A Sorted-Partitioning Approach to Fast and Scalable Dynamic Packet Classification

Sorrachai Yingchareonthawornchai, James Daly, Alex X. Liu, and Eric Torng

***Abstract*— The advent of software-defined networking (SDN) leads to two key challenges for packet classification on the dramatically increased dynamism and dimensionality. Although packet classification is a well-studied problem, no existing solution satisfies these new requirements without sacrificing classification speed. Decision tree methods, such as HyperCuts, EffiCuts, and SmartSplit can achieve high-speed packet classification, but support neither fast updates nor high dimensionality. The tuple space search (TSS) algorithm used in Open vSwitch achieves fast updates and high dimensionality but not high-speed packet classification. In this paper, we propose a hybrid approach, PartitionSort, that combines the benefits of both TSS and decision trees achieving high-speed packet classification, fast updates, and high dimensionality. A key to PartitionSort is a novel notion of ruleset sortability that provides two key benefits. First, it results in far fewer partitions than the TSS. Second, it allows the use of multi-dimensional interval trees to achieve logarithmic classification and update time for each sortable ruleset partition. Our extensive experimental results show that The PartitionSort is an order of magnitude faster than the TSS in classifying packets while achieving comparable update time. The PartitionSort is a few orders of magnitude faster in construction time than SmartSplit, a state-of-the-art decision tree classifier, while achieving a competitive classification time. Finally, the PartitionSort is scalable to an arbitrary number of fields and requires only linear space.**

***Index Terms*— Packet classification, software-defined networking, online algorithms.**

I. Introduction

A. Motivation and Problem Statement

SOFTWARE-DEFINED Networking (SDN) defines a new abstraction to separate the control plane and the data layer of the networking stack. Abstraction means the use of common interfaces for heterogeneous network devices of different vendors and architectures to communicate with the central controller and to implement the functions demanded by the controller. One instance of SDN is the OpenFlow standard [1], [2].

Manuscript received June 23, 2017; revised March 28, 2018; accepted June 21, 2018; approved by IEEE/ACM Transactions on Networking Editor J. Liu. This work was supported in part by the National Science Foundation under Grant CNS-1318563, Grant CNS-1524698, and Grant CNS-1421407, in part by the National Natural Science Foundation of China under Grant 61472184 and Grant 61321491, and in part by the Jiangsu High-Level Innovation and Entrepreneurship (Shuangchuang) Program. *(Corresponding author: Sorrachai Yingchareonthawornchai.)*

The authors are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: yingchar@cse.msu.edu; dalyjame@cse.msu.edu; alexliu@cse.msu.edu; torng@cse.msu.edu).

Digital Object Identifier 10.1109/TNET.2018.2852710

The core packet processing functions in SDN are (1) packet classification and (2) rule update which interact through a shared rule list data structure The advent of SDN leads to new challenges, dynamism and dimensionality, which make rule update have similar importance to packet classification.

In SDN, rules are frequently inserted, modified or deleted by the controller to meet an applications' requirements such as establishing a new network path with a given quality of service. Thus, update time in addition to classification time must be minimized. For example, Open vSwitch [3] uses Priority Tuple Space Search (PTSS) for its packet classifier even though PTSS has relatively slow classification due to PTSS supporting fast update. The importance of fast update in OpenFlow is highlighted by Kuźniar *et al.*'s work [4]; they show that in deployed OpenFlow switches that use ternary content addressable memory (TCAM), the data plane might lag behind the control plane by as much as 400 ms and rule updates may happen out of order. This is an obvious security risk.

Dimensional scalability has become an important factor as complicated functionality requires much more flexibility of field in packet processing. In traditional packet classification, the number of fields is 5 with relatively few update requests. In SDN space, the number of fields tend to grow larger to support sophisticated functionality of networking applications. In OpenFlow, for example, the number of header field support is 45. Recently, $P4$ has been introduced to support programming protocol independent packet processing [2]. Hence, any packet classifier must adapt to arbitrary number of fields.

The dynamic packet classification problem is defined as follows. A packet field f is a variable whose domain, denoted $D(f)$, is a set of nonnegative integers. A rule over d packet fields $f_1, f_2, \cdots, f_d$ can be represented as $(s_1 \subseteq f_1) \wedge (s_2 \subseteq f_2) \wedge \cdots \wedge (s_d \subseteq f_d) \rightarrow decision$ where each s_i is an interval in domain $D(f_i)$. For example, prefix 192.168.*.* denotes the range [192.168.0.0, 192.168.255.255]. A packet $p = (p_1, p_2, \cdots, p_d)$ satisfies the above rule if and only if the condition $(p_1 \in s_1) \wedge (p_2 \in s_2) \wedge \cdots \wedge (p_d \in s_d)$ holds. A table R consists of a sequence of n rules $\langle r_1, r_2, \cdots, r_n \rangle$. *Given a packet p, the packet classification problem is to find the first rule in $\langle r_1, r_2, \cdots, r_n \rangle$ that p matches.* In addition, dynamic packet classification must support fast-update in terms of insertion and deletion of rules without significantly slowing down classification speed.

B. Limitations of Prior Art

One of the fundamental challenges in packet classification is to simultaneously achieve high-speed classification and fast

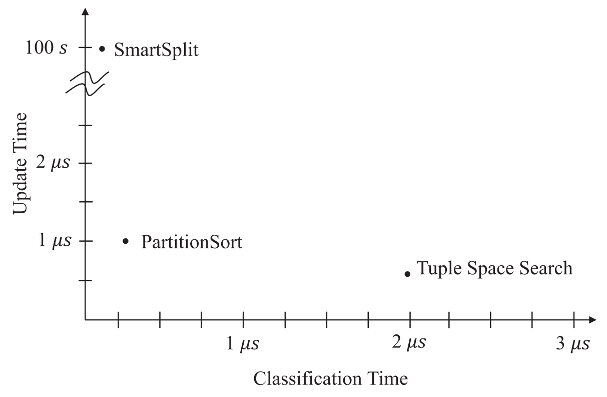

Fig. 1. PartitionSort is uniquely competitive with prior art on both classification time and update time. Its classification time is almost as fast as SmartSplit and its update time is similar to Tuple Space Search.

update. Most existing methods such as SmartSplit [5] have focused on minimizing classification time while sacrificing update time and memory footprint while some methods like Tuple Space Search (TSS) [6] have sacrificed classification time to achieve fast updates. The net result is that the high-speed classification methods are not competitive on update time whereas the fast update methods are not competitive on classification time as shown in Figure 1.

