CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification

Wenjun Li[†], Xianfeng Li[†], Hui Li[†*] and Gaogang Xie[§]
[†]School of Electronic and Computer Engineering, Peking University, China, [§]ICT, CAS, China
wenjunli@pku.edu.cn, lixianfeng@pkusz.edu.cn, lih64@pku.edu.cn, xie@ict.ac.cn

***Abstract*—Efficient algorithmic solutions for multi-field packet classification have been a challenging problem for many years. This problem is becoming even worse in the era of Software Defined Network (SDN), where flow tables with increasing complexities are playing a central role in the forwarding plane of SDN. In this paper, we first conduct an unprecedented in-depth reasoning on issues that led to the unsuccess of the major quests for scalable algorithmic solutions. With the insights obtained, we propose a practical framework called CutSplit, which can exploit the benefits of cutting and splitting techniques adaptively. By addressing the central problem caused by uncontrollable rule replications suffered by the major efforts, CutSplit not only pushes the performance of algorithmic packet classification more closely to hardware-based solutions, but also reduces the memory consumption to a practical level. Moreover, our work achieves low pre-processing time for rule updates, a problem that has long been ignored by previous decision-trees, but is becoming more relevant in the context of SDN due to frequent updates of rules. Experimental results show that using ClassBench, CutSplit achieves a memory reduction over 10 times, as well as 3x improvement on performance in terms of the number of memory access on average.**

Keywords—Packet Classification; OpenFlow; Decision Tree; Algorithm; Firewall

I. Introduction

Modern network devices provide services beyond basic packet forwarding, such as security, policy routing and Quality of Service (QoS). Packet classification is the core functionality for supporting these services. The purpose of packet classification is to find a matching rule from a packet classifier for each incoming packet, and apply a corresponding action to the packet. A packet classifier is a set of rules, with each rule consisting of a tuple of field values (exact value, prefix or range) and an action to be taken in case of a matching. An example 12-tuple OpenFlow [1] classifier is shown in Table I. As the bottleneck of advanced forwarding, packet classification has attracted research attentions for almost two decades.

Current packet classifications can be categorized broadly into two major approaches: architectural and algorithmic [2][3][4]. Architectural approaches based on Ternary Content Addressable Memory (TCAM) have been the dominated implementation of packet classification in industry. Although TCAM enables parallel lookups on rules for line-speed classification, it is expensive, area-inefficient and power-hungry, which seriously limit its scalability. During the past decade, a lot methods and algorithms had been proposed to alleviate these problems, such as *classifier minimization* [5][6][7][8], *range encoding* [9][10][11][12][13], *circuit modification* [13][14] and *pre-classifier* [15][16]. However, due to inherent limitations of TCAM, the TCAM capacity is not expected to increase significantly in the near future [4]. Worse still, with the deployment of SDN/NFV based applications, the number of rule fields and size of classifiers are increasing dramatically, outpacing the TCAM capacity evolution. For example, current OpenFlow Switch examines more than 15 fields to categorize packets into different flows, and this number is expected to grow in the future [17]. Thus, multi-field packet classification has become even more prominent and challenging than ever.

Recently, researchers have been actively investigating less expensive, more energy-efficient and more scalable algorithmic alternatives to TCAM-based hardware solutions, such as *hash-based algorithms* [17][36][39], *hardware-assisted schemes* [19][20][21][37] and *decision-tree techniques* [18][22][24][25]. Among them, decision-tree has been recognized as one of the most promising approaches, since they can be well applied to rules with more fields and pipelined for high classification throughput [26]. In general, there have been two major threads of research building decision-trees: *equal-sized cutting* and *equal-dense splitting*. Cutting based schemes, such as HiCuts [27] and HyperCuts [28], separate the searching space into many equal-sized sub-spaces using local optimizations. But both schemes have the same *rule replication* problems, especially for large rule sets. EffiCuts [22], a well-known cutting scheme, significantly reduces memory overhead of previous cutting algorithms by separating rules into at most 2^F subsets for F-tuple classifiers. As an improvement, HybridCuts [24] achieves a significant reduction on the number of subsets, which in turn reduces the overall memory accesses. In contrast, HyperSplit [29], a well-known splitting scheme, splits the searching space into two unequal-sized sub-spaces that contain nearly equal number of rules. To achieve better scalability for different rule sets, SmartSplit [25] separates rules into a few subsets to build balanced trees dynamically. However, as far as we know, seldom of these state-of-the-art approaches can make an excellent trade-off among storage, performance and updating, which seriously limit their scalability.

In this paper, we first seek to understand the reasons behind the difficulty in designing scalable decision-trees for multi-

[*]Hui Li is also with the Shenzhen Key Lab of Information Theory & Future Network Architecture, Future Network PKU Lab of National Major Research Infrastructure, Shenzhen Engineering Lab of Converged Networking Technology, Huawei & PKU Jointly Engineering Lab of Future Network Based on SDN and the PKU Institute of Big Data Technology, Shenzhen Graduate School, Peking University, China.

Table I. An example OpenFlow 1.0 classifier

Rule id	Ingress port	Ether src	Ether dst	Ether type	VLAN id	VLAN priority	IP src	IP dst	IP proto	IP ToS bits	TCP/UDP Src Port	TCP/UDP Dst Port	*Action*
R1	*3*	*	*	*2048*	*	*	*206.159.213.125/32*	*101.152.182.8/30*	*0x06f/0xff*	*0*	*1024 : 65535*	*	*drop*
R2	*3*	*	*	*2048*	*	*	*15.25.70.8/30*	*	*	*0*	*	*0:1599*	*forward*
R3	*5*	*	*	*2048*	*	*	*	*18.152.125.32/30*	*0x11/0xff*	*1*	*1024 : 65535*	*1024 : 65535*	*enqueue*
R4	*5*	*	*	*2048*	*	*	*206.159.213.125/32*	*	*0x06f/0xff*	*1*	*	*80*	*forward*
R5	*	*	*	*	*	*	*	*	*	*	*	*	*drop*

field packet classification. After that, we make some novel observations on typical *5*-tuple rule sets as well as OpenFlow based rule tables, which can help us separate rules into few subsets. Finally, we present our proposed CutSplit, a decision-tree scheme combing cutting and splitting for packet classification, which can improve storage efficiency and performance simultaneously. Moreover, our work achieves low pre-processing time for rule updates, a problem that has long been ignored by most previous decision-trees.

We evaluate our algorithm using ClassBench [30] and show that, even for rule sets up to 100K entries, CutSplit is able to produce a very small number of short decision trees with low memory overhead. Compared with EffiCuts, CutSplit achieves a memory reduction over 10 times, as well as 3x improvement on performance in terms of the number of memory access on average. Moreover, CutSplit can rebuild decision-trees in a few seconds as well as sub-trees in 50us on average.

