Baking the ruleset: A heat propagation relaxation to packet classification

Xinyi Zhang[*,§], Kave Salamatian[†], and Gaogang Xie[‡,§]

[*]Institute of Computing Technology, Chinese Academy of Sciences, China
[†]University of Savoie, France
[‡]Computer Network Information Center, Chinese Academy of Sciences, China
[§]University of Chinese Academy of Sciences, China
zhangxinyi@ict.ac.cn, kave.salamatian@univ-smb.fr, xie@cnic.cn

***Abstract*—As one of the most critical components in network appliances, the packet classification method has to deal with two frequently contradicting requirements: to classify the packets at line speed and to update the ruleset quickly. Tuple Space Search (TSS), a classical hash-based packet classification algorithm, achieves fast rule updating at the sacrifice of the packet classification rate. In TSS, each tuple is managed by a hash table and classifying a packet needs to go through all hash tables. Merging tuples can reduce the number of hash tables, but the improper merging scheme will increase the hash collisions that may even worsen the classification performance in some cases. In this paper, we propose a novel packet classification scheme to achieve fast packet classification and online rule update simultaneously. By using heat propagation to relax the tuple merging optimization problem, our method can reduce the number of hash tables while keeping the number of collisions low. Experimental results demonstrate that our method achieves $3.2\times$ classification speed and $4.6\times$ update speed on average compared with state-of-the-art algorithms.**

***Index Terms*—Routers, middleboxes, network functions and Software Defined Networking**

I. Introduction

Packet classification is one of the most critical operations in switches, routers, firewalls, load balancers and other network appliances. It is at the core of security [1], QoS [2], [3] and other advanced functions [4]. Packet classification aims at selecting the rule that matches the packet over multiple features, *e.g.*, source or destination IP addresses, TCP or UDP ports, *etc.*, or the context, *e.g.*, current state of a state machine, previous packet classification results, *etc.* And then the packets that match the same rule will be processed as a "flow". The packet classification can also be used in an elaborate intrusion detection state machines, gathering measurement statistics, *etc.* It is noteworthy that an incoming packet might match several rules, and in such case the highest priority rule, *e.g.*, the longest prefix match rule, should be selected.

Packet classification has to deal with two frequently contradicting requirements. On the one hand, packet classification needs to process packets at line speed in the data plane while ensuring a low processing delay. This requires integrating efficient, highly optimized, and sometimes very complex packet processing pipelines into the critical packet processing paths of network appliances. For these reasons, packet classification components have even been integrated into specific hardware, like ASIC, TCAM [5]–[7], FPGA [8] or GPU [9]. On the other hand, rulesets used for packet classification are not static and are updated dynamically. Such rule update affects the fast packet classification pipeline in the data plane, *e.g.* by having to stop the full processing pipeline or part of it, and buffering the incoming packets. With the emergence of Software Defined Networking (SDN) [10], Network Function Virtualization (NFV) [11], [12], high-performance cloud computing [13] and traffic engineering [14], the churn-rate, *i.e.*, the number of rules updated per second, has increased sharply. This makes the issue of rule update in packet classification more crucial. Therefore, current network architectures need a packet classification scheme that supports both fast classification and quick rule update.

As one of the primary network operations, packet classification has received enormous attention [15], [16]in the past 30 years. Existing schemes fall into three categories: hardware-based, dimensionality reduction and space partitioning. Each type has its strong points and weaknesses. In hardware-based method, while TCAM achieves high-speed packet classification performance thanks to its native parallelism offered by its hardware architecture. However, it suffers from high energy consumption and update cost [17]. In contrast, dimensionality reduction and space partitioning do not need specific hardware support and can be implemented in RAM. Dimensionality reduction approaches split the multi-dimensional ruleset into several single-dimensional ones to match individually. By making an intersection of the results from each subset, we can obtain the final result. Space partition methods fall into two main subcategories, decision tree approaches and hash-based approach. By splitting the ruleset into non-overlapping subsets, the decision tree approach uses the tree structure to direct finding subset that contains the matching rules. However, in the hash-based approach, each subset is managed by a hash table. To find the matching rule, we need to look up all the hash tables one after another.

ISBN 978-3-903176-28-7 © 2020 IFIP

The complexities of packet classification of these theoretical approaches have shown unavoidable trade-offs between classification speed, memory footprint and update complexity [18]. For example, linear search, where each incoming packet has to be matched one by one against all rules, has a memory footprint of $\mathcal{O}(KN)$, a time complexity of $\mathcal{O}(KN)$ and an update complexity of $\mathcal{O}(K)$, where N is the number of rules and K is the number of features used to define the packet classification rules. Space partition approaches using decision trees achieve a time complexity of $\Omega\left(\log N\right)^{K-1}$ and a space complexity of $\Omega\left(N\left(\log N\right)^{K-1}\right)$ [18]. Updates are known to cause a severe increase in complexity [19] as a single deletion may invalidate large portions of the tree data structure. The complexity of a mix of d queries and updates in a decision tree is known to be at worst $\Omega\left(d\left(\frac{\log d}{\log\log d}\right)^{K}\right)$ [18]. Indeed, these values are the worst case and we can expect that updates are less costly in practice. Hash-based approaches are between linear search and tree-based approaches. Linear search has low memory footprint but high classification delay. And the tree-based approaches have low classification delay but higher memory. Hash-based approaches have modest memory footprint and classification delay, and with the benefit of having the $\mathcal{O}(K)$ update cost.

Consequently, to find a hash-based method that supports both fast packet classification and quick rule update, this paper propose the Tuple Merge Relaxation (TMR) method. Compared with other state-of-the-art algorithms, TMR achieves $3.2\times$ classification speed and $4.6\times$ update speed. The remainder of this paper is organized as follows. We introduce the Tuple Space Search strategy for hash-based methods and the problems of Tuple Merge in §II. We then describe the relaxation of the tuple merge optimization problem by using the solution to the non-homogeneous heat propagation equation and propose our TMR method in §III. We evaluate the performance of TMR in §IV and review the related work in §V. Finally we conclude our work in §VI.