SmartSplit [5] is the state-of-the-art decision tree method as it builds upon the previous best methods HyperCuts [7], EffiCuts [8], and HyperSplit [9]. SmartSplit achieves high-speed classification by leveraging the logarithmic search times of balanced search trees. Unfortunately, no decision tree method including SmartSplit offers a faster update method other than reconstructing the tree because an update often requires too many modifications to the search tree. SmartSplit offers the fastest construction time for any decision tree method by analyzing the rules in question and then constructing HyperCuts or HyperSplit trees based on its analysis. This brings SmartSplit's construction time down to a few minutes which is several orders of magnitude larger than standard OpenFlow's requirements where updates should be completed in microseconds [4].

TSS [6] used in Open vSwitch [3] is the state-of-the-art fast update method. TSS achieves fast updates by partitioning an OpenFlow ruleset into smaller rulesets based on easily computed rule characteristics so that rules can be quickly inserted and deleted from hash tables; however, TSS has low classification speed because the number of partitions is large and each partition must be searched for every packet.

C. Proposed Approach

Our proposed approach, PartitionSort, introduces the concept of *ruleset sortability* that allows PartitionSort to unify the benefits of high-speed decision tree classifiers and the fast-update TSS classifier to achieve a new classification scheme that is uniquely competitive with both high-speed classification methods on classification time and fast update methods on update time as shown in Figure 1. PartitionSort partitions a ruleset into a small number of sortable rulesets. We store each sortable ruleset in a multi-key binary search tree leading to $O(d+\log n)$ classification and update time per sortable ruleset where d and n are the number of fields and rules, respectively. The memory requirement is linear in the number of rules.

PartitionSort is a hybrid approach that leverages the best of decision trees and TSS. Similar to TSS, PartitionSort partitions the initial ruleset into smaller rulesets that support high-speed classification and fast updates. Similar to decision tree methods, PartitionSort achieves high-speed classification for each partition by using balanced search trees.

D. Technical Challenges and Proposed Solution

We face four key technical challenges in designing our PartitionSort approach. The first is *defining ruleset sortability for multi-dimensional rules*. It is challenging to sort multi-dimensional rules in a way that helps packet classification. We address this challenge by describing a necessary requirement that *any* useful ordered ruleset must satisfy and then introduce our field order comparison function that progressively compares multi-dimensional rules one field at a time.

The second technical challenge is *designing an efficient data structure for sortable rulesets* that supports high-speed classification and fast update using only linear space. We show that our notion of ruleset sortability can be translated into a multikey set data structure and hence it is possible to have $O(d + \log n)$ search/insert/delete time using a multi-key balanced binary search tree. We also propose a path compression optimization to speed up classification time.

The third technical challenge is *effectively partitioning a non-sortable ruleset into as few sortable rulesets as possible*. We address this challenge by a reduction to a new graph coloring problem with geometric constraint. Then, we develop a fast offline sortable ruleset partitioning algorithm that runs in milliseconds.

The fourth challenge is *designing an effective online partitioning algorithm*. This is particularly challenging because we must not only achieve fast updates but also preserve fast classification time in the face of multiple updates. We use an offline ruleset partitioning algorithm as a subroutine to design a fast online ruleset partitioning algorithm that still achieves fast classification time.

E. Summary of Experimental Results

We conduct extensive comparisons between PartitionSort and other methods, in particular TSS and SmartSplit. Our results show that PartitionSort is uniquely competitive with both TSS in update time and SmartSplit in classification time. More specifically, PartitionSort performs updates in 0.65 μs on average which is comparable to that of TSS while SmartSplit requires 100s of seconds to reconstruct a classifier. At the same time, PartitionSort classifies packets in 0.29 μs on average which is comparable to SmartSplit while TSS requires an order of magnitude more time. With caching, PartitionSort classifies packets roughly 1.84 times faster than TSS.

The rest of the paper is organized as follows. We first describe related work in Section II. We then define ruleset sortability in Section III. We then describe an Multi-dimensional Interval Tree in Section IV, an offline partitioning scheme in Section V and the full PartitionSort algorithm in Section VI. We provide theoretical analysis of PartitionSort in both worst-case and average case in Section VII. We then describe an optimization based on Rule-Splitting to group incompatible rules in the same partition in Section VIII. We then give our experimental results in Section IX and conclude the paper in Section X.

II. Related Work

From computational geometry point of view, packet classification problem is essentially a priority Orthogonal Point Enclosure Query (OPEQ) problem where we preprocess a set of n d-dimensional axis-parallel boxes so that we can report the highest priority box that contains a query point. A special case where $d = 1$, also known as the priority interval stabbing query problem, can be solved in $\Theta(\log n)$ time using linear memory in dynamic setting [10], [11]. We can extend d-dimensional solutions to $(d+1)$-dimensional solution using segment trees at a cost of multiplicative factor of $O(\log n)$ in both space and time [12]. Unfortunately, using linear space, any data structure requires $\Omega(\log n(\log n/\log d)^{d-2}+k)$ time to process a query [13]. It is unclear if the Point Location problem which is essentially an OPEQ problem with non-intersecting boxes can be solved in sub-polylogarithmic time. The best known result is due to Rahul's result [14] for the static version where queries can be processed in $O(\log^{d-3/2} n)$ time. In packet classification, however, rulesets do not exhibit worst-case instances [15], [16]. In particular, if we restrict the structure of rulesets given by ClassBench [17], we show that it is possible to achieve $O(d\log n+\log^2\ n)$ classification time using linear memory.

We divide previous work on packet classification into four categories: decision tree methods, partitioning methods, hybrid methods that use both decision trees and partitioning, and TCAM-based methods. Decision-tree methods such as SmartSplit [5], HiCuts [18], HyperCuts [7], and HyperSplit [9] recursively partition the packet space into several regions until the number of rules within each region falls below a predefined threshold. These methods leverage the logarithmic search time of decision trees to achieve fast software-based packet classification. Their key drawback is that when partitioning, each rule in the original rule list is copied into a new sublist for each region that intersects it. It is possible that rules may be replicated into many regions leading to high memory consumption as well as slow and complicated updates since each copy must be inserted or deleted. In contrast, the decision trees in PartitionSort have no rule replication. Thus, PartitionSort uses linear space and supports fast updates.

Partitioning methods such as Tuple Space Search [6] and SAX-PAC [19] work by partitioning the original ruleset into a collection of smaller rulesets that are each easier to manage. TSS partitions rules based on tuples which are the patterns of prefix lengths of each field in a given rule. The specificity of TSS partitioning is both a strength and weakness. It is good because the resulting partition can be searched using a hash function leading to $O(d)$ classification time per partition and $O(d)$ rule update time. It is bad because it produces a large number of partitions that need to be searched, resulting in slow classification time. PartitionSort overcomes this issue by using a relaxed partitioning constraint that results in far fewer partitions. In contrast, SAX-PAC uses a more general partitioning scheme than PartitionSort; specifically, rules fit within a partition as long as they are non-intersecting. Every sortable ruleset is non-intersecting, but not vice versa. The drawback with SAX-PAC is that there is no known fast classification method for non-intersecting rulesets unless the number of fields is just two. In contrast, PartitionSort achieves high-speed and fast-update classification on sortable rulesets for any number of fields.