The rest of the paper is organized as follows. In Section II, we first briefly summarize the related work. After that, we review and analyze reasons behind the difficulty in designing scalable decision-trees in Section III. Based on these analyses, we make a set of important observations and present the technical details of CutSplit in Section IV. Section V provides experimental results. Finally, Section VI draws conclusion.

II. Background and Related work

In this section, we first review the background about decision-tree based packet classification and two major threads of research on decision-tree constructions. After that, we briefly describe the related work and recent efforts in threads of cutting and splitting respectively.

A. Decision-tree based Packet Classification

Decision-tree is one of the most wildly studied algorithmic approaches, as well as *decomposition* [31][32][33][34] and *tuple space* [17][35][36]. In decision-tree based schemes, the geometric view of the packet classification problem is taken and a decision tree is built. The root node of the tree covers the whole searching space containing all rules. Each rule is considered as a hypercube in an *F*-dimensional space, where *F* is the number of fields in a rule. Each incoming packet defines a point in this *F*-dimensional space. They work by recursively partitioning the searching space into smaller sub-spaces until a predefined number of rules are contained by each sub-space. In case a rule spans multiple sub-spaces, *rule replication* happens, which is an undesirable case (e.g., *R3, R4* and *R6* in Figure 1b). When a packet arrives, the decision tree is traversed to find a matching rule at a leaf node. According to the partitioning method on searching space, current decision-tree based techniques can be categorized broadly into two major approaches: ***equal-sized cutting*** and ***equal-dense splitting***.

Table II. An example 2-tuple classifier

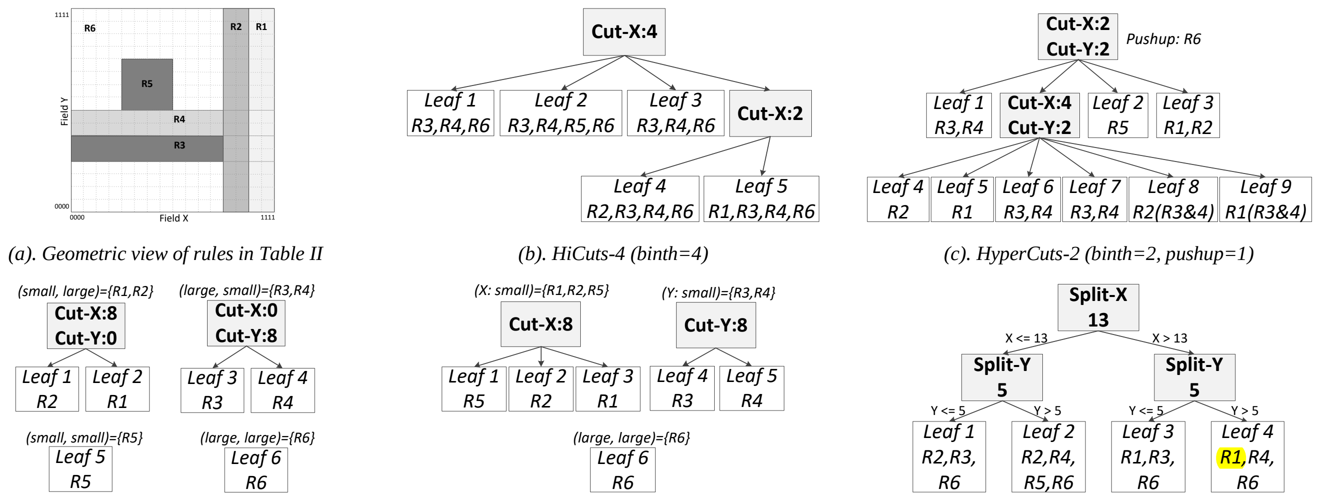
Rule #	Priority	Field X	Field Y	*Action*
R1	*1*	*111**	*	*drop*
R2	*2*	*110**	*	*forward*
R3	*3*	*	*010**	*enqueue*
R4	*4*	*	*011**	*modify*
R5	*5*	*01***	*10***	*forward*
R6	*6*	*	*	*drop*

Next, we briefly summarize the related work and some recent efforts. For the convenience of description, we use a small example of 2-tuple rule set shown in Table II for subsequent discussions. Figure 1a shows the geometric representation of example rules given in Table II.

B. Cutting based Decision-trees

Cutting based schemes, such as HiCuts [27] and HyperCuts [28], separate the searching space into many equal-sized sub-spaces using local optimizations. HiCuts cuts the searching space into many equal-sized sub-spaces recursively until the rules covered by each sub-space is less than the pre-defined bucket size called *binth*. To reduce memory consumption, HiCuts uses some heuristics to select the cutting dimension and decides how many sub-spaces should be cut using a space optimization function with a parameter called *spfac*. HyperCuts can be considered as an improved version of HiCuts, which is more flexible in that it allows cutting on multiple fields per step, resulting in a fatter and shorter decision tree. Besides, several optimizations are adopted in HyperCuts, such as *node merging*, *rule overlap*, *region compaction* and *pushing common rule subsets upwards*. But both HiCuts and HyperCuts have the same *rule replication* problem for rules spanning multiple sub-spaces, especially for large rule tables. Figure 1b and Figure 1c show decision-trees generated by HiCuts and HyperCuts.

EffiCuts [22] observed that real-life rules exhibit several inherent characteristics, and a good rule set partitioning can reduce *rule replications* dramatically. Thus, instead of building a single decision-tree for all rules, EffiCuts separates rules into several subsets with each subset creates its own decision-tree independently using HyperCuts. However, with all *F* fields considered, up to 2^F decision-trees can be generated for *F*-tuple classifiers, resulting in a large number of overall memory accesses. In contrast, HybridCuts [24] separates rules based on single individual rule field rather than all *F* fields in EffiCuts, thus, HybridCuts achieves a significant reduction on the number of subsets (i.e., from 2^F to $F+1$), which in turn reduces the overall memory accesses. However, due to the employment of HyperCuts, the worst-case search performance of HybridCuts is unbounded. Worst still, with the increase of the number of rule fields and the size of classifiers, the performance of HybridCuts may drop dramatically. Figure 1d and Figure 1e show decision-trees generated by EffiCuts and HybridCuts.