II. Tuple Space Search (TSS)

The ruleset used for packet classification consists of rules defined over K fields, $(F[1], F[2], \ldots, F[K])$. Each $F[i]$ generally comes from the packet header and represents a range, *e.g.*, $102.12.1.*$. An associated action in each rule will be applied to packets that match all the fields in the rule, *e.g.*, rejecting the packet in a firewall. Rules are generally ordered by a priority value that helps in choosing a single rule when a packet matches several rules. In Table I, the example ruleset contains eight rules with four fields each: source address (SA), destination address (DA), priority (Pri) and action (Act). For each incoming packet, we use SA and DA to search for the rule, and apply the action to the matched packet.

Hash-based methods generally follow the basic strategy of Tuple Space Search (TSS) [20]. In TSS, the ruleset is partitioned into several tuples, and the resulting set of tuples is called the tuple space. Each tuple T is identified by a K-vector $(T_1, \ldots, T_K)$, where K is the number of fields selected to classify packets and T_K is a prefix length for IP address or a range, *e.g.*, [1024,2048] for port number. Each tuple contains rules that have identical T_i over its i^{th} field $(i = 1, \ldots, K)$. So the example ruleset in Table I can be transformed into seven tuples, as shown in Table II. The tuple space can be represented as a K-dimensions rectangle with i^{th} side spanning from 0 to the number of bits in the i^{th} feature.

In TSS, each tuple is managed through a hash table that contains l entries. The *key* of the hash table is calculated over a bit string obtained by concatenating T_i first bits of field i. Consequently, the length of the key is at least $J = \lceil \log l \rceil$ and the hash table contains $2^J < 2l$ rules matching the key. When a hash collision happens, the corresponding bucket will contain more than a single rule and the rules are stored as a linked chain, *i.e.*, we implement a chaining hash table [21]. Therefore, classifying an incoming packet in TSS consists of extracting the corresponding prefix bits from the relevant field, concatenating them, calculating the hash value and checking if there is any rule in the resulting entry. If more than a single rule is stored in the hash entry, a linear search has to be launched to find the rule that matches the packet.

The complexity of calculating the hash value and finding the corresponding table entry is $\mathcal{O}(1)$. However, if there is more than a unique one in the entry, we should check all the rules in the bucket. Therefore, the performance of TSS also depends on the number of collisions in the buckets, which is related to both hash functions and the space overhead of hashing tables. The lookup complexity in a chained hash table of size N is known to be $\Theta\left(\frac{\log N}{\log\log N}\right)$ where we have $\Theta\left(N\right)$ element in the table [21]. The minimum number of bits needed to identify N different rules is $\log N$. The maximum number of possible tuples over $\log N$ bits is $\left(\log N\right)^K$. This means that the worst case time complexity of TSS is $\mathcal{O}\left(\left(\log N\right)^K\right)$ (assuming with perfect hashing). As there is no rule duplication in TSS, the memory space complexity is $\Omega(N)$. Updating a rule in TSS is straightforward. Removing a rule simply consists of finding the rule in the hash table and removing it. Rule insertion only needs hashing its key and inserting it into the corresponding hash table. The complexity of both operations is $\mathcal{O}(1)$. This means that the classification time complexity of TSS is slightly worse than that of the tree-based approaches whose footprint is relatively larger. However, TSS is still superior to it with a faster and constant update time complexity.

Nonetheless, the above complexities are the worst case and one can expect to stay away from these upper bounds in practice. In particular, to reduce the memory consumption of hash tables, we have to accept hash collisions. When a packet matches a hash entry with collisions, the complexity of rule matching due to a linear search will increase. Therefore, TSS involves a fundamental trade-off between memory footprint and additional delay resulting from collisions. However, the precise performance penalty from collisions depends on the statistics of the incoming traffic and the probability of a packet matching a hash entry that store several rules.

Several algorithms were proposed to improve the classifi-

Table I: A Sample Ruleset

Rule #	SA	DA	Pri	Act
0	101*	11010	0	Drop
1	101*	1001*	3	Fwd 0
2	11101	11100	3	Drop
3	010*	1000*	2	Fwd 1
4	00*	01001	2	Drop
5	11110	*	0	Drop
6	101*	1*	1	Fwd2
7	*	*	1	Drop

Table II: Tuples for the Sample Ruleset

Tuple #	Tuple	Member rules
0	(3,5)	0
1	(3,4)	1,3
2	(5,5)	2
3	(2,5)	4
4	(5,0)	5
5	(3,1)	6
6	(0,0)	7

cation speed of TSS. *Pruned Tuple Space Search* (PR-TSS) [20] adds tries over source and destination IP addresses to reduce the number of tuples that need to be checked. The fact that the longest prefix match of IP address always has higher priority introduces an iterative probing strategy. By shaving a $K-1$-dimensional slice in each step, such method reduces the volume of the K-dimensional rectangle of tuples that might contain the highest priority rule. This strategy reduces the worst case time complexity of TSS to $\Omega\left((\log N)^{K-1}\right)$ [20]. However, the information used for removing a slice can significantly increase the memory footprint. The complexity of precomputing such information is exponential in the number of tuples, as shown in [20]. A heuristic method is proposed that reduces the complexity of precomputing to $\mathcal{O}(W^3)$, where W is the number of initial tuples, which is still not enough to make the method tractable in practice. Moreover, the precomputed information needs to be updated when the ruleset is updated. As a result, the increase of the speed comes at the cost of a higher update complexity.

PartitionSort (PS) [22] mixes TSS and decision trees. Rather than partitioning rules based on tuples, rules are partitioned into sortable sub-rulesets that are stored through balanced search trees. By reducing hash calculations, PS can classify packets faster than TSS, but needs more time to generate and process the sortable sub-rulesets.