A few hybrid methods combining partitioning and decision trees such as Independent Set [20], EffiCuts [8] and SmartSplit [5] have been proposed to reduce rule replication and manage the super-linear space requirements of decision tree methods. Independent Set develops a weaker partitioning scheme that also eliminates rule replication. Specifically, they use only one field to partition rules requiring that rules are non-overlapping in that field. In contrast, PartitionSort uses multiple fields to partition rules. Independent Set's one-field partitioning scheme creates many more partitions leading to significantly slower classification time. On the other hand, EffiCuts and SmartSplit use partitioning schemes based on observable rule characteristics, but they do not eliminate rule replication which means rule updating is still complex. Specifically, EffiCuts and SmartSplit initially place rules into the same partition if they are small or large in the same fields. They then produce either a HyperCuts or HyperSplit tree for each partition. Whereas both methods significantly reduce rule replication and the super-linear space requirements of decision trees, neither eliminates rule replication; thus both still suffer from poor rule update performance.

Finally, there are many hardware-based TCAM approaches for packet classification [21]–[28]. TCAM can be used when the classifiers are small, but TCAM sizes are extremely small and TCAM consumes lots of power. It is not clear that TCAM-based classifiers can scale to handle the demands of OpenFlow packet classification. Furthermore, the construction times of the most compressive TCAM-based approaches, which are required to deal with the limited sizes, are too large to be used in the dynamic OpenFlow environment.

III. Ruleset Sortability

We must overcome two main challenges in defining sortable rulesets. The first is that we must define an ordering function $\leq$ that allows us to order rules which are d-dimensional hypercubes. Consider the two boxes r_1 and r_4 in Figure 2; one might argue that $r_1 \leq r_4$ since r_1's projection in the X dimension is smaller than r_4's projection in the X dimension; however, one could also argue that $r_4 \leq r_1$ because r_4's projection in the Y dimension is smaller than r_1's projection in the Y dimension. The second is that $\leq$ must be useful for packet classification purposes. Specifically, suppose $\leq$ defines a total ordering on rules r_1 through r_n and we are trying to classify packet p. If we compare packet p to rule r_i and determine that $p > r_i$, then packet p can only match some rule among rules r_{i+1} through r_n. That is, packet p must not be able to match any of rules r_1 through r_i.

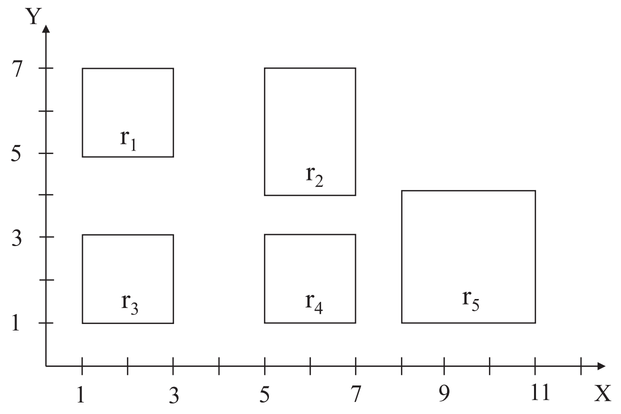

Fig. 2. Example of ordered ruleset with $\leq_{xy}$.

A. Field Order Comparison Function

We propose the following *field order comparison function* for sorting rules. The intuition comes from the fact that sorting one-dimensional rules is straightforward. The basic idea is that we choose a field order (permutation) $\vec{f}$ and then compare two rules by comparing their projections in the current field until we can resolve the comparison. Continuing the example from above, if we choose field order XY, then $r_1 \leq_{XY} r_4$ because $[1,3]$, r_1's projection in field X, is smaller than $[5,7]$, r_4's projection in field X. A more interesting example is that $r_4 \leq_{XY} r_2$ because their projections in field X are both $[5,7]$ which forces us to resolve their ordering using their projections in field Y.

We now formally define our field order comparison function. We first define some notation given a field order $\vec{f}$. Let $\vec{f_i}$ denote the ith field of $\vec{f}$. For any rule r, let $\vec{f_i}(r)$ denote the projection of r in $\vec{f_i}$, and let $\vec{f^i}(r)$ denote the projection of r into the first i fields of $\vec{f}$.

We now define how to compare two intervals using $\vec{f}$. For any two intervals $i_1 = [\min(i_1), \max(i_1)]$ and $i_2 = [\min(i_2), \max(i_2)]$, we say that $i_1 < i_2$ if $max(i_1) < min(i_2)$. We say that two intervals are not comparable, $i_1 \parallel i_2$, if they overlap but are not equal. Together, this defines a partial ordering on intervals. To illustrate, $[1,3] < [4,5]$, but $[1,3] \parallel [3,5]$ as they overlap at point 3.

We define the field order comparison function $\leq_{\vec{f}}$ based on field order $\vec{f}$ as follows.

Definition 1 (Field Order Comparison): Consider two rules s and t and a field order $\vec{f}$. If $\vec{f_1}(s) \parallel \vec{f_1}(t)$, then $s \parallel_{\vec{f}} t$. If $\vec{f_1}(s) < \vec{f_1}(t)$, then $s <_{\vec{f}} t$. If $\vec{f_1}(s) > \vec{f_1}(t)$, then $s >_{\vec{f}} t$. Otherwise, $\vec{f_1}(s) = \vec{f_1}(t)$. If $\vec{f}$ contains only one field, then $s =_{\vec{f}} t$. Otherwise, the relationship between s and t is determined by $\vec{f'} = \vec{f} \setminus \vec{f_1}$. That is, we eliminate the first field and recursively compare s and t starting with field $\vec{f_2}$ which is $\vec{f'}_1$.

To simplify notation, we often use the field order $\vec{f}$ to denote its associated field order comparison function $\leq_{\vec{f}}$.

For example, consider again the rectangles in Figure 2 and $\vec{f} = XY$. We have $r_1 \leq_{XY} r_2$ since $X(r_1) < X(r_2)$ and X is the first field, and we have $r_3 \leq_{XY} r_1$ because $X(r_3) = X(r_1)$ and $Y(r_3) \leq Y(r_1)$. The set of 5 rectangles are totally ordered by XY; that is, $r_3 <_{XY} r_1 <_{XY} R_4 <_{XY} r_2 <_{XY} r_5$. On the other hand, the set of 5 rectangles are not totally ordered by YX because $r_4 \parallel_{YX} r_5$.

We now define sortable rulesets.

