as k_a. If the parent has used up all the bits on a field for an exact value, the field becomes unavailable. EM fields are always unavailable in any case.
3) The given parameter P. Due to the correlation between D and E, the same symbol is used for either one.

Only Quad-DAGs with no more than k_a fields are potential candidates in the selection. On this basis, the selection result depends on n and P. When $n \le 4$, we only need to select a Quad-DAG whose first n rule(s) satisfy the parameter P. When $n > 4$, we denote the parameter (D or E) of the Quad-DAG itself as P_1, which is between 0 and 6, and denote the parameter (d or e) of the j-th selected virtual rule as $P_{2,j}$, which is between 0 and 4. The maximum of $P_{2,j}$ for $j = 1,2,3,4$ is denoted as $P_{2,max}$. After selecting a Quad-DAG, the remaining $n{-}4$ rules will be divided into four parts of size R_j, and their corresponding parameters are denoted as $P(R_j)$. So we get a formula:

$$P = P_1 + \sum_{j=1}^{4} \left[P_{2,j} R_j + P(R_j) \right] \tag{2}$$

where $P(R_j)$ and $P_{2,j}$ satisfy

$$0 \le P(R_j) \le M_P(R_j) \tag{3}$$

$$0 \le P_{2,j} \le P_{2,max} \tag{4}$$

According to Equation 2, 3, and 4, we get three constraints that P_1 and $P_{2,max}$ must satisfy:

$$P_1 \le P \tag{5}$$

$$P_1 \ge P - 4(n-4) - \sum_{j=1}^{4} M_P(R_j) \tag{6}$$

$$P_{2,max} \ge \frac{1}{n-4} \left[P - P_1 - \sum_{j=1}^{4} M_P(R_j) \right] \tag{7}$$

Equation 5, 6, and 7 limit the range of candidates. Then we give each P_1 a weight. Candidates with the same P_1 divide the weight equally. In order to make the parameter of Quad-DAGs on different layers relatively uniform, we make $\alpha_1 = 6 \cdot \frac{P}{M_P(n)}$ and take $\mathcal{N}_1(\alpha_1 - P_1)$ as the weight of P_1, where $\mathcal{N}_1$ is the probability density function of a normal distribution with $\mu{=}0$. A larger standard deviation of $\mathcal{N}_1$ means worse uniformity and better diversity, and vice versa. The typical value of σ is 3.

D. Isolated-node Selection

When $n{>}4$, we need to select four virtual rules to prepare for searching on the next layer. These rules are selected sequentially and independently. When selecting the j-th virtual rule, according to Equation 2, 3, and 4, we get two constraints that $P_{2,j}$ must satisfy:

$$P_{2,j} \le \frac{P_{r,j}}{R_k} \tag{8}$$

$$P_{2,j} \ge \frac{1}{R_k} \left[P_{r,j} - \sum_{k>j} P_{2,max} R_k - \sum_{k=1}^{4} M_P(R_k) \right] \tag{9}$$

where

$$P_{r,j} = P - P_1 - \sum_{k=1}^{j} P_{2,k} R_k \tag{10}$$

is the remaining parameter to be assigned in subsequent steps.

Similar to the Quad-DAG selection, we give a weight to each virtual rule which satisfies Equation 8 and 9. We make $\alpha_2{=}4 \cdot \frac{P}{M_P(n)}$ and take $\mathcal{N}_2(\alpha_2 - P_2)$ as the weight of a virtual rule with P_2, where $\mathcal{N}_2$ is the probability density function of another normal distribution with $\mu{=}0$ and $\sigma{=}2$.

Because the virtual rules are selected independently, we may select a certain rule r_v more than once. In this case, we randomly choose one field in r_v and extend its prefix. For example, if a one-field rule r_v is selected twice and the prefix is 010*, we extend it by one extra bit to 0100* and 0101*. If it is selected three or four times, two extra bits are required. After such extension, we guarantee that the four virtual rules are distinct and independent of each other.

