Meta-algorithms for Software-based Packet Classification

Peng He*†, Gaogang Xie*, Kavé Salamatian‡, Laurent Mathy§
*ICT, CAS, China, †University of CAS, China, ‡University of Savoie, France, §University of Liége, Belgium
{hepeng, xie}@ict.ac.cn, kave.salamatian@univ-savoie.fr, laurent.mathy@ulg.ac.be

***Abstract*—We observe that a same ruleset can induce very different memory requirement, as well as varying classification performance, when using various well known decision tree based packet classification algorithms. Worse, two similar rulesets, in terms of types and number of rules, can give rise to widely differing performance behaviour for a same classification algorithms. We identify the intrinsic characteristics of rulesets that yield such performance differences, allowing us to understand and predict the performance behaviour of a ruleset for various modern packet classification algorithms. Indeed, from our observations, we are able to derive a memory consumption model and an offline algorithm capable of quickly identifying which packet classification is suited to a give ruleset. By splitting a large ruleset in several subsets and using different packet classification algorithms for different subsets, our *SmartSplit* algorithm is shown to be capable of configuring a multi-component packet classification system that exhibits up to 11 times less memory consumption, as well as up to about 4× faster classification speed, than the state-of-art work [20] for large rulesets. Our *AutoPC* framework obtains further performance gain by avoiding splitting large rulesets if the memory size of the built decision tree is shown by the memory consumption model to be small.**

I. INTRODUCTION

Packet classification is a key component for network devices providing advanced network services. While the area of packet classification has been extensively researched for over two decades, the continual growth in both traffic volumes and classifier sizes are proving as challenging as ever. Indeed, traditional network applications and services, such as firewalling, IDS, VPNs, amongst others, require such fine classification of traffic that it is not uncommon to find classification rulesets in excess of 15K rules in the wild [8]. Furthermore, the trends are exacerbated by the emergence of large, multi-tenant data centers which require access control, load balancing, bandwidth sharing, traffic monitoring *etc.* between tens of thousand of virtual machines [10].

Current packet classification solutions can be categorized into two types: TCAM-based and RAM-based. RAM-based solutions, also known as algorithmic solutions, build compact and efficient data structures for packet classification, yielding cheap, software-based solutions. In contrast, TCAMs guarantee deterministic, very high performance, so these expensive and power-hungry devices are still the *de facto* standard solution adopted in network devices. Nevertheless, the recent advent of Software-Define Networking (SDN), enabling on-demand network infrastructure deployment based on virtual network appliances in the cloud [14], does call for high-performance software-based packet classification solutions, while the rate of growth in ruleset size may well, in some contexts such as the cloud, outpace TCAM capacity evolution.

/978-1-4799-6204-4/14$31.00 ©2014 IEEE

Algorithm	Ruleset(size)	Memory size	Mem. accesses
HyperSplit	ACL1_100K	2.12MB	32
	ACL2_100K	83MB	43
EffiCuts	ACL1_100K	3.23MB	65
	ACL2_100K	4.81MB	136

TABLE I: Performance comparison on different rulesets

RAM based solutions are plagued by the tradeoff between memory size and lookup speed. Theoretical bounds states that for checking N general rules in K dimensions ($K > 3$), the best algorithm have either $\mathcal{O}(\log N)$ search speed at the cost of $\mathcal{O}(N^K)$ space, or $\mathcal{O}(\log^{K-1} N))$ search time at the cost of $\mathcal{O}(N)$ space [11]. This means that, in general, RAM-based solutions need large memory for fast classification speed. Fortunately, Gupta and McKeown reported in [5] that real rulesets have more *structures*, which can be exploited by the algorithms to achieve fast classification speed with small memory footprint. However, different algorithms are based on different observations of rule structures. When applying algorithms on different rulesets, a performance *unpredictability* problem may occur. To illustrate this issue, we present in Table I the performance in terms of memory size and maximum number of memory accesses[1] of two state-of-art algorithms (HyperSplit[2] [13] and EffiCuts [20]) on two ACL rulesets. We can see that for two similar firewall rulesets, ACL1_100K and ACL2_100K, containing nearly equal number of rules, the memory size needed by HyperSplit for ACL1_100K is around 40 times larger than ACL2_100K (from 2.12MB to 83MB). While the memory requirement of EffiCuts on ACL1_100K and ACL2_100K are nearly equal and small, the maximum number of memory accesses needed by EffiCuts on ACL1_-100K is 2 times that of HyperSplit.

These wide variations in performance demonstrate how crucial applying, in practice, the "right" algorithm to a given ruleset, actually is. For example, recent CPUs usually contain several Mbytes of last level cache and several GBytes of DRAM. Therefore, the memory size of HyperSplit algorithm on ACL1_100K can fit in the CPU's cache but the memory size for ACL2_100K cannot. Generally accessing a data in the external DRAM requires around 50 nanoseconds, while accessing a data in cache requires only $1 \sim 5$ nanoseconds,

[1]The number of memory accesses is the limiting factor, and thus a direct indicator, of classification speed.

[2]Here, we use an improved HyperSplit implementation, see Section VII for details.

meaning that one should use HyperSplit algorithm on ACL1_-100K for smaller memory size and fewer memory accesses, but should use EffiCuts on ACL2_100K to trade more memory accesses for fewer memory access latency. In general, for a given ruleset, we need to select an "right" algorithm for the memory size and the number of memory accesses trade-off.

A straightforward method to solve the problem would be to implement various algorithms on a given ruleset and choose the one with best performance results. However, packet processing platforms are often resource-constrained, and such comparison is sometimes very time consuming (*e.g.* The HiCuts [6] algorithm may need over 24 hours to process some large rulesets [13]), making this approach at best impractical, and at worst infeasible in more dynamic environments, such as OpenFlow-based networks or virtual data centers, where rulesets may change much faster than this processing time.

In our work, we therefore seek to understand the reasons behind these observed temporal and spacial performance variations, with a view to quickly identify the "right" classification algorithm for a given subset. In Section II, we analyze the characteristics of rulesets that do have a primary bearing on both the memory footprint and classification speed and we review three of the main state-of-the-art packet classification algorithms.

As the memory footprint of the ruleset for a given algorithm is an important factor, we present in section III a memory consumption model, to be used as a fast memory size checker, which helps to select for best memory-performance tradeoff.

In Section IV, we describe an offline recommendation algorithm that analyses rulesets for the above mentioned characteristics, and recommends algorithms for the given ruleset, based on classification performance alone. With this analysis tool, we present in Section IV a new multi-tree algorithm SmartSplit. The SmartSplit algorithm is built on recent work [20] that showed how to trade classification performance for much reduction in memory consumption by splitting the ruleset into several subsets and classifying against these subsets in sequence. However, going beyond [20] which uses HyperCuts [15] on every subset, SmartSplit seeks to maximize classification speed, while meeting overall memory consumption constraints, by using different classification algorithms for the stages of the classification sequence (e.g. for the various sub-rulesets). We also present a packet classification framework AutoPC in Section V. The AutoPC framework, which is based on the memory consumption model, tries to further improve the performance by avoiding ruleset splitting if the memory size of rulesets is shown to be small.

Sections VI and VII present our evaluation methodology and experimental results, respectively. Section VIII summaries the related work, and Section IX concludes the paper.

II. Background and Observations

We first give a brief review of factors explaining why the performance of packet classification algorithms can exhibit wide variations from one ruleset to another. More detailed explanations are available in [12]. A packet classification ruleset can be considered as a collection of *ranges* defined on different fields. Table II shows an example of classification ruleset containing 6 rules defined over two 4-bit fields, where "*" represents a "don't care" value. This ruleset can be translated into four distinct *ranges* on Field1: $[14, 15]$ (defined by rule R1), $[12, 13]$ (R2), $[4, 7]$ (R5), $[0, 15]$ (all rules); and four on Field 2: $[4, 5]$ (R3), $[6, 7]$ (R4), $[8, 11]$ (R5), $[0, 15]$ (all rules).