Definition 2 (Sortable Ruleset): A ruleset R is sortable if and only if there exists a field order $\vec{f}$ such that R is totally ordered by $\vec{f}$.

B. Usefulness for Packet Classification

We now prove that our field order comparison function is useful for packet classification. We first must define how to compare a packet p to a rule r. Using the same comparison function does not work because we want p to match r if the point defined by p is contained within the hypercube defined by r, but $\leq_f$ would say $p \parallel_{\vec{f}} r$ in this scenario. We thus define the following modified comparison function for comparing a packet p with a rule r using field order $\vec{f}$.

Definition 3 (Packet Field Order Comparison): If $\vec{f_1}(p) < \min(\vec{f_1}(r))$, then $p <_{\vec{f}} r$. If $\vec{f_1}(p) > \max(\vec{f_1}(r))$, then $p >_{\vec{f}} r$. If $\vec{f}$ contains only one field, then p matches r. Otherwise, the relationship between p and r is determined by $\vec{f'} = \vec{f} \setminus \vec{f_1}$.

Lemma 1: Let n d-dimensional rules r_1 through r_n be totally ordered by field order $\vec{f}$, and let p be a d-dimensional packet. For any $2 \leq i \leq n$, if $p >_{\vec{f}} r_i$, then $p >_{\vec{f}} r_j$ for $1 \leq j < i$. Likewise, for any $1 \leq i \leq n-1$, if $p <_{\vec{f}} r_i$, then $p <_{\vec{f}} r_j$ for $i < j \leq n$.

Proof Sketch: We discuss only the first case where $p >_{\vec{f}} r_i$ as the other case is identical by symmetry. Consider any packet p and rules r_i and r_j where $p >_{\vec{f}} r_i$ and $r_i >_{\vec{f}} r_j$. Applying Definitions 1 and 3, we can prove that transitivity holds and $p >_{\vec{f}} r_j$. □

IV. Multi-dimensional Interval Tree (MITree)

We describe an efficient data structure called Multi-dimensional Interval Tree (MITree) for representing a sortable ruleset given field order $\vec{f}$. In particular, we show that MITrees achieve $O(d + \log n)$ time for search, insertion, and deletion; we refer to this as $O(d + \log n)$ *dynamic time*.

An obvious (and naive) approach is to store an entire rule for each node and construct a balanced binary search tree because the rules are totally ordered by field order $\vec{f}$. The running time is clearly $O(d \log n)$ because the height of a balanced tree is $O(\log n)$ and each comparison requires $O(d)$ time. The problem is this running time is not scalable to higher dimensions. Figure 3 shows an example sortable ruleset with its corresponding naive binary search tree.

We can speed up the naive approach by a factor of d by using one field at a time with a clever balancing heuristic. We show this in two steps. First, we show $O(d + \log n)$ dynamic time for a special case where all intervals are points. We extend this solution to the general case of sortable rulesets.

The special case where all intervals are points is exactly a multi-key dictionary problem. A multi-key dictionary is a dynamic set of objects that have comparisons based on a lexical order of keys. One example of a multi-key dictionary is a database table of n rows and d columns. There exists a multi-key binary search tree supporting $O(d + \log n)$ dynamic time for the multi-key dictionary problem [29], [30].

	X	Y	Z
r_1	[0,2]	[0,6]	[1,2]
r_2	[3,4]	[0,1]	[1,2]
r_3	[3,4]	[2,3]	[3,5]
r_4	[3,4]	[4,5]	[3,6]
r_5	[5,6]	[0,5]	[1,2]
r_6	[5,6]	[0,5]	[4,6]
r_7	[5,6]	[0,5]	[7,9]

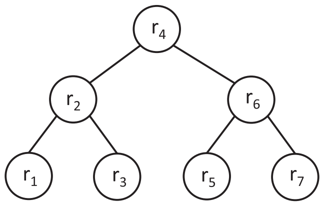

Fig. 3. A sortable ruleset with 7 rules and 3 fields, field order $\vec{f} = XYZ$, and the corresponding simple binary search tree where each node contains d fields.

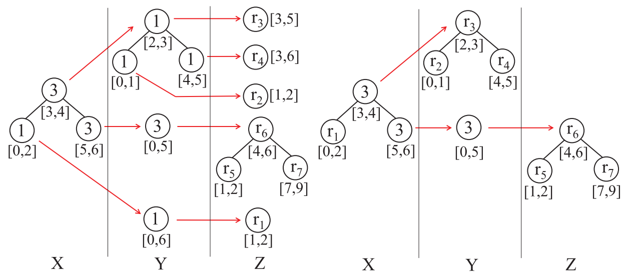

Fig. 4. We depict the uncompressed (left) and compressed (right) MITree that correspond to the sortable ruleset from Figure 3. Regular left and right child pointers are black and do not cross field boundaries. Next field pointers are red and cross field boundaries. In the compressed MITree, nodes with weight 1 are labeled by the one matching rule.

In the general case of sortable rulesets, we show that the $O(d + \log n)$ result is attainable. First, all keys are compared in lexical order because, by definition of sortable rulesets, field order $\vec{f}$ is essentially a permutation of the natural order $[1..d]$. Furthermore, it is immediate by the definition of sortable rulesets that if two rules are identical in $[f(1), f(2), \ldots, f(i-1)]$ then at $f(i)$, they either overlap fully or have no intersection. This property allows us to construct an MITree that behaves as if the interval is only a point. Hence, we can use the same balancing heuristic to achieve $O(d+\log n)$ dynamic time. The search procedure is identical to point keys except that we need additional checking for intervals for each interval tree node.

These results yield the following Theorem.

Theorem 1: Given a sortable ruleset, Multi-dimensional Interval Tree (MITree) supports searches, insertion, and deletion in $O(d + \log n)$ time using only linear space.

We perform one additional path compression optimization. We remove duplicate intervals for each field in the Multi-dimensional Interval Tree so that each unique interval is shared by multiple rules. Each node v thus represents a particular interval in a given field. Each v thus has a third child containing the rules that match on v. Figure 4 shows an MITree for the ruleset in Figure 3. Let $c(v)$ be the number of rules that match node v. We observe that c decreases from earlier fields to later fields. Thus if $c(v) = 1$, then there is only one rule r that can match. In this case, we store the remaining fields of r with v as shown in Figure 4. For the rest of this paper, we use MITree to refer to a compressed MITree.

V. Offline Sortable Ruleset Partitioning

Rulesets are not necessarily sortable, so we must partition rulesets into a set of sortable rulesets. Because the classification time depends on the number of sortable rulesets after partitioning, our goal is to develop efficient sortable ruleset partitioning algorithms that minimize the number of sortable rulesets. Here we consider the offline problem where all rules are known *a priori*. We start by observing that the offline ruleset partitioning problem is essentially a coloring of an intersection graph (to be defined shortly) by the $d!$ possible field orders for sortable rulesets.