(a). Geometric view of rules in Table II *(b). HiCuts-4 (binth=4)* *(c). HyperCuts-2 (binth=2, pushup=1)*

(d). EffiCuts-1 (binth=1) *(e). HybridCuts-1 (binth=1)* *(f). HyperSplit-4 (binth=4)*

Figure 1. Review on related work

C. *Splitting based Decision-trees*

In order to reduce *rule replications* suffered from equal-sized cuttings, schemes based on splitting divide the searching space into unequal-sized sub-spaces containing nearly equal number of rules. HyperSplit [29], a well-known splitting scheme, splits the searching space into two unequal-sized sub-spaces containing nearly equal number of rules. Due to its simple binary separation in sub-spaces, the worst-case search performance is explicit (i.e., $F*log(2N+1)$ for N F-tuple rules). However, even with the optimized binary space splitting, the memory consumption of HyperSplit still grows exponentially as the number of rules increases. Figure 1f shows the decision-tree generated by HyperSplit, we can see that in each internal node, HyperSplit splits the selected field range into two sub-ranges, with each sub-range covering rules as balanced as possible.

ParaSplit [23] proposes a rule set partitioning algorithm to reduce rule set complexity, which significantly reduces the overall memory consumption in HyperSplit. However, ParaSplit employs a complex heuristic for rule set partitioning, which may require tens of thousands iterations to reach an optimal partitioning. SmartSplit [25], a recently proposed splitting based decision-tree, achieves high-speed classification by leveraging the logarithmic search times of balanced search trees. Unfortunately, its variable data structures and operations raise its barrier for practical implementation. Besides, it takes a long pre-processing time to build decision-trees.

Clearly, none of the existing decision-tree techniques can make an excellent trade-off among storage, performance and updating, which seriously limit their scalability.

III. In-depth Challenge Review

In this section, we seek to further understand the reasons behind the difficulty in designing scalable decision-trees. We first list some challenges and key metrics for scalable packet classification. Then we make several quantitative evaluations on cutting and splitting to reveal the key problems faced by decision-trees. Finally, we review some efforts and analyze their effectiveness. For the convenience of evaluation and comparison, we adopt typical *5*-tuple classifiers and ClassBench [30] as our test bed in this work.

A. *Review on Metrics*

Packet classification is a challenging problem due to the line-speed requirement of forwarding engines, in which a packet has to be processed within a very short time. On-chip caches can reduce memory access time, but cache memory is not scalable with the size of the flow table. Thus, two primary metrics for software-based packet classification are memory consumption and the number of memory accesses. A scalable packet classification should meet the following design goals:

- **Low memory consumption**: Memory efficient scheme enables the constructed data structure to accommodate in small on-chip memory.
- **Low memory accesses**: The fewer memory access for each lookup, the higher throughput for classification, which is critical to high-speed network.
- **Bounded worst-case performance**: To guarantee the overall performance in real-life systems, it should bound the memory access under the worst-case.

To achieve high classification speed, most existing methods focused on improving search performance while sacrificing update performance. However, due to frequent updates of rules in the context of SDN, rule update is becoming more relevant than ever. Thus, update performance is another key metric for algorithmic packet classification. To achieve fast rule updates, a scalable classification algorithm should also meet the following design goals:

- **Low pre-processing time**: Low pre-processing time enables quick reconstruction of data structure, which is important for rule updates.
- **Easy for incremental updates**: For each rule update, modest modifications to the data structure enable fast incremental updates other than rebuild from scratch.

B. *Review and Analysis on Challenges*

By examining cutting and splitting processes for different classifiers, we identify the main trouble-maker for decision-trees: ***Rule replication***.

Unlike prior *rule overlapping* in [21] or *orthogonal rules* in [25], *rule replication* is a more comprehensive metric for evaluating effectiveness of decision-trees. When a set of rules are highly overlapped or orthogonal with each other, it will be very difficult to separate them from each other by cutting or splitting. Take HiCuts in Figure 1b as an example, in order to separate *R1* from *R2*, HiCuts needs to cut *Field X* into at least 8 sub-ranges intuitively. However, it will result in serious *rule replications* for *R3&R4&R6*, where these rules are overlapped and orthogonal with *R1&R2*. To avoid overall memory explosion caused by *rule replications*, a two stage cuttings are adopted in Figure 1b, which may in turn lead to higher decision-trees. Thus, *rule replication* can not only reflect the degree of *rule overlapping*, but also the influence on overall memory consumption and memory access.

To get more insights on the problem of *rule replication*, we make some experimental analyses for both cutting and splitting based decision-trees. Before that, we first give some definitions and notations:

- *Rule replication factor*: #stored rules / rule set size, for example, the *rule replication factor* for HiCuts in Figure 1b is *(3+4+3+4+4)/6=3*.

Figure 2 shows the results of *rule replication factor* for different type of seed rule sets, which are available in ClassBench. We can see that, cutting based HyperCuts suffers from serious *rule replications* problem, especially obvious for firewall rule set which contain a significant fraction of rules with many wildcard fields. In contrast, by using a simpler binary splitting in searching space, HyperSplit reduces *rule replications* obviously, which can be seen from Figure 2. However, with the increase of rule set size, both HyperCuts and HyperSplit suffer from an exponential explosion of *rule replications*. For example, for *fw-10k* rule set in Figure 3, both the *rule replication factor* of HyperCuts and HyperSplit could be more than 1000, indicating that at least 10 million rule pointers are required in decision-trees. Clearly, this is impractical in current cache hierarchies.

C. *Review and Analysis on Prior Art*

To alleviate the *rule replication* problem, a lot optimization methods and algorithms had been proposed. However, these efforts focus on reducing *rule replications* while sacrificing searching and updating performance. Next, we briefly review these efforts, and then analyze their effectiveness.

Optimization Methods. Many optimizations have been proposed to alleviate above *rule replication* problem, such as *pushing-upwards*, *rule overlap* and *region compaction* in HyperCuts. Among these optimizations, *pushing-upwards* is the most effective and widely used method, which has been applied in many recent decision-trees, such as EffiCuts, HybridCuts and SmartSplit. Essentially, the rationale behind this optimization is simple: by pushing common rules upwards, the following cuttings can avoid vast *rule replications* caused

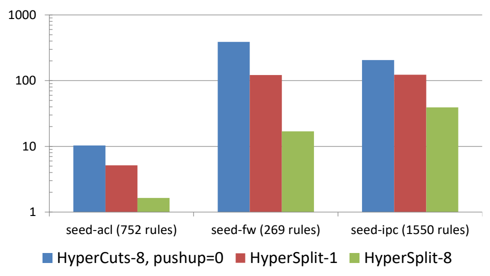


Figure 2. Evaluation of *rule replication factor* for seed rule sets

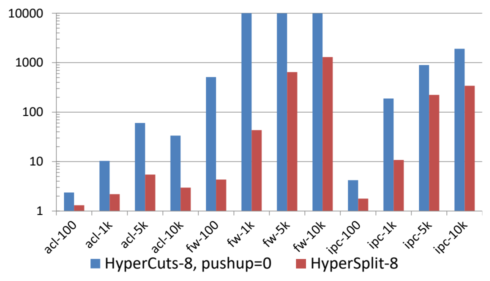


Figure 3. Evaluation of *rule replication factor* for different rule sets

by these few boring rules, which in turn reduces the overall memory consumption significantly (e.g., rule *R6* in Figure 1c). Figure 4 shows the effectiveness of *pushing-upwards* optimization for HyperCuts. Obviously, the *rule replication factor* significantly drops down with the adoption of *pushing-upwards* optimization. Unfortunately, this optimization does not work well in all cases. Due to the reason that each pushed rule requires an individual lookup during decision-tree searching, the overall number of memory access may be increased compared with original algorithms. As illustrated in Figure 5, the overall memory accesses overhead the worst-case tree depth significantly, with up to 10 times in some cases.