TABLE II: An example ruleset

Rule #	Field 1	Field 2	Action
R1	111*	*	DROP
R2	110*	*	PERMIT
R3	*	010*	DROP
R4	*	011*	PERMIT
R5	01**	10**	DROP
R6	*	*	PERMIT

A packet classification ruleset has a simple geometric interpretation: packet classification rules defined on K fields can be viewed as defining *K-orthotope*, *i.e.* hyper-rectangle in the K-dimensional space, and rulesets define intricate and overlapping patterns of such orthotopes. For example, rule R1 in Table II defines a rectangular band over the two dimensional space of features, where the short side is 2 units long (from 14 to 15, along the axis defined by Field 1), and the long side spans the whole range of the second dimension. This structure results from the wildcard existing on the second dimension field that generates a *large range*. Similarly, rule R3 defines another rectangular region but with the short side along the second dimension.

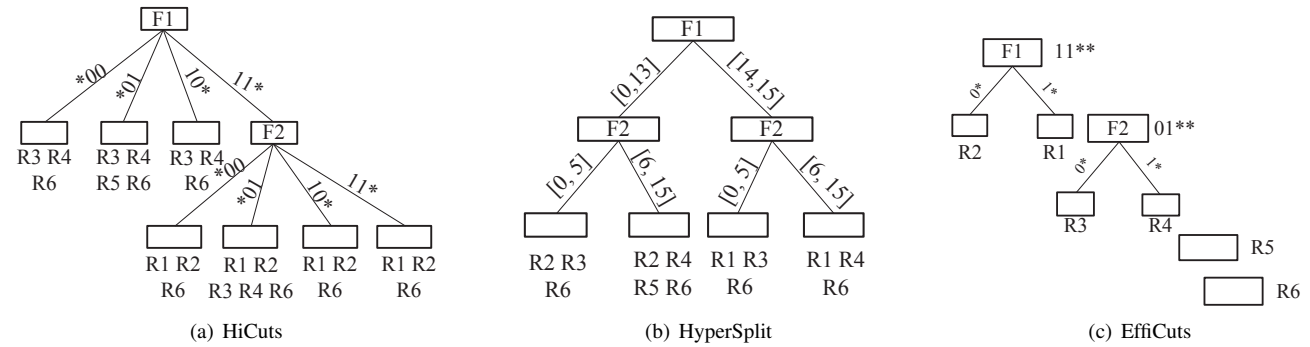
A. Influence on temporal performance

A node in a packet classification decision tree (DT) can be considered as making a spatial partition of the geometric space into non-overlapping parts. The aim of a DT is to partition the space of features into regions that will hopefully contain the smallest number rules. Different classification algorithms apply different heuristics for dividing the space. In particular two types of partitioning is applicable. The first type is the "cut", that consists of dividing a given range into multiple equal-sized intervals, the second type is a "split", consisting in dividing an interval at a split point into two sub-intervals, a right and a left one.

At first glance the cut-based division seems more efficient than the split-based one. Indeed, when ranges have roughly similar sizes and are uniformly distributed along a dimension, equal-sized cuts can be very efficient at separating those ranges. However, ranges observed in practice are sometimes non-uniformly distributed (*e.g.* dissimilar and/or in clusters along the dimension), in which case applying equal-sized cuts will become inefficient as either some cuts will simply split a rule in regions of the space where this rule has already been isolated, and/or deeper (i.e. finer-grained) cuts will be necessary in other regions, to isolate clustered rules. Under such conditions, the resulting DT would be skewed, with some branches significantly longer than others. We need to evaluate the uniformity of ranges before applying cuts or split.

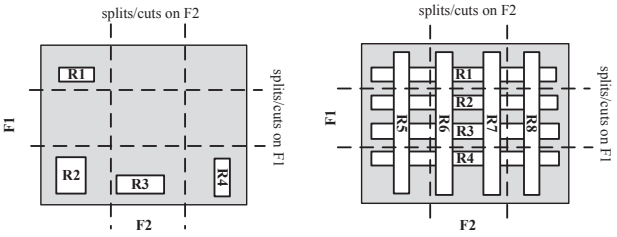
B. Influence on spacial performance

In real-world rulesets, some specific patterns are commonly encountered that can have a bearing on the efficiency of the



(a) HiCuts (b) HyperSplit (c) EffiCuts

Fig. 1: Decision trees built by different algorithms



(a) sparse rules (b) orthogonal structure rules

Fig. 2: Geometric View of Packet Classification Rules

corresponding DT. Such patterns include: orthogonal structures like that resulting from rules R1, R2, R3, R4 (a more general case is show in Figure 2(b)), and sparse structures like the one defined by rule R5 (more general case is shown in Figure 2).

A major problem occurs when orthogonal structures are present in the ruleset. In this case, rules cannot be completely separated into regions containing a single rule with hyperplane divisions, and the best that can be achieved is to use divisions, forming $\mathcal{O}(N^K)$ regions containing K orthogonal rules, where N is the number of orthogonal rules and K is the dimension of the feature space. Moreover, each division is likely to intersect with $\mathcal{O}(N)$ other rules' subregions. When this happens, each rule that is cut has to be duplicated in the DT nodes as the cut does not separates these rules, *i.e.* rules with orthogonal structure will cause a large amount of rule duplication in Decision Tree based algorithms, creating large memory footprints.

On the other hand when the rule structure is sparse, $\mathcal{O}(N)$ spatial divisions can isolate each rule without cutting through other rules, yielding modest memory requirements.

C. Application to existing algorithms

We briefly describe three major packet classification algorithms proposed in the literature – HiCuts, HyperSplit and EffiCuts – and identify the specific factors that negatively impact their performance. For illustration purposes, three decision trees built on the example ruleset using three algorithms are shown in Figure 1 .

1) HiCuts and HyperCuts: We first describe HiCuts [6] and HyperCuts [15], two closely related and classical DT based algorithms. The two algorithms work essentially by cutting the full range of each dimension of the multi-dimensional feature space into equal-size intervals. In Figure 1(a), we show the decision tree generated by HiCuts algorithm where the Field 1 is cut into 4 equal-sized sub-spaces: $[0, 3]$, $[4, 7]$, $[8, 11]$, $[12, 15]$, and Field 2 is further cut into 4 equal-sized sub-spaces. HiCut suffers from a combination of the previous described issues. On one hand as the distribution of ranges is non-uniform, *e.g.*, the ranges in Table II leaves 50% of the full range $[0, 15]$ uncovered, *equal-sized cutting* becomes inefficient as several cuts are spurious. Moreover as orthogonal rules are present, each spurious cuts, which intersects with orthogonal rules result in rule duplication in several leaves of the decision tree. As empirically up to 90% of memory footprint of a built DT is consumed by pointers pointing to rules, rules duplication increases the memory footprint significantly.

HyperCuts which extends HiCuts by allowing to cut multiple fields in each node of the tree, suffers from the same issues caused by the inefficiency of equal-sized cuts when there are non-uniform ranges.

2) HyperSplit: In order to overcome the non-uniformity of range coverage described earlier, HyperSplit [13] adopts a different method to separate rules. It splits the chosen field into unequal ranges that contain nearly equal number of rules, *e.g.*, in Figure 1(b) the Field 1 is split into two unequal size intervals: $[0, 13]$ and $[14, 15]$, which separate R1 and R2 using a single memory access. In order to minimize the number of comparison, HyperSplit implements a binary tree, *i.e.*, each node contains only one split point splitting the given range into two regions.

By using *unequal-sized splitting*, HyperSplit avoids unneeded cuts reducing the memory footprint. However the main source of redundancy remains because splits intersect with orthogonal rules. Moreover, the binary tree structure adopted by HyperSplit increases the tree depth, resulting in more memory accesses than HiCuts and HyperCuts.