Formally, the basic approach is to partition ruleset R into χ ordered rulesets $(R_i, \vec{f}(i))$ for $1 \le i \le \chi$ where $\bigcup_{1 \le i \le \chi} R_i = R$, $R_i \cap R_j = \phi$, and each subset R_i is sortable under field order $\vec{f}(i)$. There is no relation between field orders $\vec{f}(i)$ and $\vec{f}(j)$; they may be the same or different. Let R consist of n, possibly overlapping, d-dimensional rules v_1 through v_n. For each of the $d!$ field orders $\vec{f}(i)$ for $1 \le i \le d!$, we define the corresponding interval graph $G_i = (V, E_i)$ where $V = R$ and $(v_j, v_k) \in E_i$ if $v_j \parallel_{\vec{f}(i)} v_k$ for $1 \le j < k \le n$. We then define the interval multi-graph $G = (V, \bigcup_{1 \le i \le d} E_i)$ where each edge is labeled with the field order that it corresponds to. Our sortable ruleset partitioning problem is that of finding a minimum coloring of V where the nodes with the same color form an independent set in G_i for some $1 \le i \le d!$; the minimum number of colors is denoted $\chi(G)$.

Our offline partitioning framework is to find a maximal independent set of *uncolored* nodes in some G_i. We "color" these nodes which means we remove them from the current ruleset to construct an MITree. We then repeat until all nodes are colored. We now describe how to find a *large* number of rules that are sortable in one of the $d!$ possible field orders.

We use the weighted maximum interval set from [31] to compute a maximum independent set for any G_i in $O(dn \log n)$ time. Since there are $d!$ possible field orders, finding the maximum set out of all of the field orders is infeasible for large d. Instead, we propose a greedy approximation, GrInd, that finds a maximal independent set and runs in time polynomial in both n and d. GrInd achieves this by avoiding full recursion for each subproblem and choosing $\vec{f}$ and the maximal independent set simultaneously.

We now describe GrInd which uses d rounds given d fields. In the first round, we have one set of rules which is the complete set of rules. GrInd selects the first field in the field order as follows. For each field f, it projects all rules into the corresponding interval in field f. It then weights each unique interval by the number of rules that project onto it plus one. We "add one" to bias GrInd towards selecting more unique intervals. GrInd then computes a maximum weighted independent set (using the method in [31]) of these unique weighted intervals. The first field is the field f that results in the largest weighted independent set. GrInd then eliminates all other intervals and their associated rules.

In subsequent rounds, the process is identical except for the following. Rather than having one set of rules, we have multiple partitions or subsets of rules to work with. Specifically, the rules corresponding to each unique interval from the previous round correspond to a subset of rules that might have

Algorithm 1 GrInd $(R, \vec{f})$

```
Input: Rulelist $R$
Output: A maximal sortable rulelist $R'$ and field order $\vec{f}'$
1  $L = [R]$
2  $\vec{f}' = []$
3  for $i = 1$ to $d$ do
4      for unselected field $f$ do
5          foreach rulelist $\ell$ in $L$ do
6              let $S$ be the set of projections from $\ell$ onto $f$
7              let $w_s = 1+$ num rules with projection $s \in S$
8              let $S_\ell = \text{MaxWeightIndSet}(S, \vec{w})$
9          let weight $w_f = \sum_{\ell \in L} |S_\ell|$
10     add $f$ with max weight to $\vec{f}'$
11     remove from each $\ell \in L$ any $r$ where $f(r) \notin S_\ell$
12     subpartition each $\ell \in L$ based on projection $f(r)$
13 let $R'$ equal one rule from every list in $L$
14 return $(R', \vec{f}')$
```

conflicts and must be processed further. GrInd processes each partition in parallel. For each remaining field, GrInd chooses the next field to add to $\vec{f}$ based on the sum of maximum independent sets from each partition.

This process is repeated until all fields have been used or every set has only one rule. The remaining rules are sortable under field $\vec{f}$, and ready to be represented by an MITree. Pseudocode for GrInd is shown in Algorithm 1. The running time is $O(d^2\ n \log n)$ as shown in Theorem 2.

We give an example of GrInd in action in Figure 5. The input set of boxes with two fields is shown in Figure 5 (a). For both possibilities of first fields, we create the corresponding intervals and the corresponding rules. In this example, the maximum weight choice is to use field X as we can get four rules that corresponds to weight of 6 (4 rules plus 2 partitions). If we used field y instead, the maximum weight choice would have been only 3 rules of total weight 5. We then proceed with the remaining rules r_1, r_2, r_6, and r_7 as illustrated in Figure 5 (b). We compute maximum independent sets for both partitions independently and see that we can choose all rules to get the final result of 4 rules.

Theorem 2: GrInd computes a field order $\vec{f}(i)$ for $1 \le i \le d!$ and a maximal independent set R_i for G_i such that R_i is sortable on $\vec{f}(i)$. GrInd requires $O(d^2\ n \log n)$ time.

Proof: The correctness follows from choosing a maximum weighted independent set in each round. For round $1 \le i \le d$, for each of the $d-i+1$ choices for fields in round i, the running time of computing the maximum weighted independent set is $O(n \log n)$. In later rounds, we likely have fewer than n boxes or rules left, but in the worst case for running time, all n boxes are considered in each round. This leads to the $O(d^2\ n \log n)$ running time. □

VI. PartitionSort: Putting It All Together

In this section, we complete the picture of our proposed packet classifier PartitionSort. We present a fully online rule partitioning scheme where we assume the ruleset is initially empty and rules are inserted and deleted one at a time. We first describe how PartitionSort manages multiple MITrees. We then describe our rule update mechanism which still achieves a small number of partitions. We conclude by describing how PartitionSort classifies a packet given multiple MITrees.

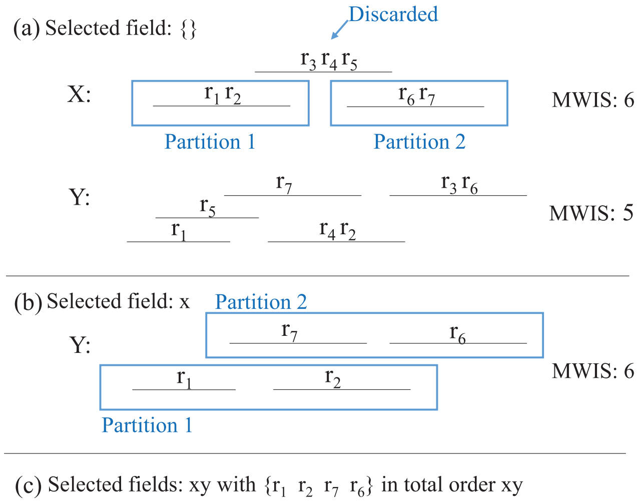

Fig. 5. Example execution of GrInd. (a) Selected field: {}. (b) Selected field: x. (c) Selected fields: xy with {r_1 r_2 r_7 r_6} in total order xy.