TupleMerge (TM) [23] leverages the idea of reducing the number of tuples by merging neighboring tuples and their corresponding hash tables into a single one. For example, the hash tables relative to tuples (1,1),(1,2),(2,1) and (2,2) can be merged into a single hash table. However, merging two hash tables might lead to "overlaps". Overlap happens when rules have the same prefix values by the reduction of the prefix length. For example in Table II, if we merge tuple 1 and 5, both rules 1 (101*,1101*) and 6 (101*, 1*) will be changed into(101*,1*), and the overlap happens. Overlaps have the same impact as hash collisions in TSS. As we have to check all rules in a hash table entry with a linear search, overlaps slow down the classification speed. If the hash tables are merged without care, a large number of resulting overlaps will significantly slow down the classification speed even though the number of hash tables is reduced. In consequence, to achieve high classification speed with the hash-based structure, it is necessary to minimize the number of overlaps when reducing the number of hash tables. In order to manage the overlaps, TM sets a threshold to split the hash table into two when it is exceeded. However, changing a single rule, in particular with a small prefix, can generate (or remove) a large number of overlaps. Such procedure will result in splitting or joining of hash tables with a very high update cost. Consequently, this paper aims to improve Tuple Merge by taking a different approach on merging tuples while controlling the number of overlaps.

III. Relaxation of the Tuples Merge optimization problem

The previous section introduces Tuple Merge and the issue of *overlaps* while merging tuples. In this section, we first formalize the Tuple Merge problem. Then we illustrate our TMR method over the case where the ruleset has only two matched dimensions: IPv4 source and destination addresses. After that, we extend it to more dimensions.

For each tuple $T = (T_1, \ldots, T_K)$, we can define a neighborhood that contains all tuples with prefix length ± 1 of prefix lengths in T, *e.g.*, tuples (1,2), (2,1), (1,3), (3,1) are neighbors of tuple (2,2). The concept of the neighborhood can be extended to a larger distance of prefix length. The merged tuples combine a "*Range Tuple*" (RT), a vector $R = (R_1, \ldots, R_K)$ where R_i represents a range of prefix-lengths that are stored in the RT. For example, (1–2,1–2) is a RT resulting from merging four tuples (1,1),(1,2),(2,1) and (2,2). Matching a packet with a RT R is done by extracting the prefix bits length defined by the lower bound of each range R_i. And in the above example, matching can be done by using one bit of each field. After that, the matching procedure in the corresponding hash table is similar to TSS.

For the 2-dimension ruleset, the tuple space can be represented by a 33x33 tuple matrix T, in which each cell T_{ij} represents the number of rules in the tuple (i, j). Any contiguous rectangular region of the matrix can be merged into a Range Tuple. However, merging tuples will result in overlaps. Therefore, we are in front of two contradicting factors. On the one hand, merging reduces the classification delay by reducing the number of hash tables to match. On the other hand, overlaps and collisions in the hash table increase the delay as we have to check all rules linearly in the matched entry. However, these two factors are not equivalent. Merging any tuple results in removing a hash calculation for all packets, while adding a collision (or overlap) increase the matching time only if an incoming packet matches the hash entry where the collision happens. Moreover, we measured

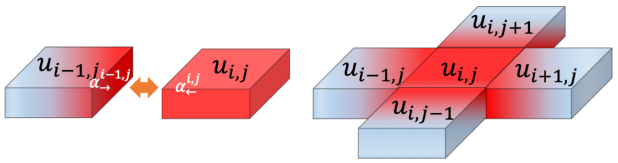


Figure 1: Heat Propagation

that calculating a hash takes on average 20 nano seconds while making a rule comparison entails 5 nano seconds. This means that the overlap is somewhat acceptable unless the number of it is large. However, it is not realistic to find an optimal balance between these two factors by doing an exhaustive search in all possible merging methods. One can easily see that if the number of possible Range Tuples in a matrix of dimension $n \times m$ is F(n,m), we have $F(n+1, m+1) = (2^m + 1)(2^{n+1} + 1)F(n, m)$, which results in an exponential blowup of the number of Range Tuples. As a result, we need a flexible heuristic method that optimizes the tuple merging process by balancing these two effects and achieving an overall excellent performance.

In this paper, we propose Tuple Merge Relaxation (TMR), a continuous approximation method of the initial discrete Tuple Merge optimization problem that is based on solving a non-homogeneous heat propagation problem. The neighbors of tuple(i, j) are $(i, j+1)$, $(i, j-1)$, $(i-1, j)$, and $(i+1, j)$. When we merge two neighboring tuples, the resulting hash table will have $T_{i,j} + T_{i+1,j}$ rules. We represent such merging in the matrix T by setting both cells (i, j) and $(i+1, j)$ to contain $\frac{T_{i,j}+T_{i+1,j}}{2}$ rules. We can easily extend this idea to cases where the merging involves a larger rectangle in matrix T, *i.e.*, after merging all the cells inside the rectangle, the number of rules in each cell is the average value of the total rules in the rectangle region.

With the above representation, the tuple merging procedure can be regarded as a diffusion process [24] inside the matrix T that diffuses the rules of tuples to their neighbors. However, different from the the traditional diffusion process, the "diffusion" in the ruleset is highly related to the number of resulting overlaps. This situation is very similar to an adiabatic heat propagation problem in two dimensions, where we have an initial heat energy distribution over a 2-D space. The heat energy is diffusing through conduction in the two dimensions and the heat resistance slows down the propagation speed. The analogy in our case is that we have heat energy equal to the value $T_{i,j}$ at each point cell (i, j), and heat resistance between any two cells is proportional to the number of overlaps when merging corresponding tuples.

In the following paper, we first give some background information relative to the heat propagation. Then we introduce our TMR method with a 2-dimensional ruleset. After that, how to construct hash tables according to the heat propagation result is illustrated. Finally, we extend our TMR method to K-dimensions.

A. Heat Propagation

The propagation of heat in a medium through conduction is governed by the Fourier Equation :

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{1}$$

where $u(x, t)$ is the temperature at point x and time t. This temperature is proportional to the density of heat energy when no heat energy is added or removed from the system. α represents the diffusivity of the medium, *i.e.*, larger (resp. smaller) values of α means that the medium is more (resp. less) conductive to heat. α might be constant (homogeneous case), or depends on x (non-homogeneous case), or even depends on the temperature (non-linear case).