3) EffiCuts: Instead of building a single decision tree for all the rules, EffiCuts [20] builds multiple trees for one ruleset. To do so, EffiCuts categorizes the ruleset-defined ranges into *small* and *large* ranges. A range is labelled as *large* if it covers a large enough proportion, determined by a threshold of the full range. Otherwise, this the range is labelled as *small*. The

Field 1	Field 2	Field 3
00*	*	01
01*	01	*
10*	*	10
11*	10	*

TABLE III: Ruleset with a lot of distinct $small$ ranges on Field 1

threshold is set as 0.50 for most of fields. The ruleset shown in Table II has one *large range*: $[0, 15]$ and three *small ranges*: $[14, 15]$, $[12, 13]$ and $[4, 7]$ on Field 1. Based on this labeling one can classify each rule in the ruleset into at most 2^K categories in $\{small, large\}^K$ for a K dimensional classifier. For example, for the ruleset in Table II, R1 and R2 are classified as $(small, large)$, R3 and R4 as $(large, small)$, R5 as $(small, small)$ and R6 as $(large, large)$. EffiCuts builds separate decision trees for rules in each category. We show in Figure 1(c), the resulting decision trees.

By putting rules with the $large$ label on different fields in separate decision trees rules, EffiCuts untangles existing "orthogonal structures" and remove completely the induced rule duplication. This results in a dramatic reduction of the memory size compared to HiCuts and HyperCuts. However, one need to traverse all trees in order to find the most specific match, resulting in a large number of memory accesses and this reduces significantly the throughput [20].

D. Discussions

The above description of different packet classifications gives insight for understanding classification performance issues. Using the geometrical view, we observed the major impact of "orthogonal structures" and the non-uniformity of range sizes on memory footprint and on the performance. A noteworthy case happens when a ruleset contains only $small$ ranges in at least one of its dimension, like the ruleset in Table III. For such cases one can separate all the rules, using a decision tree working only on the dimension with only $small$ ranges, as the cuts/splits on this dimension will not intersect any "orthogonal structures" happening in other dimensions. In this cases, using EffiCuts that would generate two trees for the two categories $(small, small, large)$ and $(small, large, small)$, will be inefficient. The above observations, and the fact that all in all the main issue is to be able to separate subregions with a small number of memory accesses, drive us to propose these guidelines:

1) "Orthogonal structures" should be considered, and rules should be eventually splitted in order to untangle these structure and avoid memory explosion.
2) When splitting a ruleset, if a dimension appears that contain only $small$ ranges, it should be used to separate the rules with a single tree.
3) Equal-sized cutting becomes more efficient when ruleset ranges are uniform, if not splitting with non-equal sized intervals should be considered.

Indeed, these obvious observations, cannot be used by a network operator if the structure of the ruleset is not analyzed. We therefore propose and evaluate methods and algorithms that analyze rulesets in order to extract metrics that will help in deciding the best packet classifier for a given ruleset.

III. Memory footprint estimation

Given a ruleset, the first concern is whether the size of built DT can fit in the available memory (CPU cache). As we saw in Section II-B, orthogonal structures within the ruleset are a major cause of large memory requirements. We have therefore to characterize these orthogonal structures in order to estimate the DT memory footprint. The goal here is not derive a precise estimation of the memory footprint, it is to use rulesets features in order to achieve a rough estimate which gives an order of magnitude of the size.

We will adopt the ruleset portioning into 2^K categories in $\{small, large\}^K$ described previously in EffiCuts [20]. As in practice, $50\% \sim 90\%$ of the cuts or splits are performed on the IP source and destination fields [16], we will first concentrate on these two dimensions and ignore others, without losing much in estimation accuracy. We therefore analyze orthogonal structures involving only the IP source and destination fields, and label rules as $(small, small)$, $(large, small)$, $(small, large)$ or $(large, large)$ based on these fields. The number of rule in each category is denoted respectively as ss, ls, sl, and ll respectively.

To simplify, for the time being, we will assume that large range rules cover the whole span of the associated dimension, *i.e.*, the corresponding IP address range is a wildcard. This will result in overestimation of the memory footprint which we will address in the next section. We also denote the number of distinct ranges on the source and destination IP fields as us and ud. These two values can be calculated by a simple scan of the ruleset. Let $\alpha = \frac{us}{us+ud}$ be the proportion of distinct source IP ranges.

The $(small, small)$ rules can be easily separated by either using source or destination IP ranges. We assume that they are separated by source or destination IP field without duplication and in proportion to α and $1 - \alpha$. The memory needed to separating these $(small, small)$ rules is therefore $M_{ss} = ((1 - \alpha) \times ss + \alpha \times ss) \times PTR = ss \times PTR$, where PTR is the size of a pointer (pointing to a rule).

Orthogonal structures are created by $(small, large)$ and $(large, small)$ rules. When isolating the small range side of any of these rules (i.e. when cutting in the direction of the dimension of their large range), all large ranges along the other dimension are cut, resulting in the need to duplicate the corresponding rules on either side of the cut. For instance, all the cuts (or splits) on source IP field, to separate every $(small, large)$ rules, will duplicate all $(large, small)$ rules, generating ls duplicated $(large, small)$ rules, and similarly for each $(large, small)$ rule, there will be sl duplicated $(small, large)$ rules.

Furthermore, the $ss \times \alpha$ rules labelled $(small, small)$ that have been separated using the source IP ranges, will also duplicate each $(large, small)$ rule, and similarly the $ss \times (1 - \alpha)$ rules labelled $(small, small)$, separated using the destination IP ranges, will duplicate each $(small, large)$ rule.

Overall, the upper bound on the number of duplication of $(large, small)$ rules is thus $ls \times (sl + ss \times \alpha)$, while that for the duplication of $(small, large)$ rules is $sl \times (ls + ss \times (1 - \alpha))$. However, in practice DT algorithms stop building the DT when there is at most a given threshold number, $binth$, of rules in any leave. This means that the number of duplicates are over-estimated by a factor of $\frac{binth}{2}$ (2 rules per leaves .vs. $binth$ rules per leaves) yielding:

$$M_{ls} = ls \times \frac{sl + ss \times \alpha}{binth/2} \times PTR \quad (1)$$

$$M_{sl} = sl \times \frac{ls + ss \times (1 - \alpha)}{binth/2} \times PTR \quad (2)$$

The last category of rules, the $(large, large)$ one, will get duplicated either by splitting of cutting on source or destination IP fields. The $(large, large)$ rules need therefore a memory size:

$$M_{ll} = ll \times \frac{sl + ss \times \alpha}{binth/2} \times \frac{ls + ss \times (1 - \alpha)}{binth/2} \times PTR \quad (3)$$

The the total memory size is finally estimated as the sum of the four elements: $M = M_{ss} + M_{ls} + M_{sl} + M_{ll}$.

A. Improving memory size estimation

The assumption that all orthogonal rules are duplicated over-estimates the memory requirement, as some large ranges might not cover the full range and therefore might not be duplicated in all cases. Nonetheless, if we partition the feature space into smaller subspace the assumption is more likely to hold as the "full range" in a subspace is necessarily smaller or equal to the full range in the "full space".

So in order to reduce the over-estimation and improve the quality of the memory footprint estimation, we first divide the large feature space into n equal-sized rectangular sub-space, and apply the memory estimation model to each one of these subs-space separately. We will illustrate this with the ruleset example in Figure 3. In the initial memory estimate, R1 and R4 are considered as $(large, small)$ rules, and Cut 2 is supposed to cause duplication of R1 and R4. However, as the R1 and R4 are not wide enough, they are not duplicated by Cut 2. After dividing the space into sub-space, we can witness that any cut on the source IP field in sub-space A (*.resp.* C) will surely cause the duplication of R1 and R4, but not in subspace B. This therefore improves the memory footprint estimation.

It is noteworthy that in the process of dividing the space into sub-spaces, some $(large, large)$ rules may become fully covered by more specific and higher priority rules in this sub-space. These redundant rules must be removed before calculating parameters ll, ls, sl and ss of the the orthogonal structure in the subspace.

B. Limitations

The assumption that all the splits are performed *only* on IP fields is also a source of the memory size over-estimation, as splitting or cutting on other dimension can reduce the impact of orthogonal structure (see Section II-D).