E. Instantiation

We complete the instantiation work in three steps to finalize the selected results into tangible solid and virtual rules.

1) Bit Instantiation: As shown in Table III, the pre-computed result comes from a depth-first search in which 0 is searched before 1 for each bit, resulting in imbalance and limiting the diversity. To solve this problem, we perform a bit instantiation on the solid and virtual rules selected. For each of the k fields, we generate a random mask and use it to XOR the field in every rule. This is equivalent to performing multiple symmetric transformations on a set of hypercubes in the matching space, which does not change the dependencies between the rules.

2) Field Instantiation: FlowBench allows users to specify the number, weights, bit widths, and match types of the fields in a rule. Since the rules selected from the pre-computed results (abbreviated as pre-computed rules) contain no more than four fields, and they are all LPM fields, in order to meet the user specifications, we need to relocate the fields.

To guarantee the matching range of the instantiation result is a subset of the parent, we use the remaining bit width (RBW) of each field to describe its capability to accept a pre-computed LPM field. The exact definition of RBW depends on the match type.

1) For LPM fields, RBW = field bit width - prefix length of the parent.
2) For RM fields, RBW = $\lfloor \log R \rfloor$, where R is length of the parent's matching range.
3) For EM fields, RBW = 0 regardless of the parent.

For each of the k pre-computed rules, we randomly select one of the fields that has never been selected before, according to the user-provided weights. If its RBW is equal to or greater than the maximum prefix length in pre-computed rules, we relocate the pre-computed field to the selected field. Otherwise, we ignore the field and re-select, until we find a selected field with a sufficient RBW or there is no field to select from. The latter situation may occur when non-EM fields are too narrow and n is too large, which leads to an error report.

EM fields can never accept pre-computed fields because RBW = 0. After an EM field is selected according to the weights, we will generate a random value, and every solid or virtual rule will be assigned with the exact value on the field.

3) Rule Instantiation: Finally, we restrict the matching range of these solid and virtual rules to a subset of the parent. The embedding method depends on the field type.

1) For the LPM fields, we concatenate the prefixes and sum the prefix lengths.
2) For the RM fields, we linearly interpolate the endpoints from the entire matching space to the range of the parent.
3) For the EM fields, if the parent has an exact value, the value is inherited; otherwise no change is made.

After the above steps, we complete the whole instantiation process. The solid rules are added to the flow table as part of the final result, and the virtual rules are used as the parents for searching in the next layer if $n>4$.

F. Optional Features

In addition to the basic algorithm, FlowBench provides the support for some optional features. Users can decide whether or not to enable these functions.

1) Arbitrary Ranges (AR): Since the pre-computed fields are all of the LPM type, the RM field generated by the above algorithm appears as a range that can be represented by a prefix. To support arbitrary ranges like $[1, 16]$, we introduce an algorithm which can be executed after rule instantiation.

The algorithm applies a random offset to the left and right endpoints of every RM field in every solid or virtual rule independently. Suppose we want to apply an offset to the left endpoint of range $[l, r)$. We make $U = \lfloor \frac{r-l}{4} \rfloor$ as the initial value of the upper bound of the random offset. Then we generate a random offset Δ between $[-U, U]$ and consider the new range $[\max\{L, l + \Delta\}, r)$, where L is the left endpoint of the parent. The new range will replace the old one if no dependency would be affected; otherwise, we let $U = \lfloor \frac{U}{2} \rfloor$ and regenerate the offset. The initial value ensures the range is reduced by at most half of the original size, avoiding the rapid decline of RBW.

Since multiple iterations are needed, this algorithm may affect the performance of FlowBench. Given many flow table algorithms require converting a range to prefixes first (*e.g.*, for TCAM-based tables), we made this feature optional.