In the homogeneous case (α is constant), the heat propagation equation illustrates that as long as the temperature is not homogeneous, *i.e.*, $\nabla^2 u \neq 0$, the temperature will change. In other terms, asymptotical temperature will become constant over all points and α only controls the speed of convergence from the initial condition to the stable asymptotic uniform temperature. However, for the non-homogeneous or the non-linear case, the temperature might vary at different points which depend on the diffusivity of the medium. An intuitive example of this is an electric kettle with a wooden handle, where the temperature of the wooden handle remains very low despite the metallic part of the kettle is very hot. In this case, $\alpha(x)$, controls both the shape of the asymptotic distribution of temperature and the speed of convergence to the stable state.

The heat propagation equation in homogeneous case has generally closed form solution. However, in non-homogeneous and non-linear case, one has to resort to the numerical methods for solving it. The approach used to solve the heat propagation problem in continuous space is mainly Finite Element methods [25]. In our application, the merging process plays the role of heat diffusion. If there is no overlap, all tuples will be merged into a single RT. However, the overlap would hinder the diffusion and play the role of the diffusivity barriers that make the heat propagation non-homogeneous. Consequently, as the tuples matrix T is discrete in its row and column, we are already in the discrete settings and can use a simpler approach to solve our problem.

The non-homogeneous heat propagation Eq. 1 can be discretized over a $N \times M$ rectangular grid of points using discrete Laplacian:

$$\frac{u_{i,j}(t+\Delta) - u_{i,j}(t)}{\Delta} = \frac{\alpha^{ij}_{\rightarrow} u_{i+1,j} + \alpha^{ij}_{\leftarrow} u_{i-1,j} + \alpha^{ij}_{\uparrow} u_{i,j+1} + \alpha^{ij}_{\downarrow} u_{i,j-1} - 4\bar{\alpha}^{ij} u_{i,j}}{\delta^2} \tag{2}$$

where $\alpha^{ij}_{\rightarrow}$, $\alpha^{ij}_{\leftarrow}$, $\alpha^{ij}_{\uparrow}$ and $\alpha^{ij}_{\downarrow}$ are non-homogeneous diffusivity coefficients in 4 cardinal directions, and $\bar{\alpha}^{ij} = \frac{\alpha^{ij}_{\rightarrow} + \alpha^{ij}_{\leftarrow} + \alpha^{ij}_{\uparrow} + \alpha^{ij}_{\downarrow}}{4}$ is the average diffusivity at point (i, j). Δ is a small time-step and δ a small space-step size. Moreover, we have $\forall i < N, \alpha^{ij}_{\rightarrow} = \alpha^{i+1,j}_{\leftarrow}$, $\forall i > 1, \alpha^{ij}_{\leftarrow} = \alpha^{i-1,j}_{\rightarrow}$,

$\forall j < M, \alpha_{\uparrow}^{ij} = \alpha_{\downarrow}^{i,j+1}$ and $\forall j > 1, \alpha_{\downarrow}^{ij} = \alpha_{\uparrow}^{i,j-1}$. We have to adapt the equation to the boundaries that will not have the four direction neighbors. For example, on the right-most column of the 2D area, the discrete heat propagation equation becomes:

$$\frac{u_{N,j}(t+\Delta) - u_{N,j}(t)}{\Delta} = \frac{\alpha_{\leftarrow}^{Nj} u_{N-1,j} + \alpha_{\uparrow}^{Nj} u_{N,j+1} + \alpha_{\downarrow}^{Nj} u_{N-1,j-1} - 3\bar{\alpha}^{Nj} u_{i,j}}{\delta^2} \quad (3)$$

with $\bar{\alpha}^{Nj} = \frac{\alpha_{\leftarrow}^{Nj} + \alpha_{\uparrow}^{Nj} + \alpha_{\downarrow}^{Nj}}{3}$.

This means that one can model the evolution of the heat propagation equation as linear time evolution equation from time $k\Delta$ to $(k+1)\Delta$ as

$$U^{k+1} = U^K + A^k U^k \quad (4)$$

where U^k is the vector with all N^2 temperatures points, and A is a $N^2 \times N^2$ matrix where each row and column have at most 5 non zero values. The non-zeros elements in A are in form of $\alpha^{ij} \frac{\Delta}{\delta^2}$ or $-\bar{\alpha}^{ij} \frac{\Delta}{\delta^2}$.

We are interested in the asymptotic values of U^{∞}. This value can be found by running iteratively the Eq. 4, till convergence happens, *i.e.*, $||U^{k+1} - U^k||^2 < \epsilon$. The timescale of convergence of the homogeneous heat propagation equation is calculated to be $\tau = \frac{1}{4\alpha}$ [26], *i.e.* convergence happens at the time scale of τ. We therefore set $\Delta = \frac{1}{10\bar{\tau}}$, where $\bar{\tau} = \frac{1}{4\bar{\alpha}}$ and $\bar{\alpha}$ is the average value of α^{ij} over the whole 2 dimension space.

B. Relaxation procedure

The core idea of TMR is to relax the complex integer-valued 2-dimensional Tuple Merge optimization problem into a simpler continuous problem. The relaxation procedure consists of two steps. In the first step, TMR transforms the ruleset into a discretized 2-dimensional space and solves it with a specific heat propagation equation whose parameters depend on the initial ruleset. In the second step, TMR quantifies the resulting asymptotic temperature distribution and constructs hash tables with a split and merging procedure. In the forthcoming, we will introduce the above two steps in detail.

1) Continuous time heat propagation equation: In the heat propagation equation, the temperature $u(x,t)$ is proportional to the density of heat energy at point x. If we define the total energy of a ruleset as the number of rules in it, the density of energy at any position (i,j) in the tuple space is proportional to the number of rules in the tuple (i,j). Therefore, we set the values in tuples matrix T to be this proportion and represent it with a heat map in Figure2(a).