Besides, some other optimizations, such as *rule overlap* and *region compaction*, can further reduce memory consumptions for decision-trees. However, these optimizations may increase the difficulties for rule updates. Take Figure 1c as an example, with the adoption of *rule overlap*, rule *R3* and *R4* should be removed from *Leaf 8/9*, as they are completely covered by another rule with a higher priority (i.e., *R2* and *R1*). However, it is always easy to throw away, but difficult to pick up. If rule *R1* or *R2* is deleted from classifiers, a reconstruction of the decision-tree is needed since this delete operation may lead to confusion in *Leaf 8/9*.

Rule Set Partitioning. Separating rules into subsets can reduce rule overlapping in each subset, which in turn improves the problem of *rule replication* in each individual decision-tree. EffiCuts is the most representative algorithm using rule set partitioning. Figure 6 and Figure 7 show the effectiveness as well as the influence for partitioned EffiCuts, compared with un-partitioned HyperCuts. As shown in Figure 6, EffiCuts improves *rule replications* dramatically, which in turn significantly reduce memory consumptions. However, this aggressive partitioning method also brings trouble to lookup performance illustrated in Figure 7. We can see that there are still a lot of sub-trees generated by EffiCuts even with the adoption of *sub-tree merging* option, which may lead to a larger overall memory accesses.

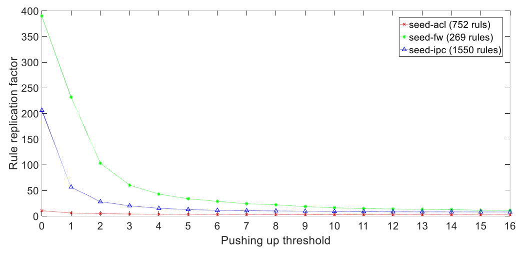


Figure 4. Effectiveness of *pushing-upwards* for HyperCuts-8

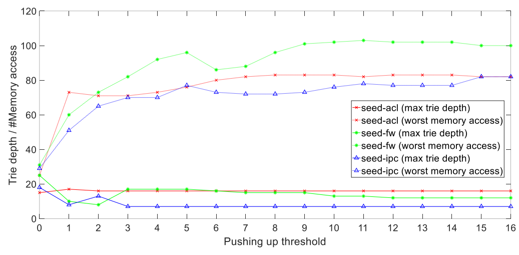


Figure 5. Influence of *pushing-upwards* for HyperCuts-8

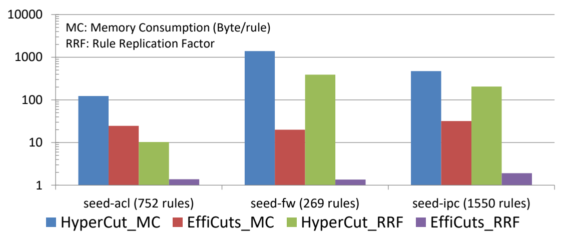


Figure 6. Effectiveness of *rule set partitioning* for EffiCuts-8

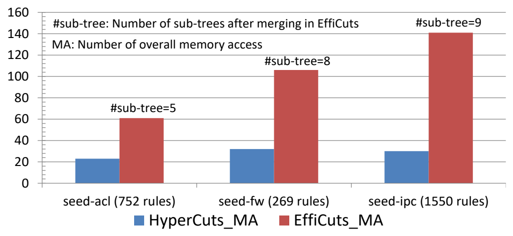


Figure 7. Influence of *rule set partitioning* for EffiCuts-8

Recently, HybridCuts proposes an advanced partitioning which can reduce the number of sub-trees from 2^F to $F+1$ for F-tuple classifiers, and this number is further reduced to 3 with an optimization for typical 5-tuple rules. SmartSplit, another partitioning based scheme, separates rules into at most 4 subsets based on a *memory estimator*. However, both these partitioning schemes are based on the observations about address fields for typical 5-tuple classifiers, which seriously limit their adaptability for general flow tables.

Cutting or Splitting? Recently, more and more researchers have realized that a simple cutting or splitting is not enough for more and more complex classifiers, and a combination of them is a practical choice to reduce *rule replications*. But most of current schemes suffer a poor scalability from classifier size.

EffiCuts, employs HyperCuts with an *equi-dense* cutting (i.e., splitting) option to build sub-trees for partitioned subsets. HybridCuts, as suggested by its name, makes use of a combination of one- and multi-dimensional cuttings for tree constructions. Instead of using these predetermined cutting or splitting, SmartSplit introduces a *memory estimator* to make use of cutting or splitting dynamically. These combinations are far from achieving optimal effectiveness. From experimental evaluations in Section V, we can conclude that the performance of these algorithms drop quickly with the size of rule sets increases. One of the primary reasons is that, all these schemes simply adopt HyperCuts or HyperSplit during the construction of the decision-trees, which may seriously limit their scalability for larger classifiers. Actually, with the increase of the size of the classifier, we will encounter more and more *overlapped rules* or quasi *orthogonal rules* in partitioned subsets, which may affect the overall performance significantly.

Thus, these existing efforts focus on reducing *rule replications* while sacrificing search or update performance.

IV. The Proposed Algorithm

In this section, we first introduce ideas behind the design of scalable packet classification. After that, we make several important observations on typical 5-tuple rule sets as well as OpenFlow based rule tables, which can help us to separate rules into a very few subsets. Finally, we describe a practical framework called CutSplit to exploit the benefits of cutting and splitting techniques adaptively. By addressing the central problem caused by uncontrollable *rule replications* suffered by the major efforts, CutSplit not only pushes the performance of algorithmic packet classification more closely to hardware-based solutions, but also reduces the memory consumption to a practical level. Moreover, our work achieves low pre-processing time for rule updates, a problem that has long been ignored by previous decision-trees, but is becoming more relevant in the context of SDN due to frequent updates of rules.