2) Countermeasures for insufficient bit width: In FlowBench each solid layer requires a bit width of 3 to 5 bits. Therefore, insufficient bits will make it difficult to generate a large-scale flow table. For example, a 32-bit destination IP can only hold 6 layers ($<10^4$ solid rules). FlowBench takes two measures to solve this problem. (1) An optional *dense mode* is provided, in which another pre-computed set of Quad-DAG candidates will replace the common one. The set can no longer guarantee that every solid rule can be hit by packets, but requires fewer bits to generate a large table. (2) It is not very efficient to divide the matching space into four parts at the cost of 3 to 5 bits on each layer. If the DAG is relatively sparse, we can divide the entire space into several equal parts first, and then apply the layer-based algorithm in each part. Equal division use far fewer bits, which enables FlowBench to generate up to 2^{32} rules on destination IP when $E = 0$.

3) Parameters larger than $M_P(n)$: Eq.1 indicates $M_P(n) = O(n \log n)$, but D and E can be larger. In fact, the upper bound reaches $O(n^2)$ for complete graphs. An order-n complete graph requires $n-1$ bits, so the theoretical bound is limited by the number of available bits.

FlowBench provides an optional feature to achieve parameters larger than $M_P(n)$. We first build a Trie, in which each internal node corresponds to a solid rule, and each external node acts as an initial parent to apply the layer-based algorithm. Through a simple dynamic programming (DP) method, we can calculate the upper bounds of D or E under the condition that the size of the Trie is n_T and the height is h_T. We select the Trie with the smallest h_T to save as many bits as possible. This approach allows FlowBench to generate a relatively larger range of parameters.

G. Design of Trace Generator

Apart from the flow table generator, FlowBench provides a packet trace generator to generate the corresponding traces for a given flow table, which are useful to evaluate some lookup algorithms. The trace generator is a separate program from the flow table generator, but needs to take the flow table characteristics into consideration. In addition, like the characteristics of real traffic, the generated traces can be made to exhibit traffic locality to test the performance of certain algorithms (*e.g.*, flow caches).

The ClassBench-like flow table generators cannot directly control the dependencies between rules, so some rules may never be hit because their ranges are fully covered by some other rules with higher priority. As a result, the ClassBench-like trace generators only consider the flow-level locality, but ignore the rule-level locality.

In contrast, by adding constraints to the solid rules in a Quad-DAG (see Section IV-A), FlowBench guarantees that every rule in the flow table can be hit by certain packets. Therefore, FlowBench's trace generator is able to present the locality at both the flow and rule level. According to the distribution $\mathrm{Pareto}(a_r, b_r)$, we first calculate how many flows hit each rule. Then we calculate how many packets are in each flow according to the distribution $\mathrm{Pareto}(a_f, b_f)$. Finally, we do a random shuffle and output the packet trace.

V. Evaluation

In this section, we evaluate the advantages of FlowBench from three aspects: flow table scale (Section V-B), custom field configuration (Section V-C), and DAG control (Section V-D). In addition, we demonstrate the validity (Section V-E) and advantage (Section V-F) of FlowBench.

A. Evaluation Setup

We implement FlowBench using 6,417 lines of C++ code and compare it with ClassBench (since the ClassBench-like tools all use a similar approach, we select the classic ClassBench as their representative). FlowBench and ClassBench are hereinafter referred to as FB and CB, respectively. The configurations are summarized in Table IV.

TABLE IV: Configuration of the experiment platform

Item	Configuration
Mainboard	Supermicro X11DPG-OT-CPU
CPU	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz
RAM	DDR4 128 GB (2666 MT/s)
OS	Ubuntu 18.04.5 LTS
Core version	4.15.0-169-generic
G++ version	7.3.0
Compile options	-std=c++17 -O2 -fconcepts-ts

The actual size of the flow table generated by CB is smaller than the number specified by user. For comparison, we first use CB to generate a flow table, and take the actual size of it as n; then we use FlowBench to generate a flow table of the similar 5-tuple rules with the same size n. We modify the source code of CB to make its random seed the same as ours

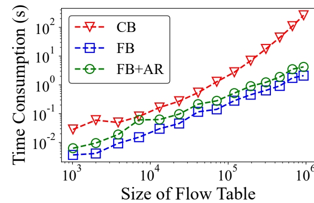


Fig. 5: Time consumption to generate a flow table.