Then we define the number of overlaps when merging tuple (i,j) and $(i+1,j)$ as $O_{\rightarrow}^{ij}$. Similarly, the number of overlaps when merging tuple $(i-1,j)$ and (i,j) is $O_{\leftarrow}^{ij}$. So do $O_{\uparrow}^{ij}$ and $O_{\downarrow}^{ij}$. Then we define the diffusivity parameter α^{ij} to be a decreasing but positive function $g(.) \geq 0$ of the corresponding O^{ij}, *e.g.*, $\alpha_{\rightarrow}^{ij} = 1 - e^{-O_{\rightarrow}^{ij}}$ or $\alpha_{\rightarrow}^{ij} = \frac{1}{1+O_{\rightarrow}^{ij}}$.

After submitting the above parameters into Eq. 4, the asymptotic solution of the heat propagation gives the results of a continuous diffusion approximation for the tuple merge problem. The result is shown in Figure2(b). The shape of the function $g(.)$ controls the relationship between overlap and diffusion, as it defines the value of diffusivity.

2) Split and Merging Process: After obtaining the asymptotic result of heat propagation that works on continuous space assumption, we need a procedure to partition the result into a set of non-overlapping rectangles that cover the whole space, and each rectangle represents a hash table. To achieve the above goal, TMR carries the split and merging process. The procedure sets a decision probability threshold as a parameter, *e.g.* 99%. In the split phase, we use the Quad-tree segmentation approach [27] to partition the tuple space into four regions, NW, NE, SW, SE. Before splitting, a statistical test of equality of means between the large region and each of its contained four regions is carried out. In order to ensure the statistical significance, we use t-student test when the number of points in each region is more than 20, while we use the Mann-Whitney U test [28] when it is below 20. If the hypothesis cannot be rejected for all of the sub-regions, we stop splitting the region. If the hypothesis can be rejected for even a single region, we split the region and do the same procedure on each one of the sub-regions recursively.

At the end of the split phase, we obtain a set of rectangular regions. The merging phase aims at reducing the number of regions by merging some of them that are homogeneous and neighboring with each other. For each region, we check all of its neighbors (at most four neighbors) and use the t-student test or Mann-Whitney U test to do a pairwise homogeneity test. If we cannot reject the null hypothesis of homogeneity, we merge the two regions. We continue this process iteratively until no more merge can be done. The result of the split and merging process is shown in Figure3. Finally, we obtain a set of rectangular regions (Range Tuples) which guides the construction of hash tables.

C. Construction of hash tables

After obtaining the Range Tuples, we need to build the hash tables concretely according to them. The classical rule of thumb for the hash table is to set the hash table size to be L^2 to store L entries in order to ensure the probability of collision to be very small. However, as merging tuples generates overlaps, we can accept some additional hashing collisions. Suppose that we have to insert L entries in the hash table, we hash the bit string defined by the Range Tuple into $\lceil \log L \rceil$ bits and set the size of the hash table to be $2^{\lceil \log L \rceil}$ *i.e.*, the size of the hash table is between L and $2L$. As we are using a multi-position cuckoo hashing with 2 hash functions [29] the load factor with negligible collision can go up to 0.93. The additional free positions in the hash table are used as positions for the future insertions resulting from updates. In the case where the initial load factor of the table is too high (>0.9), we can add one more bit and address elements in the hash table using $\lceil \log L \rceil + 1$ bits. Then the hash table will contain $2^{\lceil \log L \rceil + 1}$ entries. With such choice, we ensure a low collision probability and reserve the space for the future inserted rules. Updating a rule entails

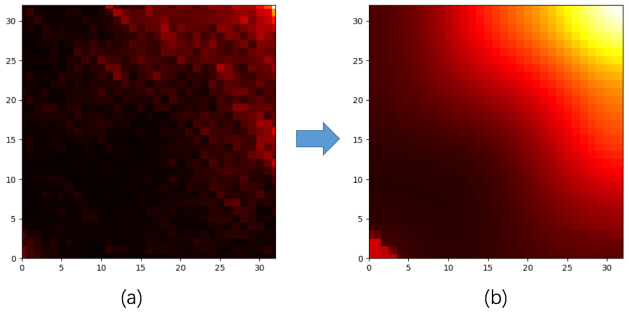


Figure 2: Continuous time heat propagation

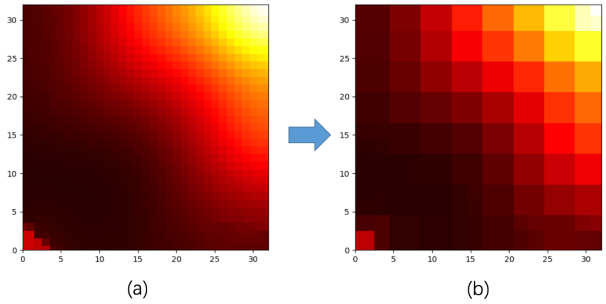


Figure 3: Splitting and mergering process

adding or removing it from the hash table, *i.e.*, an update can at most add or reduce one overlap. As stated before, for packets that will match the corresponding hash table entry, the cost of delay incurred by adding one additional comparison is only 5 nanosecond under our test environment.

D. Extension to K-dimensions

The above relaxation procedure is described in 2-dimensions. However, our TMR method can be extended to K-dimensions. In K-dimensions, the tuple space is projected into a discretized version of a K-dimensional space. The U matrix will have N^K elements and the A matrix is a $N^K \times N^K$ matrix. The stationary temperature distribution can be derived in the same manner as above by iteratively applying Eq. 4. The split and merging procedure has to adapt to K-dimensions. In the split phase, we divide each region by 2^K sub-regions. In the merge phase, each remaining region is compared with 2^K neighboring regions for a potential merge. In other terms, the complexity of the scheme is exponential in the number of classification fields. This is the case for all classification methods beyond the TCAM, which can overcome this issue by its native parallelism at the hardware level.