A. Ideas

According to above reviews, cutting can separate searching space into smaller sub-spaces quickly for faster classification, but it suffers from serious *rule replications* and implicit worst-case search performance. In contrast, splitting can significantly alleviate *rule replication* problem and offer a bounded worst-case search performance, but its binary splitting worsens the overall search performance. Therefore, to foster the strengths and circumvent the weaknesses of cutting and splitting, the idea directly perceived is to combine the following two strategies:

- **Faster Pre-Cutting**: Cutting based separating on the searching space, which can reach any smaller sub-spaces with a very few steps.
- **Explicit Post-Splitting**: Splitting based searching on sub-spaces, which can achieve deterministic worst-case search bound and lower memory consumption.

Figure 8 shows the framework of the following CutSplit. However, in order to design scalable algorithms to meet above goals, effective combination of these two ideas still faces several difficulties and challenges:

- For memory storage: Since *rule overlapping* can lead to serious *rule replications*, how to alleviate this problem during cutting and splitting?

- For memory access: Since prior cuttings have no explicit worst-case bound for search, how to guarantee the worst-case bound for pre-cuttings? Besides, if partitioning employed, in order to reduce overall memory accesses, how to generate rule subsets as few as possible?
- For rule updates: Since the awareness of the importance of optimizations alleviating rule overlapping problems comes into consideration, how to minimize impacts on update?

The answers to these questions are key ideas in this paper. Our solution can be summarized as the following three steps:

- Step 1: **Partitioning based on a very few small fields**. In order to reduce rule overlapping and number of rule subsets, we separate rules into subsets based on their characteristics shared in a very few fields.
- Step 2: **Modified cuttings without prior optimizations**. After partitioning of rule sets, we get a set of fields for each subset, where a set of simpler but more effective cuttings can be applied for faster classification.
- Step 3: **Combination of pre-cutting & post-splitting.** Thanks to the clever partitioning and the first stage faster cuttings, we can obtain any sub-space containing much less rules without the trouble of serious *rule replications*, which enables a simpler binary splitting for subsequent processing with bounded worst-case performance. Besides, *rule replication* only appear at the second splitting stage, leading to high performance for incremental updates.

B. *Observations & Partitioning for Rule Sets*

Classifier rules in real-life applications have structural redundancies and several inherent characteristics that can be exploited to reduce the complexity [19][27][28][31][33][38]. Thus, we use the publicly available ClassBench and OpenFlow-like rule tables for study to make observations on common characteristics of rule sets. The two OpenFlow-like rule tables are generated based on 216 real-life rules from enterprise customers. We first give a few definitions, then we present the key observations related to the following rule set partitioning.

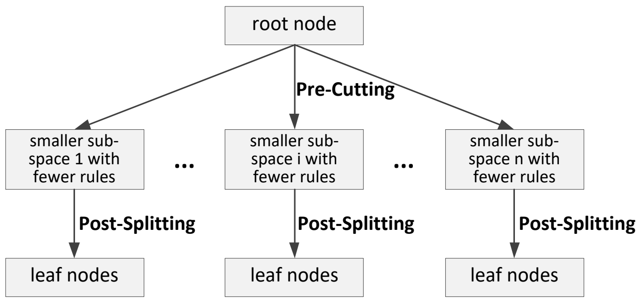
1) Definitions

Given an N-dimensional rule $R = (F_1, \dots F_i, \dots F_N)$ and a threshold value vector $T = (T_1, \dots T_i, \dots T_N)$, where $i \in \{1, 2, \dots N\}$, we give some definitions for field F_i as follows:

- F_i is a ***big field***: the range length of field $F_i >$ threshold value T_i.
- F_i is a ***small field***: the range length of field $F_i \leqslant$ threshold value T_i.

Based on above definitions for rule field F_i, we further give some definitions for rule R as follows:

- R is a ***big rule***: $\forall i \in \{1, 2, \dots N\}$, F_i in R is a *big field*.
- R is a ***small-k rule***: R contains at least k *small fields*.



root node
Pre-Cutting
smaller sub-space 1 with fewer rules
...
smaller sub-space i with fewer rules
...
smaller sub-space n with fewer rules
Post-Splitting
Post-Splitting
Post-Splitting
leaf nodes
leaf nodes
leaf nodes

Figure 8. The framework of CutSplit

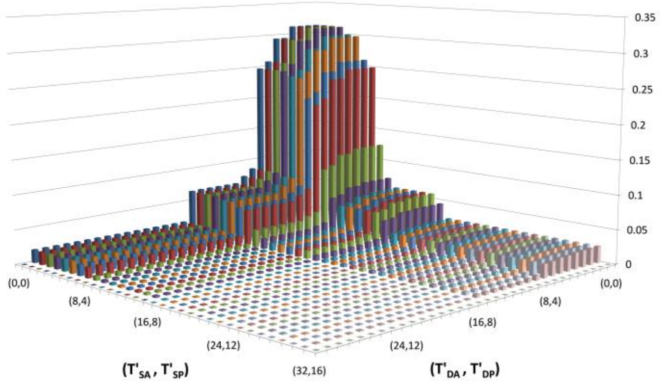


Figure 9. The ratio of *big rules* for *seed-ipc* rule set

For the sake of convenience in writing, we use a logarithmic vector T' to represent T equivalently. For example, if we set $T_i = 2^{16}$, then index logarithmic $T_i' = 16$.

2) At Least One Small Field & Very Few Small Fields

Based on above definitions, we make several observations for a number of rule sets. For example, Figure 9 shows the ratio of *big rules* for IP Chains (IPC) rule set. It is clear that the ratio of *big rules* is very low even under very demanding thresholds. Due to length limitation, more results for Access Control Lists (ACL) and Firewalls (FW) rule sets will be available in [40]. Table III shows statistical results for 5-tuple rule sets and OpenFlow-like rule tables. Clearly, this observation is still effective for OpenFlow-like rule tables. From Table III, we can also draw a conclusion that the vast majority of rules have at least one as well as a very few number of *small fields*.

3) Rule Set Partitioning

Based on above observations, we propose a partitioning algorithm to separate rules into a few subsets. The purpose of our partitioning is to obtain a few subsets without duplicates among each other. For each subset, all contained rules should share a common characteristic for a set of rule fields: *small field*. Next, we introduce a simple heuristic as follows:

--Step1: Remove *big rules*. Since the number of *big rules* is negligible, we can simply apply HyperSplit for these rules.

--Step2: Select a few distinct fields. Count the number of distinct *small field* in each field and pick up a few highest ones, where the vast majority (e.g., >95%) rules contain at least one *small field* in selected fields. For the remaining rules, we can simple merge them into above *big rules*.