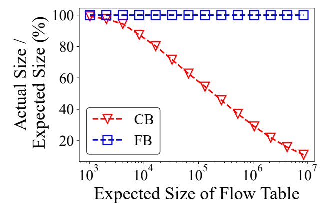


Fig. 6: Comparison of the expected and actual table size.

(note that by default the random seed of CB is time-variant, making it inconvenient for repeating experiments).

If not specified otherwise, we set D_r=0.1 in FB, and set $smoothness$=32, $address_scope$=0, $application_scope$=0, and $seed$= IPC1 in CB.

B. Evaluation on Flow Table Scale

First, we compare the time consumption of FB and CB to generate the same flow table size, and show the results in Fig. 5. Since the time complexity of FB is $O(n)$, there is an approximately linear relationship between the time consumption and n.

Although CB's time complexity is also $O(n)$, where n represents the expected table size provided by user, the number of distinct rules generated by CB, n', is smaller than n. The ratio of n'/n shrinks as n grows, as demonstrated in Fig. 6. This problem causes a 380x time consumption gap between CB and FB when generating a table containing 10^6 rules. If the feature of Arbitrary Ranges (AR) is enabled, FB's time consumption will increase by 2-3 times, but still far better than CB. These results show that FlowBench has a significant advantage over ClassBench in generating large-scale flow tables, and can always generate a table with the expected size.

C. Evaluation on Custom Field Configuration

This section demonstrates the flexibility of FlowBench in terms of rule fields. We first compare the performance of FlowBench under four flow table types:

1) IPv4 5-tuple.
2) OpenFlow 1.0 with 12 fields.
3) A protocol with 12 32-bit LPM fields (Custom-12).
4) A protocol with 24 32-bit LPM fields (Custom-24).

As shown in Fig. 7, the curves for different protocols are very close, indicating the time consumption of FB for generating a flow table is approximately proportional to the field number, but less affected by the bit width and match type.

Next we verify the effectiveness of the field weight parameter using IPv4 5-tuple rules (*i.e.*, SIP, DIP, SP, DP, and protocol). Fixing the sum of weights to 1, we set the weights of SP, DP, and protocol to 0, and adjust the weights of SIP and DIP. When the weight of SIP changes from 0 to 1, the weight of DIP changes from 1 to 0. We set $n = 1,024$, and get multiple sets of data by different random seeds. Fig. 8 shows

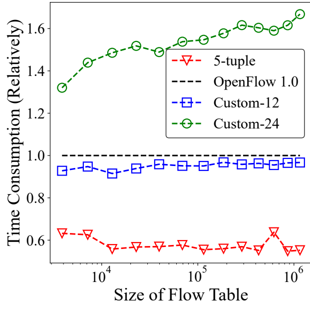


Fig. 7: Time consumption for different protocols.

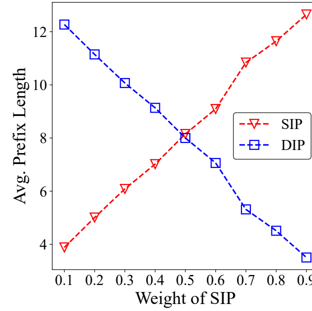


Fig. 8: Effect of weights on average prefix length.

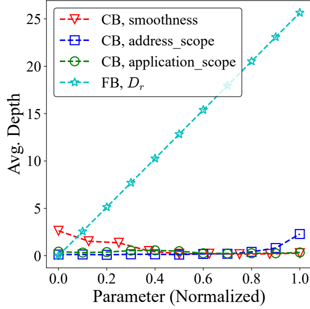


Fig. 9: Effect of parameters on average depth.