In summary, the relaxation of the tuple merge optimization consists of transferring the tuples matrix into a discretized continuous space with heat density and applying heat diffusivity to the tuple space based on the number of rule overlaps. After running the heat propagation equation over the tuple space, we derive an asymptotic stable distribution of heat energy over the continuous space. After that, a split and merging quantification method is applied to the continuous space in order to retrieve the method of constructing hash tables. In §IV, we will validate the performance of TMR and compare it with other state-of-the-art algorithms.

Table III: Rulesets for Experiments

Ruleset	#Rules	#Hash Tables	
		TSS	TMR
ACL1	9672	768	58
ACL2	9454	252	25
FW1	9375	474	58
FW2	9701	822	66
IPC1	9118	449	45
IPC2	10000	526	52

IV. Validation Results

We carry out extensive experiments to compare the performance of the TM Relaxation (TMR) with four state-of-the-art hash-based packet classification algorithms, including Tuple Space Search (TSS), Pruned Tuple Space Search (PR-TSS), PartitionSort (PS), and TupleMerge (TM). Rulesets used in practice generally have port pair used for detecting specifying application and usually consists of a wildcard and an exact port number [30]. Therefore, the possibility of merging in dimensions relative to port number is limited. If we merge these ports, we will have range (0-16), which seems like we are not using the port dimension in tuples. For this reason, we apply our TMR on tuples defined over the source and destination IP addresses, *i.e.*, port dimensions are fully merged. All the other methods we compare with are implemented with the schemes in their original papers.

We first introduce the validation environment setup in this part. Second, we evaluate the packet classification performance without rule update, then the performance of rule update and packet classification with rule update. Finally, we test the memory footprint.

A. Experimental Setup

In our experiment, all the experiments run on a server with an Intel Xeon CPU E5-2630 v3 @ 2.40GHz, 32 cores and 128GB DDR3 memory. Each core is equipped with a 64 KB L1 data cache and a 256KB L2 cache. A 20MB L3 cache is shared among all cores. Ubuntu 16.04.1 with Linux kernel 4.10.0 is installed as the operating system. To test the performance of different algorithms accurately and in the similar setting, we carry out packet classification and packet updates for all the competitors in separate threads, each allocated to a single core. We choose six different rulesets generated by ClassBench [30] as shown in Table III.

B. Packet Classification without Update

We first compare the packet classification throughput performance of TMR with other algorithms. Figure 4 shows the packet classification throughput without updates in *kilo packets per second* (kpps). We observe that on average TMR's packet classification throughput is $2.8\times$ that of PS, $5.0\times$ that of TSS, $3.0\times$ that of PR-TSS and $1.9\times$ that of TM. Compared with

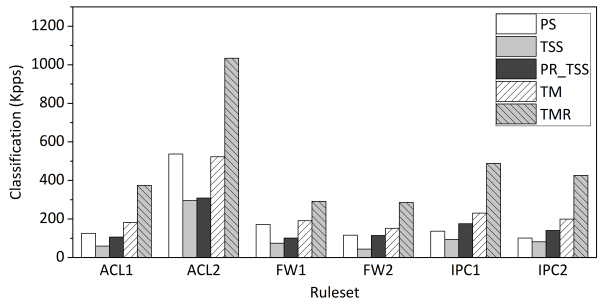


Figure 4: Packet classification performance without update: varying rulesets

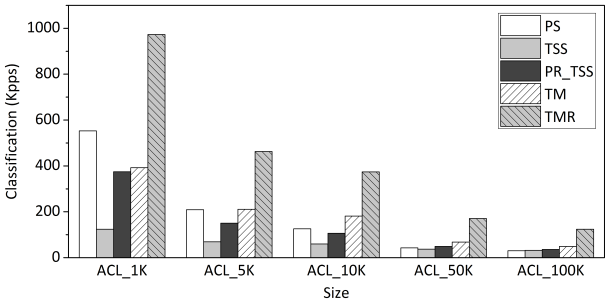
TSS, TMR significantly reduces the number of hash tables as shown in Table III. The better performance of TMR compared to TM shows that the heat propagation heuristics generated better merging of tuples by reducing rule overlapping. Besides, we evaluate the impact of ruleset size on the classification performance. For each of ACL, FW and IPC, we choose the same seed file and generate five different sizes of rulesets: 1K, 5K, 10K, 50K, 100K. Figure 5 indicates that the classification throughput decreases with larger ruleset size. Nonetheless, the classification throughput of TMR is still significantly higher than those of other methods.

C. Update time

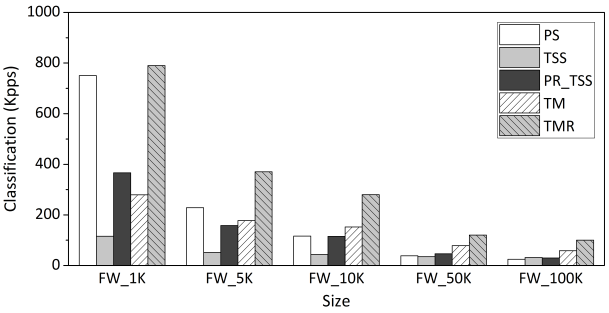
Update operations contain both rule insertion and deletion. To evaluate the delay of the update operation, we split each ruleset from 20% of it. We use the remaining 80% of the ruleset for defining the corresponding packet classification data structure and the split 20% as updates. Then we measured the average dealy of a single update operation. Figure 6 shows that the update speed of TMR is $10.2\times$ that of PS, $1.5\times$ that of PR-TSS and $5.4\times$ that of TM on average. The update speed of TSS is similar to TMR, as they both only need to update the hash table without other structures. The poor update performance of PS is due to the utilization of the tree structure. PR-TSS also uses tries to accelerate the classification at the cost of higher update time. TM fails to achieve fast update due to the splitting-up of the hash table when the number of collisions exceeds a certain threshold. In summary, TMR guarantees fast rules update by preserving the structure of the hash table.