To facilitate fast rule update and fast packet classification, PartitionSort sorts the multiple MITrees by their maximum rule priorities. Let $\mathcal{T}$ be the set of trees, $t = |\mathcal{T}|$, $pr(T)$ be the maximum rule priority in tree $T \in \mathcal{T}$, and $\vec{f}(T)$ be the field order of $T \in \mathcal{T}$. PartitionSort sorts $\mathcal{T}$ so that for $1 \le i \le j \le t$, $pr(T_i) > pr(T_j)$. To maintain this sorted order, PartitionSort uses the following data structures. First, a lookup table $M : R \to \mathcal{T}$ identifying which table each rule is in. Second, for each tree $T \in \mathcal{T}$, PartitionSort maintains a heap $H(T)$ of the priorities of every rule in T.

A. Rule Update (Online Sortable Ruleset Partitioning)

Given an existing set of rules, we now describe how PartitionSort performs rule updates (insertions or deletions). We start with the simpler operation of deleting a rule r. Using M, we identify which table T r is in. We then delete r from T, remove $pr(r)$ from $H(T)$, and resort $\mathcal{T}$ if necessary, typically using insertion sort since $\mathcal{T}$ is otherwise sorted. If no rules remain in T, we delete T from M and from $\mathcal{T}$.

We now consider the more complex operation of inserting a rule r. We face two key challenges: choosing a tree $T_i \in \mathcal{T}$ to receive r and choosing a field order for T_i. We consider trees $T_i \in \mathcal{T}$ in priority order starting with the highest priority tree. When we consider T_i, we first see if T_i can receive r without modifying $\vec{f}(T_i)$. If so, we insert r into T_i. However, before committing to this updated tree, if $|T_i| \le \tau$ for some small threshold such as $\tau \le 10$, we run GrInd (cf. Section V) on the rules in T_i (now including r) to use a new field order if GrInd provides one partition with a different field order. This reconstruction of T_i will not take long because GrInd is fast and τ is small. If so, we update the field order and use the

TABLE I
GRIND (G), ONLINE (O) AND TSS (T) AVERAGE NUMBER OF PARTITIONS (TUPLES) AND PERCENTAGE OF RULES IN FIVE PARTITIONS WITH HIGHEST PRIORITY RULES

64k	Average #Partitions			% rules in first five partitions/tuples		
	$\lvert G\rvert$	$\lvert O\rvert$	$\lvert T\rvert$	%G	%O	%T
acl	21.9	23.5	263.0	99.18	98.63	31.01
fw	26.5	32.1	99.8	99.57	97.26	8.01
ipc	10.0	11.4	184.5	99.94	99.89	19.74

new tree T_i. If we cannot insert r into T_i, we reject T_i as incompatible and move on to the next $T_i \in \mathcal{T}$. If no existing tree T_i will work, we create a new tree T_i for r with an empty heap $H(T_i)$. In all cases, we clean up by inserting $pr(r)$ into $H(T_i)$ and resort $\mathcal{T}$ if necessary.

B. Packet Classification

We now describe packet classification in the presence of multiple MITrees. We keep track of hp, the priority of the highest priority matching rule seen so far. Initially $hp = 0$ to denote no matching rule has been found. We sequentially search T_i for $1 \le i \le t$ for packet p. Before searching T_i, we first compare $pr(T_i)$ to hp. If $pr(T_i) < hp$, we stop the search process and return the current rule as no unsearched rule has higher priority than HP. Otherwise, we search T_i for p and update hp if a higher priority matching rule is found. If we reach the end of the list, we either return the highest priority rule or we report that no rule was found. This priority technique was used in Open vSwitch [3] to speed up their implementation of TSS.

We argue that the priority optimization benefits PartitionSort more than other methods. The reason for this is that our partitioning scheme is based on finding maximum independent set iteratively. This produces partitions where almost all the rules are in the first few partitions. In our experiments, we measured the percentage of rules contained in the first five partitions for both PartitionSort and TSS. This data is highlighted in Table I below. To summarize, we find that 99% or more of the rules in all classifiers reside in the first five partitions when ordered by maximum priority. In contrast, TSS exhibits a more random behavior where only some classifiers have many rules in the first five partitions.

VII. ANALYSIS OF PARTITIONSORT

We derive upper bounds on the cost of PartitionSort operations focusing on search or classification time. Insertion time will essentially be the same as search time, though deletion time is faster as we only need to process one tree T_i for deletion. We consider both the worst case and a geometric sequence of sortable ruleset sizes that characterizes many of our experimental rulesets. The main results are shown in Table II. We assume the input ruleset of n d-dimensional rules has been partitioned into b sortable rulesets or trees where n_i is the number of rules in the ith tree T_i, $n_i \ge n_j$ for $1 \le i < j \le b$, $\sum_i n_i = n$, and $\tilde{n} = \prod_{i=1}^{n} n_i^{1/b}$ is the geometric mean of n_i.

TABLE II
BOUNDS FOR PARTITIONSORT (PS) OPERATIONS

	No Assumption	Fat-tail Distribution	
	Worst Case	Worst Case	Average Case[†]
Search	$O(db + b\log\tilde{n})$	$O(d\log n + \log^2 n)$	$O(d + \log n)$
Insertion	$O(db + b\log\tilde{n})$	$O(d\log n + \log^2 n)$	$O(d + \log n)$
Deletion	$O(d + \log n)$	$O(d + \log n)$	$O(d + \log n)$

A. General Case

We first give a conservative analysis of the worst case times for general rulesets. We get a loose upper bound of $O(b(d + \log n))$given b sortable rulesets since each can be represented by an MITree with running time $O(d + \log n)$. We can tighten this bound by using the actual sizes n_i for each ruleset rather than the upper bound of n.

Theorem 3: Given b sortable rulesets with n total rules where sortable ruleset i has n_i rules, PartitionSort requires $O(db + b\log\tilde{n})$ search time and rule-insertion time and $O(d + \log n)$ rule-deletion time where $\tilde{n} = \prod_{i=1}^{b} n_i^{\frac{1}{b}}$.