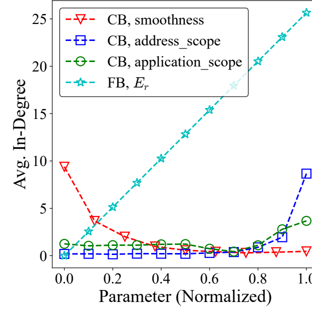


Fig. 10: Effect of parameters on edge number.

that the effect of weights on the average prefix length is also approximately linear.

D. Evaluation on DAG Control

This section shows that the parameters of FB can effectively control the DAG properties, and D and E can be selected in a large range. We also observe the effect of $smoothness$, $address_scope$, and $application_scope$ of CB on the properties of the DAG. For effective comparison, we normalize all the parameters to the interval of $[0, 1]$. The flow table size is set to n=$1,024$. Since CB cannot accurately generate n rules, we use the average depth and average in-degree of nodes for comparison.

As shown in Fig. 9, the parameters in CB cannot control the depth effectively. The DAG generated by CB has a relatively lower average depth, while FB can accurately generate a large range. For the number of edges, as demonstrated in Fig. 10, CB also shows poor controllability. Taking the $address_scope$ parameter as an example, in the range of $[-1, 0.6]$ (*i.e.*, $[0, 0.8]$ in Fig. 10), it has little effect on the number of edges and has poor sensitivity, while in the range of $[0.8, 1]$ (*i.e.*, $[0.9, 1]$ in Fig. 10) it is too sensitive to control. The line of FB remains straight because its parameters directly control the DAG, allowing users to get exactly what they want.

Fig. 10 also shows that for CB, the average in-degree is negatively correlated with $smoothness$, and positively correlated with $address_scope$. Therefore, we can estimate the range of average in-degrees by letting these parameters take the endpoint values. Fig. 11 shows that different seeds result in different in-degree ranges for CB, but the ranges are all covered by the range of FB.

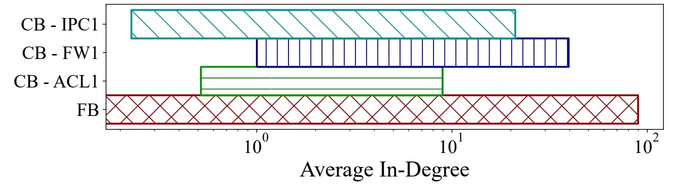


Fig. 11: Range of average in-degree.

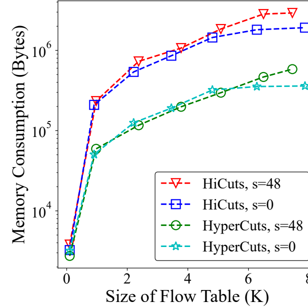


Fig. 12: Space evaluation of Hicuts&HyperCuts with CB.

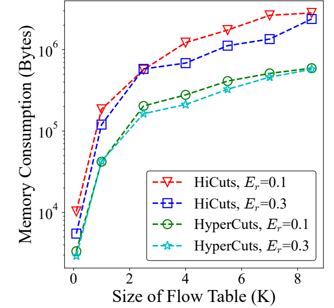


Fig. 13: Space evaluation of Hicuts&HyperCuts with FB.

E. Validity for Algorithm Evaluation

In this section, we take the classic packet classification algorithms HiCuts [31] and HyperCuts [32] as examples to demonstrate the validity of FB. For comparison, we first observe the results of evaluation using CB. We use ACL1 as the seed, set $address_scope$ and $application_scope$ to 0, and observe the performance of these two decision-tree algorithms with the configurations of $smoothness$=48 or $smoothness$=0 ($smoothness$ is denoted as s for short in Fig. 12 and Fig. 14), representing sparse and dense cases, respectively.