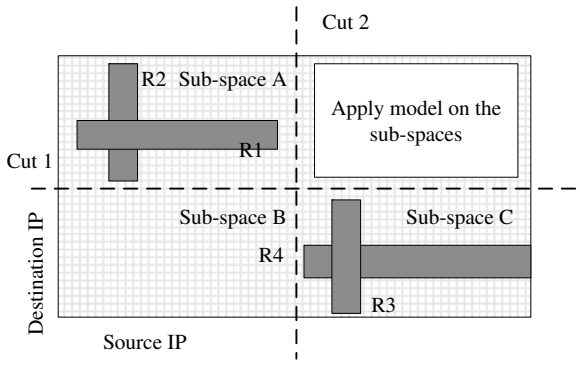


Fig. 3: Improved Memory Size model

However the main aim of the calculation in this section is to obtain a rough estimate giving an order of magnitude of the memory footprint. We will show in Section VII, software based packet classification performances are not sensitive to the precise memory size but roughly to its order of magnitude. Our memory footprint estimation can therefore be used as a fast memory size checker, especially for large rulesets.

A last limitation of the model is that we assume that we can separate N rules with N cuts/splits. While this is usually correct for splits, this can be incorrect for cuts due to the inefficiency of equal-sized cutting over non-uniform rules. We expect therefore better estimates for HyperSplit than HiCuts/HyperCuts.

IV. Characterizing range distribution uniformity

As explained in the previous section the uniformity for small range distribution (we call it coverage uniformity for short) is an important factor for deciding to apply cuts or splits when building the decision tree. We show in Table IV the number of unique small ranges in large rulesets and observe that, the number of unique small ranges on IP fields is frequently comparable to the total number rules. Therefore, the rulesets can be separated *only* by the small ranges on IP fields and the uniformity of small ranges on IP fields is important for choosing cut or split. In the forthcoming, we will propose a simple variant of a centered interval tree [2] and characterize the coverage uniformity by computing shape metrics on such trees.

A. Interval tree

A centered interval tree [2] is a well-known tree used to efficiently represent intervals or ranges (in the context of packet classification). Each node of the interval tree is defined by a centre point which is used to separate ranges: The ranges completely to the left of the centre point (left ranges for short), those completely to the right of the centre point (right ranges), and those containing the centre point. The latter are then associated with the node itself (and removed from further consideration). A left sub-tree is then built using the left ranges and a right sub-tree is built using the right ranges. This procedure is repeated until all ranges have been associated with nodes in the tree.

Ruleset	unique src. IP small range	unique dst. IP small range	#src/rules(%)	#dst/rules
acl1_10K	4023	750	41%	7%
acl2_10K	6069	6527	64%	69%
acl3_10K	1017	1110	10%	11%
acl4_10K	918	1864	10%	19%
acl5_10K	371	1527	5%	21%
fw1_10K	3389	6665	36%	70%
fw2_10K	8309	3080	86%	32%
fw3_10K	2835	6209	31%	69%
fw4_10K	3884	6797	44%	76%
fw5_10K	3414	5327	39%	60%
ipc1_10K	1332	2768	14%	29%
ipc2_10K	4748	8923	47%	89%
acl1_100K	99053	236	99%	0.2%
acl2_100K	8315	8092	11%	11%
acl3_100K	85355	86603	86%	87%
acl4_100K	88434	32766	89%	33%
acl5_100K	43089	78952	43%	80%
fw1_100K	26976	66173	30%	74%
fw2_100K	81565	30602	85%	32%
fw3_100K	15960	62993	19%	75%
fw4_100K	38076	67073	45%	80%
fw5_100K	29786	54004	35%	64%
ipc1_100K	86210	90433	87%	91%
ipc2_100K	47228	89135	47%	89%

TABLE IV: the number of unique IP small ranges in large rulesets

While the original centered interval tree algorithm picks centre points to keep the tree as balanced as possible, we use a slightly different strategy to build a tree whose shape will reflect the degree of uniformity in the ranges. We start with the full range, its widest possible span, for the field under consideration and pick as centre point for the root node the middle of this range. We then use the left (resp. right) half range for the left (resp. right) child. Note that with this approach, the centre point in a node depends solely on the original full range and the position of the node in the tree. As in practice DT algorithms stop cuttings/splittings on nodes associated with less then $binth$ rules, we will stop the growth of our interval tree when the number of rules associated with a node containing less than $binth$ rules.

In the interval tree, the large ranges are likely to be "absorbed" by the nodes near to the root, while the small ranges are usually associated with leaf nodes. So the shape of interval trees actually represents the distribution of small ranges. The main insight into our method is that centre points correspond to equal-sized cuts of the original full range. And since a branch of the tree only growth if there are ranges on either side of the corresponding centre point, a balanced tree would indicate uniform coverage of ranges. In such a case, an algorithm using equal-sized cuts (e.g. HiCuts/HyperCuts) would very efficiently separate the ranges and these associated rules and produce a very fast classifier.

In fact, each node at the kth level of the tree, with the root being the level 0, covers a portion $\frac{1}{2^k}$ of the full range. These range portions can be efficiently represented by 2^k equal-sized cuts on the full range. Assume a node N resides in the kth level of the interval tree, rules intersecting with the range portion managed by N can be found by collecting associated rules in the path from the root to N. These intersected rules will be duplicated when performing $2^l, l > k$ equal-sized cuts on the full range. Since rules in nodes at the same level of the tree

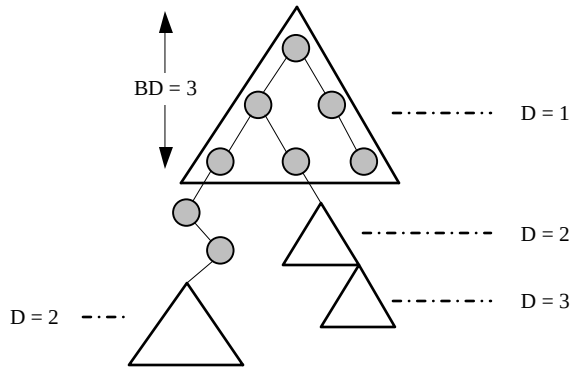


Fig. 4: Balanced Tree Distance and Balanced Tree Depth

are non-overlapping, a node is missing in this tree means that there is no rules on that side of the parent node, in which case, performing any cut in this interval would be useless (separate no rules but duplicate the intersected rules). This means that the interval tree structure gives interesting insights into the efficiency of using cuts. When the interval tree is balanced, or as will explained later quasi-balanced, it is meaningful to use cuts and there will be not any, or better said not too many, spurious cuts. If the interval tree is un-balanced, using splits will avoid these spurious cuts resulting in smaller duplicates.

However, a perfectly balanced interval tree may be too strict a condition to pick equal-sized cutting. We therefore define *quasi-balanced tree* as a tree where the following condition is verified at each level of the tree:

$$\frac{\#Nodes\ in\ the\ k^{th}\ level}{\#Nodes\ in\ the\ (k-1)^{th}\ level} \geq B_{ratio} \qquad (4)$$

As our interval tree is a binary tree, $B_{ratio} \in (0, 2]$. We will set $B_{ratio} = 1.5$ for a good approximation of balance for the tree. Note that since we set $B_{ratio} > 1$, a quasi-balanced tree contains at least 3 nodes, and the height of one quasi-balanced tree is at least 2. This is the reason why chains of isolated nodes do not belong to any quasi-balanced subtrees as in Figure 4.

B. Characterizing the shape of interval trees

In practice, interval trees built from rulesets are unbalanced, containing nodes with single child or even leaves at various levels in the tree. These nodes break the overall tree into several quasi-balanced subtrees (triangles) of different sizes (see Figure 4). In order to characterize these quasi-balanced subtrees, we define two for each node metrics: the *balanced depth* BD, the height of the quasi-balanced subtree the node belongs to, and *balance tree distance*, D, the number of quasi-balanced sub-trees between a given sub-tree and the top one.

The full interval tree is characterized by D_{max}, the maximum value of *balance tree distance*, and BD_{max}, the maximum *balance depth*, calculated over all quasi-balanced subtrees. When the range coverage is non-uniform, the interval tree contains many quasi-balanced sub-trees with small height values, and its D_{max} will be large. On the other hand, a small D_{max} value means a more uniform coverage.