Proof: We first observe that the i-th sortable ruleset can be represented by an MITree which requires $O(d + \log n_i)$ comparisons for search, insertion and deletion, which is $k(d + \log n_i)$ for some constant k. For classification time, we have to search all trees in the worst case. This requires $\sum_{i=1}^{b} k(d + \log n_i) = k\sum_{i=1}^{b}(d + \log n_i) = k(bd + \sum_{i=1}^{b}\log n_i) = k(bd + \log\prod_{i=1}^{b} n_i) = O(db + b\log\tilde{n})$. Rule insertion is essentially identical. Rule deletion is faster since we have a pointer to the correct tree and thus we only process one tree. □

The exact values for n_i can make a significant difference in search time. At one extreme, if $n_1 = n - b + 1$ and $n_i = 1$ for $i \ne 1$, then $\tilde{n} \le n^{1/b}$ and the running time is $O(db + \log n)$. On the other hand, if $n_i = n/b$ for all i, then $\tilde{n} = n/b$ and the search time is $O(db + b\log n)$. This leads to the following Corollary.

Corollary 1: The search time and rule insertion time for PartitionSort is in the range $O(db + \log n)$ and $O(db + b\log n)$.

B. Number of Trees and Geometric Progression

In this section, we derive tighter bounds on PartitionSort's search time assuming that r_i, the number of rules that are not contained in the largest i trees, follows a geometric progression until we have only a small number of rules left to be placed into a tree. We formalize this property in the following definition.

Definition 4: A sequence of positive integers s_i is a (γ, τ) geometric progression if $s_i \le n/\gamma^i$ until $s_i \le \tau$ where γ and τ are constants > 1.

We show the r_i values for our 64k ACL, FW, and IPC rulesets in Figure 6. From this figure, we see that the IPC, ACL, and FW r_i values are $(2, 1)$, $(2, 65)$, and $(2, 108)$ geometric progressions, respectively, and all the r_i values are $(1.38, 1)$ geometric progressions.

Assuming we work with rulesets where the r_i are geometric progressions, we get the following bound on the number of trees.

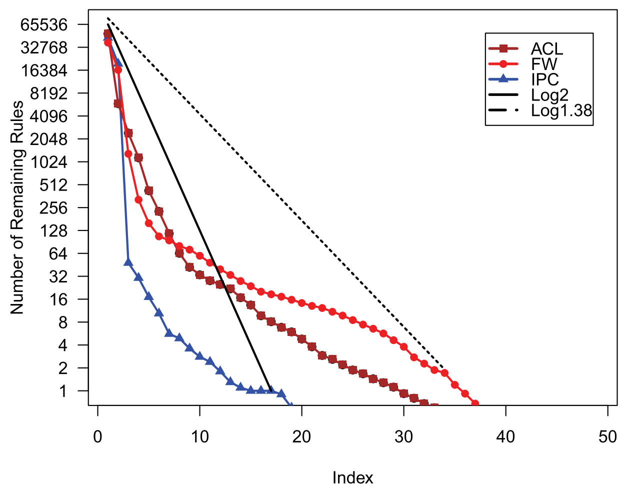

Fig. 6. Average number of remaining rules (r_i) plotted in log scale with base 2 and 1.38. The average is across from partition at i index including those with zeros.

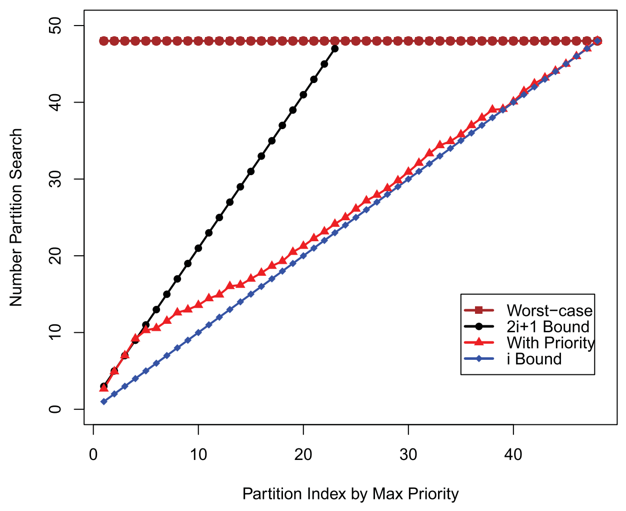

Fig. 7. Plot of $\Delta(T_i)$ for our ACL, FW, and IPC 64k rulesets. The worst-case is the maxmimum number of partitions over all rulesets of size 64k. The average excludes missing values.

Lemma 2: If r_i is a (γ, τ) geometric progression then $b = O(\log n)$.

Proof: The value of i where $r_i \leq \tau$ is at most $\log_\gamma n$ given that $r_i \leq n/\gamma^i$ until then. The remaining number of trees is at worst τ, so the result follows. □

This leads to the following classification time result.

Theorem 4: If r_i is a (γ, τ) geometric progression then the total number of comparisons required by PartitionSort for queries and insertions is at most $O(d \log n + \log^2 n)$.

Proof: This follows directly from Theorem 3 and Lemma 2. Specifically, from Theorem 3, we get that the number of comparisons is $O(db + b \log \tilde{n})$. From Lemma 2, we get that $b = O(\log_\gamma n)$. Combining the two and observing that $\tilde{n} \leq n$, the result follows. □

Looking at the data more closely, all the r_i values drop very quickly until less than roughly 600 or less than 1% of the rules remain. Then, the r_i values taper off more slowly but still at a steady rate. There may be other options to handle the last 1% or so of rules. For example, as suggested by Kogan *et al.*, we could use a TCAM classifier if one is available [19]. Another option is to use a standard decision tree such as a SmartSplit tree. Finally, in Section VIII, we propose a rule-spliting optimization that allows us to further reduce the number of sortable rulesets required.

C. Successful Searches and Priority Optimization

So far, we have assumed that searches must explore all $T_i \in \mathcal{T}$. However, for successful searches, the priority optimization described in Section VI-B allows us to skip some T_i once the correct rule is found. We now analyze how many trees need to be searched for successful searches assuming that each rule is equally likely to be the correct matching rule.

Consider any rule r in the ruleset. Recall that trees are ordered by their maximum priority first. We define $j = \delta(r)$ to be the smallest possible index j such that $pr(r) > pr(T_j)$; this means that if r is the matching rule, we would not need to search tree $T_{\delta(r)}$ or any later rule since all these rules have lower priority than r. For tree T_i, we define $\Delta(T_i) = \sum_{r \in T_i}(\delta(r) - 1)/|T_i|$; that is, $\Delta(T_i)$ represents the average number of trees that need to be searched when T_i contains the matching rule. We plot $\Delta(T_i)$ in Figure 7 along with the best possible values for T_i which is i and the worst possible value which is $|\mathcal{T}|$. As we can see from this plot, the $|T_i|$ is roughly bounded by $2i + 1$ for small i and this improves for larger i. Add to this the fact that the vast majority of our rules are contained in the first five trees (see Table I), then on average only a relatively small number of trees are searched for any successful search.

We now combine the properties of the geometric progression and the priority optimization to obtain the following average case bound on successful searches.