Table III. Statistical results for 5-tuple & OpenFlow-like rules (Assuming the value of T_i is half of range length in field F_i)

Rule set(#rules)	**Number of *big rules***	**Number of *small-k* rules**				
		k=1	***k=2***	***k=3***	***k=4***	***k⩾5***
seed-acl(752)	3	749	739	425	0	0
seed-fw(269)	4	265	218	17	2	0
seed-ipc(1550)	2	1548	1472	789	5	0
openflow-1(716)	0	716	708	655	426	0
openflow-2(864)	0	864	852	761	429	0

--Step3: Fields-wise partitioning. Assume *M* fields have been selected for *F*-tuple rule sets. We categorize rules based on field length (i.e., *big* or *small*) in all selected fields, leading to at most 2^M*-1* subsets. This partitioning is different from EffiCuts from two perspectives: fewer fields and more flexible definition about *small/big field*.

--Step4: Selective subset merging. For subsets containing a very few rules, we can merge these rules into other subsets that have fewer *small fields*. Due to the consideration on its relevance and space limitation, we do not elaborate on this algorithm in this paper.

Take rule set in Table I as an example, we first remove *R5* into a *big rule* subset, after then, we calculate the number of distinct *small fields* in each field and pick up *ip_src* & *ip_dst* as two most distinct fields. Thus, we can partition the rule set into four subset: *big_subset*={*R5*}, ($small_{ip\_src}$, $small_{ip\_dst}$)={*R1*}, (big_{ip_src}, $small_{ip\_dst}$)= {*R3*} and ($small_{ip\_src}$, big_{ip_dst})={*R2*, *R4*}. Finally, we can merge ($small_{ip\_src}$, $small_{ip\_dst}$) with (big_{ip_src}, $small_{ip\_dst}$) for a new subset (*NULL*, $small_{ip\_dst}$)={*R1*, *R3*}. Thus, three subsets are generated for the sample rule set: {*R5*}, {*R2*, *R4*} and {*R1*, *R3*}.

C. *CutSplit: Pre-Cutting & Post-Splitting*

The rationale behind above strategy of rule set partitioning is simple: by grouping rules that are *small* in the same fields, we get a few dimensions in the space where the extensive *rule replications* bothering traditional cuttings by wide overlaps are significantly reduced. Moreover, subsets partitioned by this way enable simple and space-efficient fixed-cutting algorithms. The framework of CutSplit is shown in Figure 8. Next, we describe Pre-Cutting and Post-Splitting respectively.

Stage 1: ***Faster Pre-Cutting***. For partitioned rule subsets, some simpler and more space-efficient cuttings can be applied to *small fields*, separating searching space into much smaller sub-spaces in a few steps. Figure 10 shows two cutting examples for two sample rule sets with different number of *small fields*. Although the two cuttings in Figure 10 share some similarities to HiCuts and HyperCuts, they are different from the two classical techniques in two aspects: fixed cutting fields and no optimizations. Compared with FiCuts in HybridCuts, our multi-field cuttings are more adaptive to rule sets with more than one *small field*. The cutting processes stop when the number of contained rules is smaller than a pre-defined value *binth* or the sub-spaces reach the threshold of sub-space which is defined by threshold values of each small field. Take Figure 10 as an example, assume *binth* < 4 and the threshold value vector *T = (4, 4)*, the first stage cuttings should stop in gray

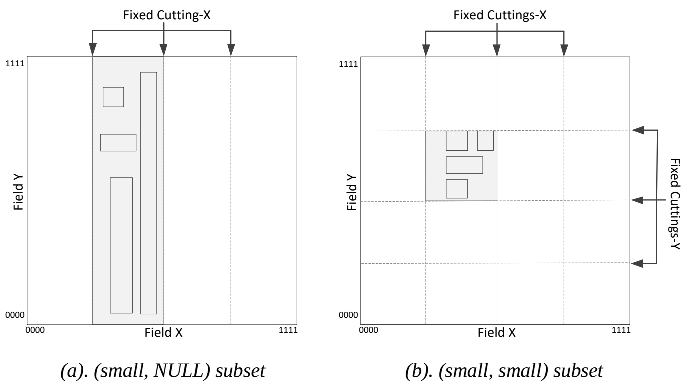


Figure 10. Simpler and more space-efficient Pre-Cuttings on *small fields*

areas respectively and resort to the next stage methods. It is easy to see that *rule replication* cannot appear in the pre-cutting stage, as all cuttings are conducted in *small fields* and stop at the threshold of sub-spaces.

Stage 2: ***Explicit Post-Splitting***: After the first stage cuttings, the searching space has been separated into much smaller sub-spaces, where each sub-space contains much fewer rules compared with the original rule set. Table V in Section V shows the number of rules after pre-cuttings for 12 rule sets containing 100K rules, it is clear that the number of rules in sub-space is quite small after pre-cuttings. Thus, based on our above review and analysis, HyperSplit can be well applied for the following decision-tree construction.

V. Experimental Results

In this section, we present the performance results of CutSplit with other representative decision-tree techniques: HyperCuts, HyperSplit, EffiCuts, HybridCuts and SmartSplit. We are very grateful to the authors of these algorithms, their open source codes and selfless personal help enable us to make a fair and justifiable comparison. As a response, our implementation of CutSplit is also publicly available in [40]. Based on key metrics in Section III, we evaluate our algorithm from *memory consumption*, *memory access* and *pre-processing time* respectively. All experiments are run on a machine with AMD A8-5600K CPU@3.60GHz and 8G DRAM. The operation system is Ubuntu 14.04.

A. *Memory Consumption*

Figure 11 shows the memory consumption of data structure for different decision-trees. It is clear that the memory consumption of HyperCuts and HyperSplit increase dramatically with the size of rule sets. For other four algorithms using rule set partitioning, the memory consumption scale well with the size of rule sets. This scalability is also stay for larger rule sets in Figure 13. As shown in Figure 13, CutSplit achieves a significant memory reduction compared with EffiCuts, ranging from 3 times to 20 times, with an average reduction of over 10 times. Compared with HybridCuts and SmartSlit, CutSplit also reduces memory consumption from 20% to 90%, with an average reduction of 65% and 50% respectively.