C. Algorithm decision framework

In practice, we observed that small rulesets usually exhibits a non-uniform distribution of small ranges (non-uniform coverage), and therefore HyperSplit is suited to them. However, as the size of rulesets grows, the size of orthogonal structure, as well as the number of uniformly distributed ranges also grows. When our memory footprint model indicates that the size of the built DT is too large, one need to split the ruleset into sub-rulesets and build a single DT for each set. However, due to the probable existence of the "coverage uniformity" in some of the subsets, rather than using HyperSplit algorithm on all the sub-rulesets, it is well worth checking whether one sub-ruleset is uniform enough to warrant an attempt to use the faster classifier (use HiCuts/HyperCuts algorithm) on each sub-ruleset or not.

Now that we have a metric for characterizing range coverage uniformity we can use this metric to decide if cut based algorithms should be used or split based one. Let us denote the height of an interval as H and $D_{\max}$ the maximum number of quasi-balanced from top to bottom.

If the height of each of the quasi-balanced tree is $h_1, h_2, \ldots, h_n$ we have therefore

$$\underbrace{h_1 + h_2 + \cdots + h_n}_{D_{max}} = \overline{h} \times D_{max} \leq H \tag{5}$$

where $\overline{h}$ is the average height of quasi-balanced trees. As quasi-balanced tree has a least a height of 2, we will have $\overline{h} \geq 2$, so that:

$$2 \times D_{max} \leq \overline{h} \times D_{max} \leq H \tag{6}$$

For matching a set of K non-overlapping small rules we need at best a binary decision tree of height at least $log_2 K$. When using the interval tree, all rules in leaves are non overlapping and the overlapping rules are absorbed by rules in higher levels. As explained before we stop the growth of interval tree when there are $binth$ rules in a node. Therefore the height of a balanced interval tree should be close to its lower bound that $\log_2(\frac{\#(non-overlapping\ rules)}{binth})$. On other hand if one wants make a partition of all rules using splits he will need a decision tree of height at least $\log_2 \frac{\#rules}{binth}$, so there is an interest in using a cut-based algorithm only if $H < \log_2 \frac{\#rules}{binth}$. This means that when an interval tree height is between $\log_2(\frac{\#(non-overlapping\ rules)}{binth}) \leq H < \log_2(\frac{\#rules}{binth})$, there is a benefit in term of tree height or equivalently memory access in using cut. The higher bound can be rewritten as $D_{\max} < \frac{1}{2}\log_2 \frac{\#rules}{binth}$. We will use this last criterion to decide to implement a DT with cut or with splits. Indeed, the closer is the tree height from its lower bound the more balanced will be the interval tree.

D. SmartSplit algorithm

Now we can describe the SmartSplit algorithm that build a multiple DT similar to EffiCuts. We first categorize the rules in the ruleset into $small$ and $large$ based on source and destination IPs. We put aside $(large, large)$ rules and build a specific tree for them that will use HyperSplit as these rules should be separated by port fields that have generally non-uniform coverage.

Since $(small, large)$, resp. $(large, small)$, rules are mainly separated by source IP field, resp. by destination IP field, we build the interval tree for both source and destinationIP fields, and we calculate D_{max} for both trees. We merge the set of $(small, small)$ rules with the $(small, large)$ when $D_{max}(srcIP) \leq D_{max}(dstIP)$, and with $(large, small)$ rules when $D_{max}(dstIP) < D_{max}(srcIP)$. This results in two sub-rulesets, S1 containing $(small, large)$ and S2 containing $(large, small)$ rules. One of S1 or S2 will also contains $(small, small)$ rules.

Now, we build for each one S1 and S2 a separate DT that will disentangle orthogonal structures. For the sub-ruleset containing only small ranges on source IP *.resp.* destination IP field, we use $D_{max}(srcIP)$ *.resp.* $D_{max}(dstIP)$ for algorithm recommendation using the criterion we had $D_{\max} < \frac{1}{2}\log_2 \frac{\#rules}{binth}$.

The SmartSplit algorithm is different from the EffiCuts algorithm from two perspectives. First, the SmartSplit algorithm only considers the "orthogonal structure" on IP fields, and separate a ruleset into 3 sub-rulesets, while EffiCuts considers the existence of "orthogonal structure" on both IP and port fields, resulting in $5 \sim 9$ sub-rulesets. Large number of sub-rulesets results in a large number of memory access and therefore lower classification throughput. Second, SmartSplit algorithm tries to maximize the classification speed by using different algorithms on different sub-rulesets, while EffiCuts uses only a variant of HyperCuts on all the sub-rulesets.

Besides the above points, we applied a pruning trick in our implementation of SmartSplit. As we have multiple trees, each should be sequentially tested in order to find the most specific rules. However we store for each node in the decision tree the index of the rule with minimal priority rule among all rules managed by the node. After doing the search on the first tree we use the matched rule number resulting from this first search and compare it to the minimal priority rule index stored at the node and we pursue the search if and only if the index of minimal priority rule is less than the already matched rule index. If not we prune the search for the whole decision tree. As we observed that generally rules in the $(small, small)$ set are more specific than rules in the $(small, large)$ and the $(large, small)$ set, that are more specific than $(large, large)$ rules, we first check the decision tree containing the $(small, small)$ rules, and we continue by the remaining $(small, large)$ or $(large, small)$ tree and we finish with the $(large, large)$ DT. This pruning optimization reduces the unnecessary memory access in multiple decision trees, improving the look up performance significantly.

V. The AutoPC framework

Combining all the algorithms described above, we propose *AutoPC*, a framework for autonomic construction of decision trees for packet classification. For a given ruleset, *AutoPC* first estimates the memory size requirements. If the estimate is less than a pre-defined threshold M_{th}, a single tree will be built using HyperSplit algorithm. Otherwise, the ruleset will be processed with the *SmartSplit* algorithm. The complete procedure of *AutoPC* is illustrated in Figure 5.

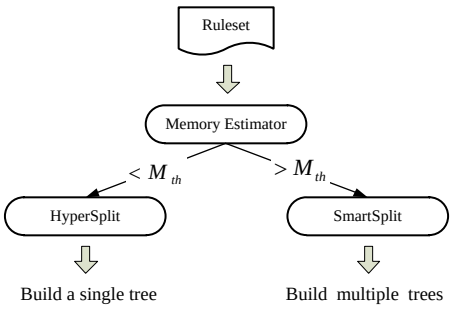


Fig. 5: The AutoPC framework

TABLE V: Node data structure size in bytes

HiCuts	**1**	header information (the dimension to cut, leaf or internal node flag *etc.*)
	6	boundary information, **4** bytes are used to store the *min* value of the boundary of one dimension. **2** bytes are used to store the number of cuts.
	1	1 byte is used for storing the bit shift value.
	4	pointer to the children pointer array.
HyperSplit	**8**	4 bytes for the split point. Other bytes for the header information.

VI. Experimental Methodology

In this section we will validate the analysis presented before. For this purpose we have implemented HiCuts, HyperSplit and EffiCuts algorithms in our experiments. In each node of HiCuts tree, we have used a pointer array instead of a bitmap to index child nodes, allowing more cuts per node (at most 65536 cuts in our implementation). However, in this case, each node needs 2 memory accesses (one for index array and one for node). Our HiCuts implementation enables `Range Compaction` and `Node Merging` optimization however disables the `Rule Move Up` for node size efficiency [6].

For HyperSplit algorithm, the code from [19] is used. To note, when calculating the memory size, the original source code does not account for the memory of rule pointers, we add this part of memory for a fair comparison. Each node of HyperSplit needs only one memory access.

For EffiCuts algorithm, we obtained the implementation from its authors and enable all its optimization techniques. The *spfac* of EffiCuts is set to 8 while the *spfac* of HiCuts is set to 4. The *binth* number is set to 16 for HiCuts, HyperSplit and EffiCuts .

For SmartSplit algorithm, we found that the number of $(large, large)$ rules are usually small compared to the size of the original ruleset, we therefore use $binth = 8$ for the HyperSplit tree built over the $(large, large)$ rules.

For all algorithms, we have stored each rule using 18 bytes [16]. Each rule needs one memory access. Note that EffiCuts has its own way of calculating the number of memory access (in their code, each rule needs less than one memory accesses). For a fair comparison, we use the results directly from the code of EffiCuts.