D. Packet Classification with Update

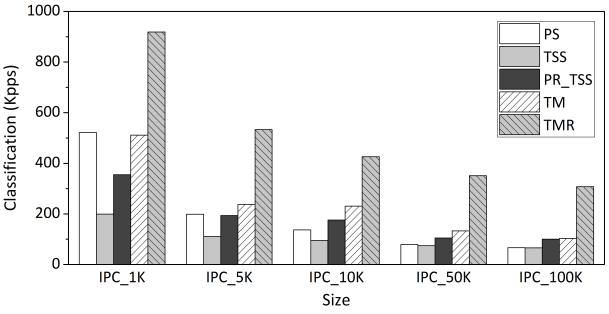
In §IV-C, we test the rule update delay and observe that TMR has a lower delay. However, we also need to check whether the update operation has a serious impact on the classification throughput. For this purpose, we evaluate the packet classification throughput of TMR with a churn rate (update speed) varying from 0.01Kups to 10Kups *(Kilo update per second)*. The 10 Kups is relevant for cases where we use packet classification for flow monitoring and 10K flows are added or removed each second. We use two threads to simulate the setting. One thread continuously classifies the packets, and the other generates updates and changes the classification data



(a) ACL with different size



(b) FW with different size



(c) IPC with different size

Figure 5: Packet classification performance without update: varying size of rulesets

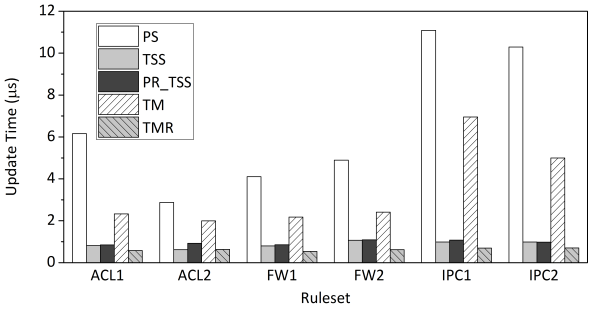


Figure 6: Rule's update time

structure at a given speed. We use a mutex lock on the hash table to manage the concurrent access. The comparison result is in Figure 7. We can observe that the updates slightly slow

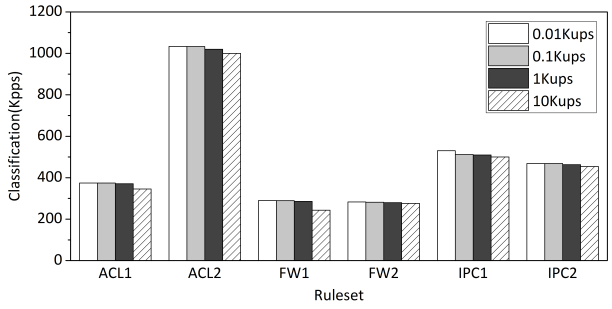


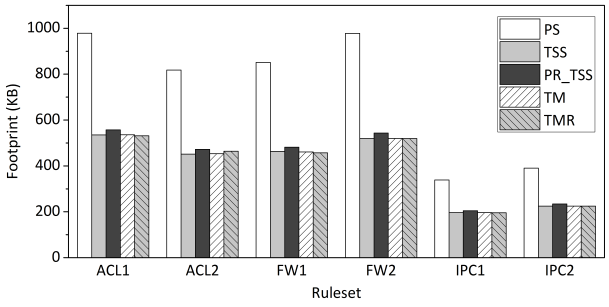
Figure 7: TMR Packet classification throughput with different update speeds

the packet classification speed. However, moving from 0.01 Kups to 10 Kups, the update operation does not decrease the throughput more than 5% .

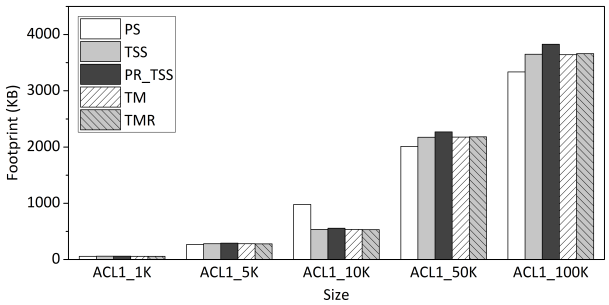
E. Memory Footprint

We compare the memory footprint of TMR with other methods on different rulesets. As shown in Figure 8(a), the memory footprint of PS is larger than the other methods. This is caused by the overhead of the use of pointers in decision trees. PR-TSS is also penalized by the use of tries to filter out some tuples. TSS, TM and TMR are all hash-based algorithms, so the memory footprint of them are similar. The subtle differences between them are due to the rounding effect of the ceiling effect of the individual hash table size. As explained in §III-C, the size is set to $2^{\lceil \log L \rceil}$. In our test, all the hash-based algorithms use multi-position cuckoo hash [29] to construct hash tables.

In Figure 8(b), we show the evolution of footprint with increasing ruleset size. For small rulesets, hash-based methods achieve a smaller footprint. However, with the increasing of the ruleset size, the memory footprint of the hash-based approaches is larger than that of the decision tree in PS. However, the footprint is still relatively low with less than 4 Mbytes for 100K rules! Because in some tree-based approaches that not mentioned in this paper, the space needed can go up to 4 Gbytes for 100K rules.



(a) with different ruleset



(b) with different size of ruleset

Figure 8: Memory Footprint

V. Related Work

Existing packet classification schemes fall into three categories: hardware-based, dimensionality reduction and space partitioning.

Hardware-Based: *T-CAM* [5]–[7] is the de facto standard chip for high-speed packet classification. It applies hardware-based parallel search to achieve low deterministic lookup time. However, *T-CAM* suffers from problems such as limited memory size, power-hungry and slow rule update. Packet classification can also run on other hardware platforms, such as GPU [9] and FPGA [8]. However, it requires specific designs of hardware instructions, chips and programming language. The limited flexibility and high cost are the barriers that prevent these solutions from widespread usage.

Dimensionality Reduction: *Cross-producting* [31] and *RFC* [32] first split multi-dimensional rules into several single-dimensional ones to match individually, and then merge the results. When the ruleset is large, however, the merging procedure becomes very sophisticated. Furthermore, the rule update is slow, because the rule tables corresponding to every dimension need to be updated for one rule update.