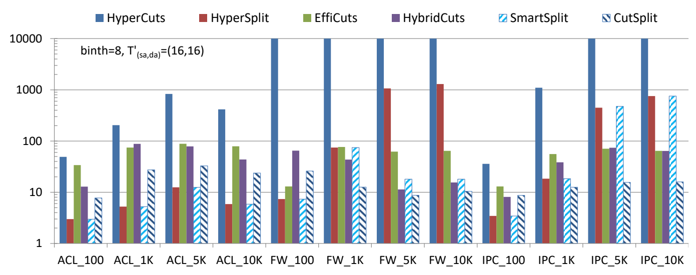


Figure 11. Memory consumption pre rule for small rule sets (Byte/rule)

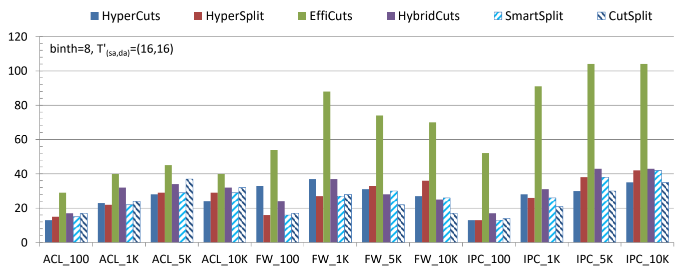


Figure 12. Memory access for small rule sets

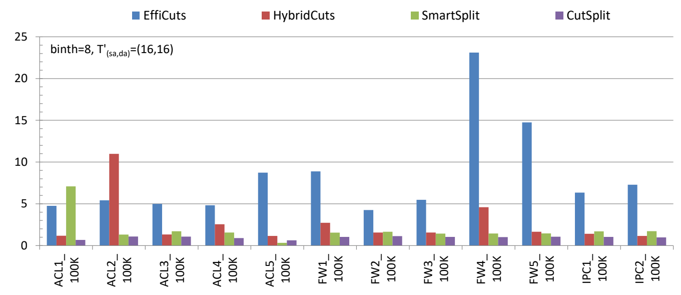


Figure 13. Memory consumption for larger rule sets (MB)

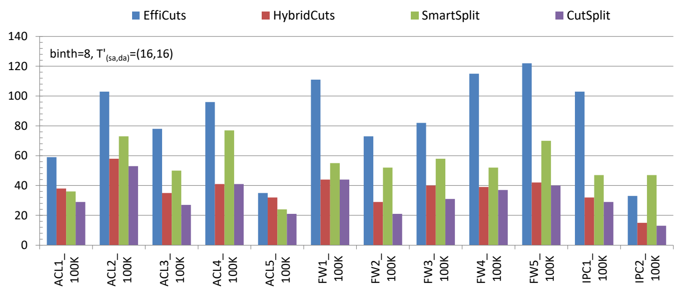


Figure 14. Memory access for larger rule sets

B. *Memory Access*

Figure 12 and Figure 14 show the performance in terms of the number of memory access for different algorithms. It is clear that CutSplit performs much better than other algorithms. As shown in Figure 14, CutSplit achieves an average of 3x speed-up compared with EffiCuts. Compared with HybridCuts and SmartSlit, CutSplit also achieves an average of 0.2x and 0.6x speed-up respectively. In addition, since the worst-case memory access can be guaranteed both in Pre-Cutting stage and Post-Splitting stage, the worst-case performance for CutSplit can be bounded.

Table IV. Pre-processing time for decision-tree construction (s)

Rule set	**EffiCuts**	**HybridCuts**	**SmartSplit**	**CutSplit**
ACL1_100K	4784.4	183.1	632.5	11.7
ACL2_100K	8338.4	91.0	427.4	4.1
ACL3_100K	8453.6	148.6	6403.7	2.6
ACL4_100K	8232.6	161.8	3336.1	3.4
ACL5_100K	8905.3	138.5	2695.9	3.0
FW1_100K	4250.7	165.1	1392.1	3.0
FW2_100K	2842.2	161.9	1652.9	2.5
FW3_100K	4281.2	187.8	3855.4	3.0
FW4_100K	1662.1	280.3	4553.6	3.5
FW5_100K	3778.4	179.2	3212.7	2.7
IPC1_100K	8615.0	151.5	3133.4	2.6
IPC2_100K	6070.4	229.6	3187.9	2.6
MEAN	5851	173	2874	3.7

Table V. More details about splitting based sub-trees in CutSplit

Rule set	**Number of rules**		**Pre-processing time (us)**	
	Worst-case	***Average***	***Worst-case***	***Average***
ACL1_100K	344	17.1	569	45.6
ACL2_100K	473	25.3	6975	125.1
ACL3_100K	31	10.4	207	21.3
ACL4_100K	320	18.7	8693	168.7
ACL5_100K	93	12.8	683	28.9
FW1_100K	193	16.4	2664	71.8
FW2_100K	10	9.4	28	14.8
FW3_100K	118	14.2	1068	43.2
FW4_100K	10	9.0	23	12.9
FW5_100K	111	14.4	869	38.5
IPC1_100K	14	9.7	57	18.1
IPC2_100K	10	9.6	129	15.1
MEAN	144	14	1830	50

C. *Pre-processing time*

1) Pre-processing time for decision-tree construction

Pre-processing time of EffiCuts, HybridCuts, SmartSplit and CutSplit are shown in Table IV. Compared to EffiCuts and SmartSplit, CutSplit runs about 1000 times faster when handling the same rule sets on average. Compared to HybridCuts, CutSplit also runs about 50 times faster on average. For most rule sets with 100K rules, CutSplit can complete decision-tree construction in few seconds, while most existing algorithms usually takes several hours. Thus, CutSplit is more practical for online updating.

2) More details about splitting based sub-trees in CutSplit

As *rule replications* only appear in post-splitting stage, incremental rule updates can be easily implemented in CutSplit. Take Figure 10 as an example. If we need to delete a rule from gray areas, only four rules in gray areas may be influenced, thus, we can simply apply HyperSplit to rebuild a sub-tree for remaining three rules. In contrast, if we need to add a rule to the rule set, we just need to identify its *small fields* and put it into a right sub-space. Thus, the most time consuming part of incremental rule update for CutSplit is the post-splitting stage. We evaluate all sub-trees constructed by HyperSplit and the results are given in Table V. As shown in this table, the average number of influenced rule is very small (i.e., 144 for 100K rules) and the average pre-processing time for sub-trees reconstruction is 50us.

VI. Conclusion

In this paper, we first conduct some in-depth review on issues that led to the unsuccess of the major decision-tree techniques for scalable packet classifications. With the insights obtained, we propose a practical framework called CutSplit, which can exploit the benefits of cutting and splitting techniques adaptively. By addressing the central problem caused by uncontrollable *rule replications* suffered by the major efforts, CutSplit not only pushes the performance of algorithmic packet classification more closely to hardware-based solutions, but also reduces the memory consumption to a practical level. Moreover, CutSplit achieves low pre-processing time for rule updates, a problem that has been ignored by most previous decision-trees. Compared with EffiCuts, experimental results show that, CutSplit achieves a memory reduction over 10 times, as well as 3x improvement on performance in terms of the number of memory access on average.