Table V shows the data structure of each node for HiCuts and HyperSplit. The header size of one node in HiCuts is 12 bytes while each node of HyperSplit needs only 8 bytes. The pointer size in all the algorithms is 4 bytes.

We use ClassBench [18] to generate synthetic rulesets. In our experiment, we have used all available types of rules including Accesses Control List (ACL), Firewall (FW) and IP Chain (IPC). For each type, we have generated rulesets containing from 1K to 100K rules.

Our experiments include performance comparison on both memory size and memory accesses observed from the built decision tree as well as real evaluation of classification speed on a commodity server. The speed are measured through averaging the lookup latency over a low locality traffic generated by ClassBench; Each trace contains 1 millions of 5 tuples. All experiments are run on Ubuntu machines, with 8 cores, Intel i7 processors, 4MB L3 Cache and 24GB of DRAM.

VII. Experiment Results

A. The memory size of the improved HyperSplit algorithm

Looking at the original code of HyperSplit, we found that the implementation [19] does not implement a simple optimization technique, *Rule Overlap*, introduced by HyperCuts. HyperCuts uses this technique to remove rules that are completely covered by another rules with higher priority in the subregion of each node. We added this optimization technique in the decision tree building process of HyperSplit.

The memory size of the original HyperSplit, the improved HyperSplit and the estimate memory size are shown in the Figure 6. In Figure 6, we observe up to $1 \sim 2$ orders of magnitude memory reduction after using the Rule overlap optimization. It is noteworthy that the memory footprint estimate is closer to the actual memory size for the optimized HyperSplit. Since the memory size of HyperSplit is usually $1 \sim 2$ orders of magnitude smaller than HiCuts [13], our memory consumption model can therefore be viewed as to provide a lower bound of both HyperSplit and HiCuts algorithms. We will use optimized HyperSplit in the remaining experiments.

B. Memory Size and Real Performance

In order to explore the relationship between the memory size and the real performance of packet classification on software based platform, we run the HyperSplit algorithm on 25 example rulesets, with memory footprint ranging from less than 10K to larger than 700MB. We measure the cache miss rate and the average memory access latency of the HyperSplit matching process on our experimental platform and we show in Figure 7 the relationship of memory size and memory access latency, and in Figure 8 the relationship of memory size and cache miss rate.

As can be seen in Figure 7, the memory access latency increases slowly with memory size varying from 10KB to 1MB. When the memory size becomes larger than 10MBytes, the latency explodes. The increasing memory access latency can be explained by the fact that the memory footprint of the DT prohibit it to fit into the processor cache memory of our platform (4MB of L3 cache). As shown in Figure 8, the cache miss rate stay below 10% when the memory size is less

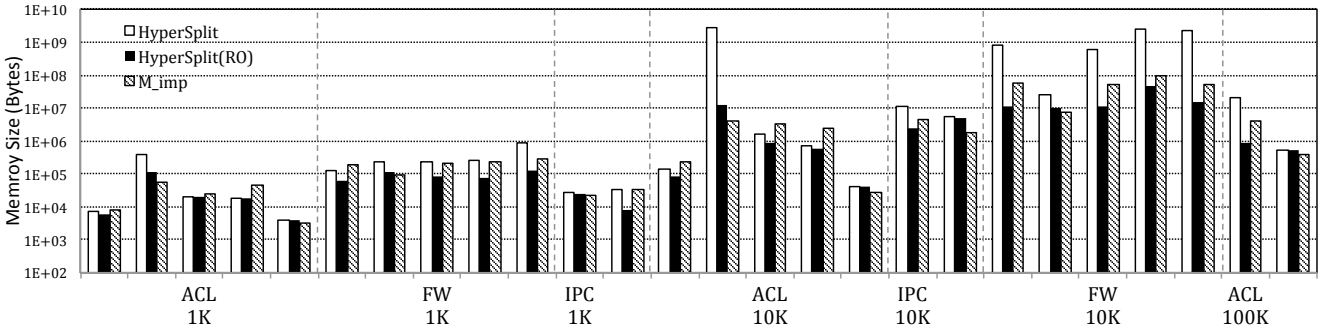


Fig. 6: The memory size of HyperSplit, the improved HyperSplit and the memory size estimate.

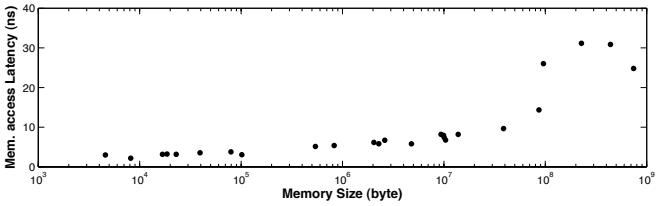


Fig. 7: Average Memory Access Latency and Memory Size

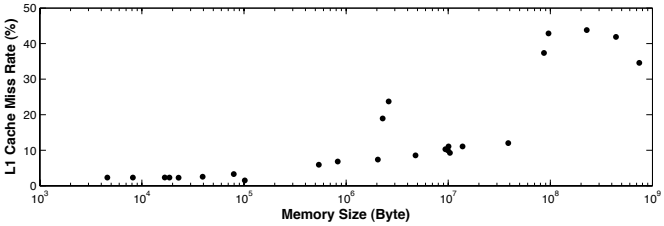


Fig. 8: Cache Misses Rate and Memory size

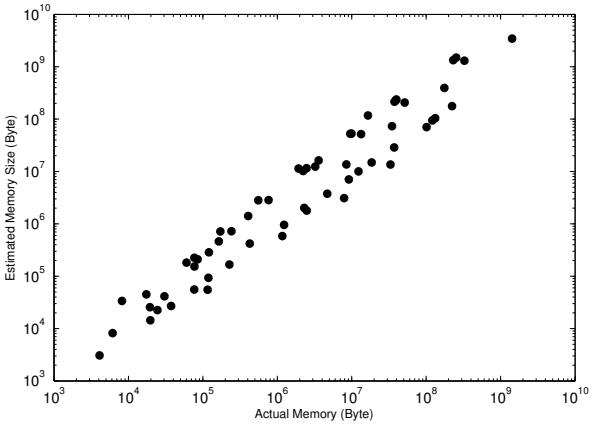


Fig. 9: Estimated and Actual memory size with $binth = 16$

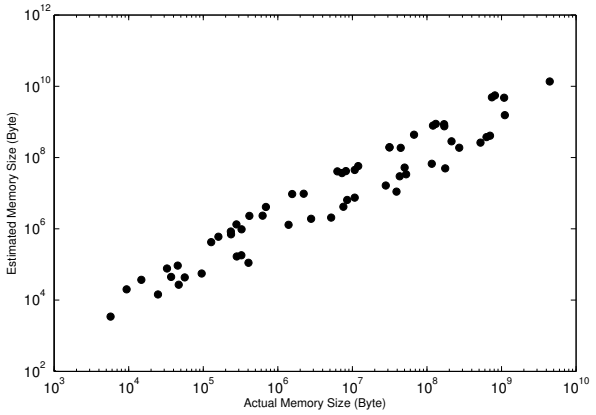


Fig. 10: Estimated and Actual memory size with $binth = 8$

than 10^7 Bytes, and it increases significantly to around 50% when the memory size goes beyond 10^8 Bytes. Based on this observation we set the memory threshold M_{th} in the AutoPC framework to 10MBytes to consider splitting rulesets when estimated memory size is larger than 10MB.

C. Estimated and Actual Memory

We apply our memory consumption model on 60 rulesets of various size consisting of 1K, 5K, 10K, 20K and 50K rules. In the experiments, we first divide the source-destination IP space into 256 equal-sized rectangular sub-space, and perform memory size estimation in each sub-space to obtain a better estimate. We also set the $binth$ to different values (16 and 8) to evaluate its impact on the memory size estimation. We present the estimated and observed memory footprint for $binth = 16$ in Figure 9 and for $binth = 8$ in Figure 10.

Both Figure 9 and Figure 10, show that the estimated and observed memory size remain aligned around a perfect prediction line in logarithmic scale, meaning that the order of magnitude of the estimated memory is correct. As mentioned before, the memory access latency increases with the order of magnitude of memory size increases. Therefore, our memory consumption can be used to predict better the classification performance of a ruleset than using the number of memory access.