Space Partitioning: The main idea of space partitioning approach is to sub-divide the rule space. Instead of matching an incoming packet against the overall ruleset, the classification procedure is divided into two steps: determining the sub-space to search, and matching the packet against the small ruleset in the corresponding sub-space. This approach further falls into two main subcategories: decision tree approaches and hash-based approaches.

The core of decision tree approaches such as *HyperCuts* [33] and *HiCuts* [34] is to partition the search space recursively into several regions until the rule number in each region is below a certain threshold. The efficiency of decision trees allows for high-speed packet classification, but the tree updating is slow. In addition, some rules may need to be copied into multiple partitions, which consumes a large amount of memory. *EffiCuts* [35] and *SmartSplit* [36] propose some other rule space partition policies to reduce the rule replication, but still fail to support fast rule update. *NeuroCuts* [37] is a machine learning-based method to classify packets with decision trees. By utilizing deep reinforcement learning, NeuroCuts provides significant improvements on classification time and memory footprint. However, this method still does not support online

rule update due to the inherent defect of the decision tree.

In contrast, hash-based approaches introduced in §II can update rules quickly but suffer from slow packet classification.

VI. Conclusion

In this paper, we propose the relaxation of the tuple merge (TMR) to support both high-speed packet classification and fast online rule update. TMR utilizes the heat propagation to guide the merging of hash tables. Based on the merging method, TMR can reduce the number of hash tables while maintaining the number of collisions low. We have evaluated the performance of TMR using different types of rulesets. TMR achieves $3.2\times$ classification speed and $4.6\times$ update speed on average compared with state-of-the-art algorithms.

Acknowledgment

This work is supported by the National Key R&D program of China (Grant NO. 2019YFB1802800), and the National Science Fund for Distinguished Young Scholars (Grant NO. 61725206).

References

[1] M. Suh, S. H. Park, B. Lee, and S. Yang, "Building firewall over the software-defined network controller," in *Advanced Communication Technology (ICACT), 2014 16th International Conference on*. IEEE, 2014, pp. 744–748.

[2] C. Lenzen and R. Wattenhofer, "Tight bounds for parallel randomized load balancing," in *Proceedings of the forty-third annual ACM symposium on Theory of computing*. ACM, 2011, pp. 11–20.

[3] M. S. Seddiki, M. Shahbaz, S. Donovan, S. Grover, M. Park, N. Feamster, and Y.-Q. Song, "Flowqos: Qos for the rest of us," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 207–208.

[4] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc)," *IEEE Network*, vol. 28, no. 6, pp. 18–26, 2014.

[5] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary cams," in *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4. ACM, 2005, pp. 193–204.

[6] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Sax-pac (scalable and expressive packet classification)," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 15–26.

[7] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast tcam updates," *IEEE/ACM Transactions on Networking (TON)*, vol. 26, no. 1, pp. 217–230, 2018.

[8] W. Jiang and V. K. Prasanna, "Scalable packet classification on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 9, pp. 1668–1680, 2012.

[9] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman, "Multilayer packet classification with graphics processing units," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2728–2741, 2016.

[10] N. Mckeown, T. W. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "Openflow: enabling innovation in campus networks," *acm special interest group on data communication*, vol. 38, no. 2, pp. 69–74, 2008.

[11] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[12] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[13] D. Firestone, "Vfp: A virtual switch platform for host sdn in the public cloud." in *NSDI*, vol. 17, 2017, pp. 315–328.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

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

[16] P. Gupta and N. Mckeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[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] B. Chazelle, "Lower bounds for orthogonal range searching: Part ii. the arithmetic model," *J. ACM*, vol. 37, no. 3, pp. 439–463, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/79147.79149

[19] M. H. Overmars, *The Design of Dynamic Data Structures*, ser. Lecture Notes in Computer Science. Springer, 1983, vol. 156. [Online]. Available: https://doi.org/10.1007/BFb0014927

[20] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 135–146.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[22] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng, "A sorted partitioning approach to high-speed and fast-update openflow classification," in *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[23] J. Daly, V. Bruschi, L. Linguaglossa, S. Pontarelli, D. Rossi, J. Tollet, E. Torng, and A. Yourtchenko, "Tuplemerge: Fast software packet processing for online packet classification," *IEEE/ACM transactions on networking*, vol. 27, no. 4, pp. 1417–1431, 2019.

[24] C. W. Gardiner, *Handbook of stochastic methods for physics, chemistry and the natural sciences*, 3rd ed., ser. Springer Series in Synergetics. Berlin: Springer-Verlag, 2004, vol. 13.

[25] K.-J. Bathe and E. L. Wilson, *Numerical methods in finite element analysis*. Prentice-Hall, 1976.

[26] D. Widder, *The Heat Equation*, ser. Pure and Applied Mathematics. Elsevier Science, 1976. [Online]. Available: https://books.google.fr/books?id=5BPILpGGGXsC

[27] M. Spann and R. Wilson, "A quad-tree approach to image segmentation which combines statistical and spatial information," *Pattern Recognition*, vol. 18, no. 3, pp. 257 – 269, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0031320385900512

[28] M. Neuhäuser, *Wilcoxon–Mann–Whitney Test*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1656–1658. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_615

[29] U. Erlingsson, M. Manasse, and McSherryn, "A cool and practical alternative to traditional hash tables," in *Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS'06), Santa Clara*, vol. 1. IEEE, 2006.

[30] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Transactions on Networking (ton)*, vol. 15, no. 3, pp. 499–511, 2007.

[31] P.-C. Wang, "Scalable packet classification with controlled cross-producting," *Computer Networks*, vol. 53, no. 6, pp. 821–834, 2009.

[32] P. Gupta and N. McKeown, "Packet classification on multiple fields," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 147–160, 1999.

[33] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2003, pp. 213–224.

[34] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Hot Interconnects VII*, vol. 40, 1999.

[35] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 207–218, 2011.

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

[37] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," *arXiv preprint arXiv:1902.10319*, 2019.