We evaluate the algorithms on both space and lookup time. We measure the memory consumption as the bytes consumed by the decision tree. On the other hand, we use the trace generator provided by CB to generate trace, and measure the throughput of the algorithm with the memory bandwidth consumption. We observe the average number of dependent memory accesses per packet lookup [33]. The number of packets in the trace is 16 times the size of the flow table, and has a high flow-level locality as $\text{Pareto}(1, 1)$. The evaluation results are shown in Fig. 12 and Fig. 14.

In FB we set $E_r = 0.1$ and $E_r = 0.3$ to represent the sparse and dense cases, respectively. FB's trace generator is also configured with the same settings to ensure meaningful comparisons. The corresponding evaluation results are shown in Fig. 13 and Fig. 15.

By comparing Fig.12∼15, we can see that FB shows similar trends to CB and can draw consistent conclusions. For memory consumption, HiCuts is much higher than HyperCuts, and

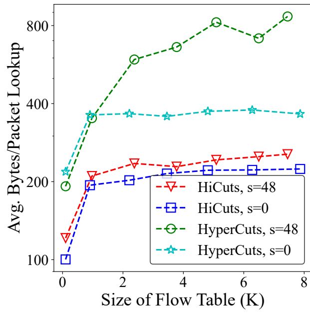


Fig. 14: Time evaluation of Hicuts&HyperCuts with CB.

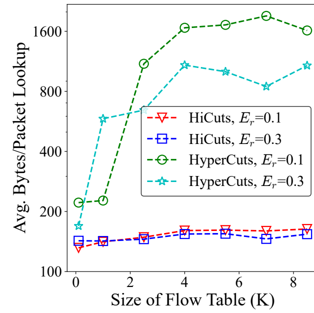


Fig. 15: Time evaluation of Hicuts&HyperCuts with FB.

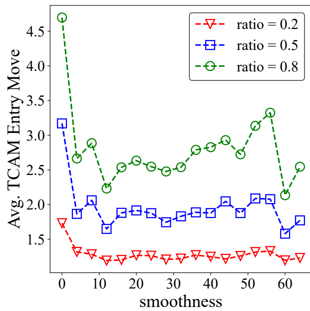


Fig. 16: Performance Evaluation of CAO_OPT with CB.

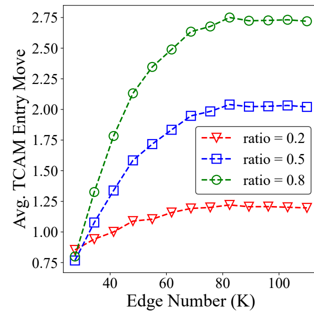


Fig. 17: Performance Evaluation of CAO_OPT with FB.

the density of DAG has no significant impact on memory consumption for both algorithms. On the other hand, HiCuts requires fewer memory accesses per lookup and therefore has higher throughput. Furthermore, the density of the DAG has little influence on the throughput of HiCuts, but has a significant effect on HyperCuts. A sparse DAG implies a flat decision tree, which lowers the performance advantage of HyperCuts.

F. Advantage in Algorithm Evaluation

We take the TCAM update algorithm CAO_OPT [34] as an example to show the advantage of FB. We generate a flow table of n=4, 096 rules, from which we randomly select $n * ratio$ rules as initial entries of TCAM, and other rules as updates.

As demonstrated in Fig.17, the performance of CAO_OPT, *i.e.*, the average number of TCAM entry moves for each update, is related to the number of edges in the DAG. It is more sensitive when the DAG is relatively sparse, and the effect is not significant when the DAG is denser. In contrast, such an observation is difficult to be derived from the evaluation result using CB as shown in Fig. 16, due to the inability of CB for precise DAG control.

This example shows the unique advantage of FB in the evaluation of TCAM-based flow table algorithms. Thanks to the greater flexibility of FB, researchers are empowered to gain more insights in understanding an algorithm.