We show in Figure 11 the estimated and actual number of rulesets within the special memory size interval. We see that our consumption model is capable of identifying the rulesets into the right categories with small errors. In our experiment,

Ruleset	HyperSplit $log_2 Mem$	HyperSplit Time(s)	Estimate $log_2 Mem$	Estimate Time(s)
ac1_100K	19.7	167	21.4	0.4
acl2_100K	26.6	234	27.2	0.6
acl3_100K	28	1794	30	0.7
acl4_100K	27	1061	29	0.6
acl5_100K	19	186	18.5	0.4
ipc1_100K	30	2424	29	0.6
ipc2_100K	29	1132	28	0.6
fw1_100K	30	2124	32	1.7
fw2_100K	30	2568	29	0.8
fw3_100K	29.5	1148	32	1.9
fw4_100K	33	6413	34	10
fw5_100K	30	1891	32	2

TABLE VI: Estimated and Actual Memory size of Large rulesets

the average memory size estimate error (mean($\frac{est}{actual}$)) with $binth = 16$ is 2.57, and with $binth = 8$ the error is 2.79.

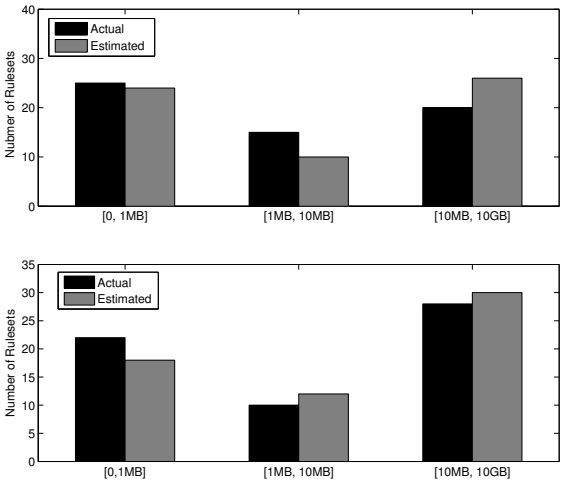


Fig. 11: The estimated and actual number of rulesets for $binth = 16$(top) and $binth = 8$(bottom)

We present the estimate and actual memory size of all the 100K rulesets with $binth = 16$ in Table VI. The memory size varies from less than 1MBytes to several GBytes, and the time for building a HyperSplit trees varies from tens of minutes to several hours. In practice, HyperSplit algorithm has the smaller building time than HiCuts and EffiCuts, *e.g.*, the HyperCuts code usually takes $1 \sim 2$ hours while the EffiCuts code usually takes $5 \sim 9$ hours to build a Decision tree.

Table VI shows that our memory consumption model is able to detect in less than one second that the large ruleset (acl1_100K and acl5_100K) which has small memory footprint avoiding the application of SmartSplit and enabling fast classification with small memory size and few memory accesses.

D. Comparing SmartSplit and EffiCuts

In this section we compare EffiCuts and SmartSplit that both use multiple trees. Figure 12 shows the memory size

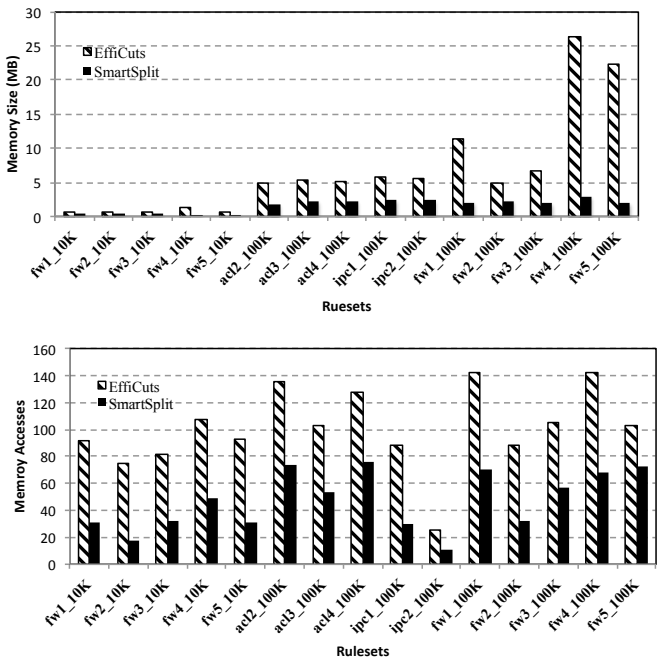


Fig. 12: Memory and Accesses for EffiCuts and SmartSplit

Ruleset	D_{max} srcIP	D_{max} dstIP	mem. acc. S_m	mem. acc. S_s	$\frac{1}{2} log_2 \frac{\#rules}{binth}$	EffiCuts tree num.
fw1_10K	2	1	4	7	4	9
fw2_10K	2	2	4	4	4	7
fw3_10K	2	2	4	7	4	7
fw4_10K	4	2	10	18	4	9
fw5_10K	2	1	4	6	4	7
acl2_100K	13	13	32	25	6	7
acl3_100K	10	1	8	26	6	8
acl4_100K	10	12	31	27	6	8
ipc1_100K	1	1	8	6	6	9
ipc2_100K	2	1	6	5	6	3
fw1_100K	14	6	20	28	6	9
fw2_100K	4	2	4	18	6	7
fw3_100K	14	2	7	28	6	7
fw4_100K	5	4	27	18	6	9
fw5_100K	13	4	21	29	6	7

TABLE VII: Detailed Information of Large rulesets

and number of memory accesses of EffiCuts and SmartSplit. As shown in Figure 12, SmartSplit outperforms EffiCuts both in memory size and in number of memory accesses. For example for fw5_100K ruleset, EffiCuts consumes 22.46MB of memory size, while the memory size of SmartSplit is only 1.98MB, about $11.3\times$ smaller; for fw2_10K ruleset, the worst number of memory accesses for EffiCuts is 75, while the number of memory accesses for SmartSplit is only 18, about $4.1\times$ less. These results show that using multiple algorithms for one ruleset, improve greatly the performance. Moreover this validate the fact that the "orthogonal structure" over IP fields is the main cause of high memory footprint for single decision trees. Through untangling the "orthogonal structure", the memory size decreases dramatically from several gigabytes to less than 2 mega-bytes.

Detailed information about large rulesets are shown in Table VII. As mentioned above, the SmartSplit algorithm split

rulesets into three sub-rulesets. We use S_m to denote the sub-ruleset resulting from merging $(small, small)$ rules with either $(small, large)$ or $(large, small)$ rules, S_{ll} to denote the sub-ruleset containing the $(large, large)$ rules and S_s for the other rules not merged with $(small, small)$.

Among all sub-rulesets, S_m and S_s contain more than 80% of the rules. The memory size and number of memory accesses of the decision trees built on S_m and S_s usually contribute the most in the total performance results. We therefore present the performance results of S_m and S_s in Table VII.

We observe in Table VII that for all the FW 10K rulesets $D_{max}(srcIP)$ and $D_{max}(dstIP)$ is very small, *i.e.*, we have applied the HiCuts algorithm on both S_m and S_s. The large number of cuts per node makes the built tree "flat", reducing the total number of memory accesses of S_m and S_s from 8 to 28. Among 100K-rules rulesets, the IPC rulesets have uniform range distribution on both IP fields, therefore the total number of memory accesses of S_m and S_s is very small (only 11 and 14).

The FW 100K rulesets have uniform range distribution on destination IP field and non-uniform range distribution on source IP field, so that SmartSplit applies HyperSplit on S_s resulting in small memory size, from 100KB to 400KB in our experiments, and HiCuts on S_m for fewer memory accesses. We see the number of memory accesses of S_s increases to around 30 while this value for S_m is still small. However, since the SmartSplit algorithm only generates 3 sub-rulesets, the total number of memory accesses remains small, while, EffiCuts algorithm usually builds $5 \sim 9$ trees on large rulesets and yield more than 100 memory accesses.