Acknowledgment

We appreciate the help from Balajee Vamanan and Jun Li in terms of technical realization. We would also like to thank Ru Jia, Qiang Su and Kaixuan Xing for kindly accepting to run several simulations. This work is supported in part by the National Keystone R&D Program of China (No. 2017YFB0803204 & 2016YFB0800101), by Natural Science Foundation of China (NSFC) (No. 61671001 & 61725206), by Guangdong Key Program (GD2016B030305005) and by Shenzhen Research Programs (JSGG20150331101736052, ZDSYS201603311739428, JCYJ20170306092030521 & JCYJ20150629144717142).

References

[1] N. McKeown, et al., "*OpenFlow: enabling innovation in campus networks*," in ACM SIGCOMM, 2008.

[2] D. E. Taylor, "*Survey and Taxonomy of Packet Classification Techniques*," ACM Computing Surveys, 37(3):238-275, 2005.

[3] H. J. Chao and B. Liu, "*High performance switches and routers*," John Wiley & Sons, 2007.

[4] C. R. Meiners, A. X. Liu and E. Torng, "*Hardware Based Packet Classification for High Speed Internet Routers*," Springer, 2010.

[5] A. X. Liu, C. R. Meiners and Y. Zhou, "*All-match based complete redundancy removal for packet classifiers in TCAMs*," in IEEE INFOCOM, 2008.

[6] 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, 18(2):490-500, 2010.

[7] R. Wei, Y. Xu and H. J. Chao, "*Block Permutations in Boolean Space to Minimize TCAM for Packet Classification*," in IEEE INFOCOM, 2012.

[8] A. X. Liu, C. R. Meiners and E. Torng, "*Packet classification using binary Content Addressable Memory*," in IEEE INFOCOM, 2014.

[9] H. Liu, "*Efficient mapping of range classifier into ternary-CAM*," in IEEE Hot Interconnects, 2002.

[10] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "*Algorithms for advanced packet classification with ternary CAMs*," in ACM SIGCOMM, 2005.

[11] H. Che, Z. Wang, K. Zheng and B. Liu, "*DRES: Dynamic range encoding scheme for TCAM coprocessors*," IEEE Transactions on Computers, 57(7):902-915, 2008.

[12] A. Bremler-Barr and D. Hendler, "*Space-efficient TCAM-based Classification Using Gray Coding*," in IEEE INFOCOM, 2007.

[13] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan and E. Porat, "*On finding an optimal TCAM encoding scheme for packet classification*," in IEEE INFOCOM, 2013.

[14] E. Spitznagel, D. E. Taylor and J. Turner, "*Packet classification using extended TCAMs*," in IEEE ICNP, 2003.

[15] Y. Ma and S. Banerjee, "*A Smart Pre-Classifier to Reduce Power Consumption of TCAMs for Multi-dimensional Packet Classification*," in ACM SIGCOMM, 2012.

[16] B.Vamanan and T.Vijaykumar, "*TreeCAM: Decoupling Updates and Lookups in Packet Classification*," in ACM CoNEXT, 2011.

[17] B. Pfaff, et al., "*The Design and Implementation of Open vSwitch*," in USENIX NSDI, 2015.

[18] S. Yingchareonthawornchai, J. Daly, A. X. Liu and E. Torng, "*A sorted partitioning approach to high-speed and fast-update OpenFlow classification*," in IEEE ICNP, 2016.

[19] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane and P. Eugster "*SAX-PAC (Scalable And eXpressive PAcket Classification)*," in ACM SIGCOMM, 2014.

[20] C. L. Hsieh and N. Weng, "*Many-field packet classification for software-defined networking switches*," in ACM/IEEE ANCS, 2016.

[21] X. Li and Y. Lin, "*TaPaC: A TCAM-Assisted Algorithmic Packet Classification with Bounded Worst-Case Performance*," in IEEE GLOBECOM, 2016.

[22] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "*EffiCuts: Optimizing Packet Classification for Memory and Throughput*," in ACM SIGCOMM, 2010.

[23] J. Fong, X. Wang, Y. Qi, J. Li and W. Jiang, "*ParaSplit: A Scalable Architecture on FPGA for Terabit Packet Classification*," in IEEE Hot Interconnects, 2012.

[24] W. Li and X. Li, "*HybridCuts: A scheme combining decomposition and cutting for packet classification*," in IEEE Hot Interconnects, 2013.

[25] P. He, G. Xie, K. Salamatian and L. Mathy, "*Meta-Algorithms for Software based Packet Classification*," in IEEE ICNP, 2014.

[26] W. Jiang and V. K. Prasanna, "*Large-scale wire-speed packet classification on FPGAs*," in ACM/SIGDA FPGA, 2009.

[27] P. Gupta and N. McKeown, "*Packet Classification using Hierarchical Intelligent Cuttings*," in IEEE Hot Interconnects, 1999.

[28] S. Singh, F. Baboescu, G. Varghese and J. Wang, "*Packet Classification using Multidimensional Cutting*," in ACM SIGCOMM, 2003.

[29] Y. Qi, L. Xu, B. Yang, Y. Xue and J. Li, "*Packet Classification Algorithms: From Theory to Practice*," in IEEE INFOCOM, 2009.

[30] D. E. Taylor and J. S. Turner, "*Classbench: A Packet Classification Benchmark*," in IEEE INFOCOM, 2005.

[31] V. Srinivasan, G. Varghese, S. Suri and M. Waldvogel, "*Fast and Scalable Layer Four Switching*," in ACM SIGCOMM, 1998.

[32] T. V. Lakshman and D. Stiliadis, "*High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching*," in ACM SIGCOMM, 1998.

[33] P. Gupta and N. McKeown, "*Packet Classification on Multiple Fields*," in ACM SIGCOMM, 1999.

[34] D. E. Tayor and J. S. Turner, "*Scalable Packet Classification using Distributed Crossproducting of Field Labels*," in IEEE INFOCOM, 2005.

[35] V. Srinivasan, S. Suri and G. Varghese, "*Packet Classification using Tuple Space Search*," in ACM SIGCOMM, 1999.

[36] J. Daly and E. Torng, "*TupleMerge: Building Online Packet Classifiers by Omitting Bits*," in IEEE ICCCN, 2017.

[37] M. Varvello, R. Laufer, F. Zhang and T. V. Lakshman, "*Multi-Layer Packet Classification with Graphics Processing Units*," in ACM CoNEXT, 2014.

[38] F. Baboescu, S. Singh and G. Varghese, "*Packet Packet Classification for Core Routers: Is there an alternative to CAMs?*" in IEEE INFOCOM, 2003.

[39] T. Yang, A. X. Liu, Y. Shen Q. Fu, D. Li and X. Li, "*Fast OpenFlow Table Lookup with Fast Update*," in IEEE INFOCOM, 2018.

[40] http://www.wenjunli.com/CutSplit/.