VI. Related Work

According to the number of fields in the generated rule table, existing tools can be classified into single-field rule table generator [35], [36] and multi-field rule table generator [27], [28], [37], [38]. The former methods, such as NRG [35] and V6Gene [36], are used to generate IPv6 forwarding tables in the early stage of IPv6 networks.

Multi-field rule tables are usually applied for Firewall, QoS, traffic management, and SDN flow tables [28]. The approaches widely used at present are derived from ClassBench proposed in 2007. These tools all use the data generation method based on conditional probability. ClassBenchv6 [37] can generate IPv6 5-tuple rule table, which adopts the idea of mapping from IPv4 rule table in the same way that NRG generates IPv6 prefixes. Based on ClassBench, Matoušek *et al.* developed a rule table generation tool ClassBench-ng [28] for IPv4, IPv6, and OpenFlow 1.0. ClassBench-ng improves the rule generation process to make the rule table more consistent with the statistical characteristics of the real flow tables. Specifically, assuming that the number of rules to be generated is n, ClassBench-ng first calls ClassBench to generate a larger rule table (*e.g.*, $100n$), and then iteratively selects the rules that obey the parameters on the Trie constructed by SIP and DIP.

In summary, the conditional probability-based methods extract some statistical features (*i.e.*, conditional probability) from real data set, and make customized corrections to these statistical features according to the parameters given by users, thereby generating flow tables resembling real flow tables.

These methods aim to generate rule sets that approximate to the seeds on some selected statistics and wish such approximation can reflect the reality. However, such an assumption is unproven and some important but non-statistical features, *e.g.*, rule dependencies, are ignored. Moreover, limited by the small set of seeds, they can only generate limited types of flow tables and are difficult to expand in scale, failing to meet the diverse needs of the current and future applications.

VII. Conclusion

The open-source FlowBench can generate large-scale flow tables containing millions of rules in seconds. The number, type, and size of rule fields can be customized by users. Without relying on real-world table samples, FlowBench recursively constructs a DAG-based on user-specified characteristics and uses it to synthesize a more flexible flow table. It provides accurate control over the average depth or edge number of the DAG, and can generate feature-rich flow tables to meet the algorithm evaluation requirements.

We expect FlowBench to become a universal tool for evaluating flow table related algorithms (*e.g.*, data structure, lookup, and update). We open source FlowBench. More detailed information and the source code can be found at *https://github.com/Flow-Bench/Flow-Bench*. We invite the research community to use it, and provide feedback and contributions to improve FlowBench. In our future work, we will use FlowBench to evaluate more existing flow table related algorithms and document the findings.

REFERENCES

[1] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," *International Conference on Passive and Active Network Measurement*, pp. 347–359, 2015.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] P. Saggurti, "An Introduction to TCAMs," 2021. [Online]. Available: https://www.synopsys.com/designware-ip/technical-bulletin/introduction-to-tcam.html

[4] A. X. Liu, C. R. Meiners, and E. Torng, "Tcam razor: A systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Transactions on Networking*, vol. 18, no. 2, pp. 490–500, 2009.

[5] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *IEEE Transactions on Computers (TOC)*, vol. 62, no. 6, pp. 1127–1140, 2012.

[6] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," *Proceedings of the Symposium on SDN Research*, pp. 1–12, 2016.

[7] R. Li *et al.*, "Taming the wildcards: towards dependency-free rule caching with FreeCache," in *Proceedings of the 28th International Symposium on Quality of Service (IWQoS)*, 2020, pp. 1–10.

[8] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proceedings of the International conference on distributed computing and networking (ICDCN)*. Springer, 2013, pp. 439–444.

[9] H. Liu, "Routing table compaction in ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, 2002.

[10] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys (CSUR)*, vol. 37, no. 3, pp. 238–275, 2005.