E. Real Performance Evaluation

We implement an optimized and fast packet matching program capable of reading the built tree data structure from multiple algorithms into memory and performing rule matching using the resulting DTs. We implemented HiCuts, HyperSplit and SmartSplit in the packet matching program, and used AutoPC framework to configure the program. We also implement EffiCuts, but disabling the `Node Co-lacation` and `Equal-dense Cuts` optimization tricks described in [20] to simplify the implementation. It is noteworthy that in Section VII-D, we compared SmartSplit with EffiCuts enabling all the optimizations.

We first compare the real measured performance of SmartSplit and EffiCuts on rulesets with large memory size in Figure 13. We see that SmartSplit runs significantly faster than EffiCuts. For all the FW 10K rulesets, SmartSplit achieves beyond 10 Millions of Lookup Per Second (MLPS) while EffiCuts only achieves $2 \sim 4$ MPLS. For larger rulesets, SmartSplit is usually 2 times faster than EffiCuts.

We present in Table VIII the lookup speed of AutoPC and EffiCuts in terms of millions of lookup per second (MPLS)[3]. The evaluation shows that the AutoPC framework is in average 3.8/18.9 times faster than using EffiCuts/HyperSplit solely on different type of rulesets.

[3]the results marked with * means AutoPC builds a single tree on the ruleset and - means the building program runs out of memory.

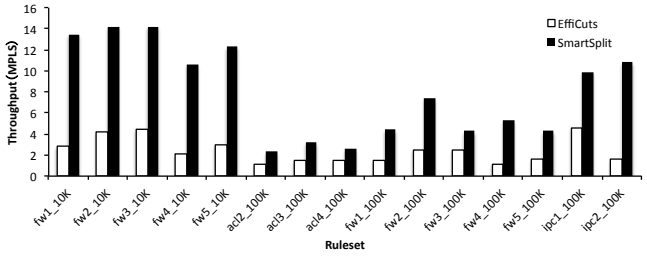


Fig. 13: Comparing the mesured performance of SmartSplit and EffiCuts

Type	Size	AutoPC (MLPS)	EffiCuts (MLPS)	speedup	HyperSplit (MLPS)	speedup
ACL	1K	11.3*	4.5	2.4	11.3	1
	10K	6.9*	3.1	2.2	6.9	1
	100K	8.6*	2.2	3.9	8.6	1
FW	1K	9.8*	2.4	4.1	9.8	1
	10K	10.7	2.1	5.1	2.6	4.1
	100K	7.4	2.5	3.0	-	-
IPC	1K	12.6*	3.0	4.25	12.6	1
	10K	5.3*	1.48	3.6	5.3	1
	100K	9.91	1.63	6.1	0.07	141
Average Speedup: 3.8(to EffiCuts) / 18.9 (to HyperSplit)						

TABLE VIII: Real Performance Evaluation of AutoPC, EffiCuts and HyperSplit

VIII. RELATED WORK

A. Packet classification algorithms

Previous packet classification algorithms can be classified into three categories: hash-based, decomposition based, decision-tree based. Tuple Space Search [17] groups rules by the prefix lengths on multiple dimensions, and constructs one hash table for each group. For each incoming packet, the hash-tables are searched in parallel, and the matched rules are pruned by the priority. However the number of hash-tables varies for different type of rulesets, resulting in the same performance unpredictability issue.

Decomposition based algorithms, such as Cross-producting [5], RFC [5], and ABV [1], perform parallel per-dimension look-up and combine the results using cross-product tables or bit vectors. These algorithms, while fast, are restricted for small rulesets because 1) the size of crossproduct tables grows rapidly with the size of rulesets. 2) It is difficult to implement wide bit vectors in hardware required by large rulesets.

This paper has discussed the decision-tree based algorithms HiCuts/HyperCuts, HyperSplit and EffiCuts. Another work, Adaptive Binary Cutting(ABC) [16], performs cutting adapted to the skewness of prefixes distribution of the rulesets. Although it overcomes the inefficiency of equal-sized cutting, it still suffers from high memory footprint when processing the FW rulesets [16]. Modular packet classification [21] bins rules by indexes formed by selected bits from different dimensions, and builds decision-trees for the rules in each bin. According to [16], ABC is more efficient than Modular packet classification both in memory size and the number of memory accesses. ParaSplit [4] proposed a simulated annealing method to partition the ruleset into sub-rulesets, however, it needs tens of thousand iterations to achieve an optimal partitioning.

B. Software based packet classification systems

Recent advance in multi-core technology gives rise to a wide research interest in building high performance packet processing system on commodity servers. Yadi Ma [9] leverages the parallelism of multi-core and build a software based packet classification system which achieves 15Gbps of throughput. Other work, such as RouteBricks [3], PacketShader [7], CuckooSwitch [22], all achieves tens of gigabits per second throughput on the commodity hardware. This paper studies the performance of software based packet classification from the algorithmic perspective. The proposed algorithms can therefore be used on these systems to achieve higher throughput.

IX. CONCLUSION

In this work, we identify the intrinsic characteristics of rulesets that yield the performance unpredictability issue in the decision-tree based algorithms. Based on these observations, we propose a memory consumption model, a "coverage uniformity" analysis algorithm and an framework capable of identifying which algorithm is suited for a given ruleset through combining the model and the analysis algorithm.

The experimental results show that our method is effective and efficient. Our SmartSplit algorithm is significantly faster and more memory efficient than the state-of-art work, and our AutoPC framework can automatically perform memory size and accesses tradeoff according to the given ruleset. In the experiments, compare to EffiCuts, the SmartSplit algorithm has achieved up to 11 times less memory consumption as well as up to 4 times few memory accesses. The real performance evaluation shows that SmartSplit is usually $2 \sim 4$ times faster than EffiCuts. The AutoPC framework achieves in average 3.8 times faster classification performance than using EffiCuts solely on all the rulesets.

Besides these performance improvement, we believe that the observations in this paper provide a new perspective to understand the connection between ruleset features and the performance of various decision-tree based algorithms.

ACKNOWLEDGEMENT

This work was supported by the National Basic Research Program of China with Grant 2012CB315801, the NSF of China (NSFC) with Grants 61133015 and 61202411, the National High-tech R&D Program of China with Grant 2013AA013501 and DNSLAB, China Internet Network Information Center, Beijing 100190.

REFERENCES

[1] F. Baboescu and G. Varghese. Scalable packet classification. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 199–210. ACM, 2001.

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[3] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *ACM SOSP*, volume 9. Citeseer, 2009.

[4] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang. Parasplit: A scalable architecture on fpga for terabit packet classification. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, pages 1–8. IEEE, 2012.

[5] P. Gupta and N. McKeown. Packet classification on multiple fields. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 147–160. ACM, 1999.

[6] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*, pages 34–41, 1999.

[7] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 195–206. ACM, 2010.

[8] Y. Ma and S. Banerjee. A smart pre-classifier to reduce power consumption of tcams for multi-dimensional packet classification. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 335–346. ACM, 2012.

[9] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging parallelism for multi-dimensional packetclassification on software routers. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 227–238. ACM, 2010.

[10] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vcrib: virtualized rule management in the cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 23–23. USENIX Association, 2012.

[11] M. H. Overmars and F. A. van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21(3):629–656, 1996.

[12] H. Peng, G. Hongtao, L. Mathy, K. Salamatian, and X. Gaogang. Toward predictable performance in decision tree based packet classification algorithms. In *The 19th IEEE LANMAN Workshop*. IEEE, 2013.

[13] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *INFOCOM 2009, IEEE*, pages 648–656. IEEE, 2009.

[14] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[15] 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*, pages 213–224. ACM, 2003.

[16] H. Song and J. Turner. Abc: Adaptive binary cuttings for multi-dimensional packet classification. *IEEE/ACM TRANSACTIONS ON NETWORKING*, 2012.

[17] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 135–146. ACM, 1999.

[18] D. Taylor and J. Turner. Classbench: A packet classification benchmark. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2068–2079. IEEE, 2005.

[19] Hypersplit source code. http://security.riit.tsinghua.edu.cn/share/index.html.

[20] B. Vamanan, G. Voskuilen, and T. Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 207–218. ACM, 2010.

[21] T. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1213–1222. IEEE, 2000.

[22] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.