[11] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *2014 IEEE 22nd International Conference on Network Protocols*, 2014, pp. 308–319.

[12] T. Yang, A. Liu, Y. Shen, Q. Fu, D. Li, and X. Li, "Fast openflow table lookup with fast update," 04 2018, pp. 2636–2644.

[13] J. Daly and E. Torng, "Bytecuts: Fast packet classification by interior bit extraction," 04 2018, pp. 2654–2662.

[14] Z. Liu, S. Sun, H. Zhu, J. Gao, and J. Li, "Bitcuts: A fast packet classification algorithm using bit-level cutting," *Computer Communications*, vol. 109, 05 2017.

[15] J. Daly and E. Torng, "Tuplemerge: Building online packet classifiers by omitting bits," 07 2017, pp. 1–10.

[16] W. Li, X. Li, H. Li, and G. Xie, "Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification," 04 2018, pp. 2645–2653.

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

[18] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "Fastrule: Efficient flow entry updates for tcam-based openflow switches," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 484–498, 2019.

[19] Y. Wan, H. Song, H. Che, Y. Xu, Y. Wang, C. Zhang, Z. Wang, T. Pan, H. Li, H. Jiang *et al.*, "FastUp: Fast TCAM Update for SDN Switches in Datacenter Networks," in *Proceedings of the 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 887–897.

[20] Y. Wan, H. Song, Y. Xu, C. Zhang, Y. Wang, and B. Liu, "Adaptive batch update in tcam: How collective optimization beats individual ones," *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, in press.

[21] H. Song and J. Turner, "Nxg05-2: fast filter updates for packet classification using TCAM," in *Proceedings of the 49th annual IEEE Global Telecommunications Conference (GLOBECOM)*, 2006, pp. 1–5.

[22] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "FastRule: efficient flow entry updates for TCAM-based openflow switches," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 37, no. 3, pp. 484–498, 2019.

[23] B. Zhao, R. Li, J. Zhao, and T. Wolf, "Efficient and consistent tcam updates," *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 1241–1250, 2020.

[24] Open Networking Foundation, "Openflow switch specification version 1.0.0," Tech. Rep., 2009.

[25] ——, "Openflow switch specification version 1.5.1," Tech. Rep., 2015.

[26] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

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

[28] J. Matoušek, G. Antichi, A. Lučanský, A. W. Moore, and J. Kořenek, "Classbench-ng: Recasting classbench after a decade of network evolution," in *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2017, pp. 204–216.

[29] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 179–188.

[30] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, p. 3–30, jan 1998. [Online]. Available: https://doi.org/10.1145/272991.272995

[31] P. Gupta and N. Mckeown, "Classifying packets with hierarchical intelligent cuttings," *Micro IEEE*, vol. 20, no. 1, pp. 34–41, 2000.

[32] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 4, 2003.

[33] H. Song and J. Turner, "Toward advocacy-free evaluation of packet classification algorithms," *IEEE Transactions on Computers*, vol. 60, no. 5, pp. 723–733, 2011.

[34] D. Shah and P. Gupta, "Fast incremental updates on ternary-cams for routing lookups and packet classification," 09 2000.

[35] M. Wang, S. Deering, T. Hain, and L. Dunn, "Non-random generator for ipv6 tables," 09 2004, pp. 35– 40.

[36] K. Zheng and B. Liu, "V6gene: a scalable ipv6 prefix generator for route lookup algorithm benchmark," in *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, vol. 1, 2006, pp. 6 pp.–152.

[37] Q. Sun, X. Huang, W. Yang, X. Zhou, Y. Ma, and C. Wang, "Classbenchv6: An ipv6 packet classification benchmark," in *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*, 2009, pp. 1–6.

[38] T. Ganegedara, W. Jiang, and V. Prasanna, "Frug: A benchmark for packet forwarding in future networks," 12 2010, pp. 231–238.

View publication stats