Lemma 3: If a sequence of remaining rules r_i is a (γ, τ) geometric progression and $\Delta(T_i) \leq ci$ for some constant c, then the average case cost of a successful search is $O(d + \log n)$.

Proof: We first observe that the geometric progression of remaining rules can be viewed as a sequence of number of rules in trees where the size of each successive tree decreases by a factor of $1/\gamma$. Second, for all the rules in tree T_i, $\Delta(T_i)$ is the average number of trees that needs to be searched and is simply some constant c times the index i; this average assumes that each rule is equally likely to be hit. Putting these two observations together, we get that the average number of trees searched is $c(1 - 1/\gamma)\sum_{i=1}^{|\mathcal{T}|} 1/\gamma^{i-1}$ which is a constant in c and γ. The search cost in any tree is $O(d + \log n)$, so the result follows. □

VIII. Rule Splitting

We propose a rule splitting optimization where we speed up classification time by creating fewer trees. Specifically, we split some of the rules into multiple pieces that are more compatible with each other. The result is a new semantically equivalent rule list that can be represented using fewer trees. Since the number of trees is more important than the number of rules, this will usually reduce the search time required.

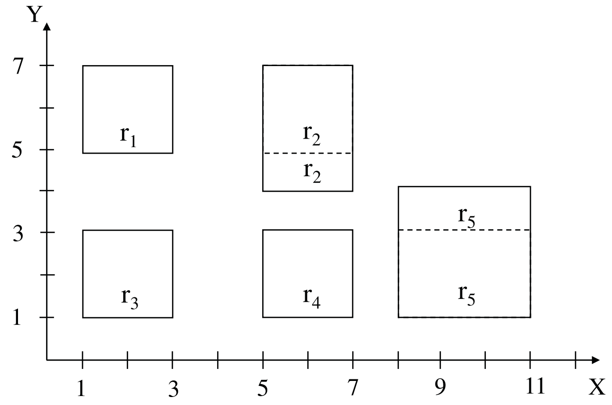

Fig. 8. Splitting the rules from Figure 2 in YX order.

For example, consider the rules in Figure 2. In YX order, only r_1, r_3, and r_4 are compatible. If we cut r_2 at $Y = 5$ and r_5 at $Y = 3$ as shown in Figure 8, then all of the rules are compatible. The resulting tree would contain 7 rules.

Rule splitting is similar to the rule replication that occurs in HyperCuts and other decision trees where a rule is replicated into multiple branches. PartitionSort has an advantage that it can adjust the number of trees and which rules are associated with them in order to control the amount of splitting that occurs. Other methods like HyperCuts must do splitting because all of the rules are placed in the same tree. These methods tend to have very high memory costs as they are forced to resolve essential rule incompatibilities. Some methods such as EffiCuts and SmartSplit do some separation to control replication, but their control is less refined. We offer a continuum of tradeoffs between the number of trees, their height, and the amount of memory required (via replication or splitting).

A. Offline Splitting

We now describe how PartitionSort builds a tree that includes split rules. First, we select a core set of rules C and their field order using GrInd. This forms the basis for our tree. Other rules will be selected, split, and added to this list.

From the remaining rules, we select the core-compatible rules: the rules that do not require splitting any rule in C. We then compute how much each of these rules would need to be split in order to add them to the rule list. We then sort them by the number of splits required. We then take rules until the number of splits required passes some threshold and reject the remaining rules. We then split the new rule list containing the core set and the accepted rules.

Using the rules in Figure 2, our core set of rules is r_1, r_3, and r_4 in YX order. We consider adding rules r_2 and r_5. Both rules are core-compatible and each only needs to be split into two rules to be acceptable. As long as our limit is at least 7 rules, we split rules r_2 and r_5 as shown in Figure 8 and place all of the rules into a single tree.

If our core set was instead r_2, r_3 and r_4 then r_1 would not be core-compatible. That is because this would require splitting r_2, which is not allowed by our method.

B. Determining Core-Compatibility

We must be able to determine if rule r is core-compatible with a core set of rules, C given field order $\vec{f}$. We do this by comparing r to each $r_i \in C$. First consider field f_0. If $r[f_0]$ is disjoint from $r_i[f_0]$, then the rules are compatible. Otherwise, if $low(r[f_0]) > low(r_i[f_o])$ or $high(r[f_0]) < high(r_i[f_o])$, then r_i would need to be split and r is not compatible. Otherwise, we look at the next field. If no fields remain in $\vec{f}$, then r_i overlaps r, but does not need splitting, and so this is allowed (we'll remove the overlapped fragment of r later).

C. Splitting a Rule

We now describe how to split a rule r against all of the rules of a list ℓ given field order $\vec{f}$. Note that some of the other rules in ℓ will also need to be split. This will happen by the same process and all of the fragments will be placed in a new list.

First, we select field f_0 and get $r_i[f_0]\ \forall r_i \in \ell$. From this list of ranges, we produce the list of disjoint segments and keep the ones that overlap $r[f_0]$. For each segment s, we produce a copy of r with $r_s[f_0] = s$. We then select all of the rules in ℓ that overlap with s. We then repeat this process on the next field with r_s and ℓ_s on the next field.

We use this same process to count how many fragments a rule must be split into. During the process, we keep track of how many fragments are produced. If they exceed a predefined limit, then we determine that the rule requires too much splitting and immediately reject it from the list.

IX. Experimental Results

A. Experimental Methods

We compare PartitionSort to two classification methods: Tuple Space Search (TSS) and SmartSplit. TSS is the de facto standard packet classifier in OpenFlow classification because it has the fastest updates. It is a core packet classifier in Open vSwitch. SmartSplit is the fastest decision tree method as it analyzes the ruleset to then choose between HyperSplit and HyperCuts. Like EffiCuts, it may construct multiple decision trees. We also compare PartitionSort to SAX-PAC to assess the effectiveness of our partitioning strategies. Our implementation of PartitionSort is available at GitHub.[1]

1) Rulesets: We use the ClassBench utility [32] to generate rulesets since we do not have access to real rulesets. These rulesets are designed to mimic real rulesets. It includes 12 parameter files that are divided into three different categories: 5 access control lists (ACL), 5 firewalls (FW) and 2 IP chains (IPC). We generate lists of 1K, 2K, 4K, 8K, 16K, 32K, and 64K rules. For each size, we generate 5 classifiers for each parameter file, yielding 60 classifiers per size and 420 classifiers total.

We also consider the effect of the number of fields on the packet classifier. We modified ClassBench to output rules with multiple sets of 5 fields. Although this does not necessarily model real rulesets with more fields, it does allow us to project how our algorithms would perform given rules with

[1]https://github.com/sorrachai/PartitionSort

more fields. We generate rulesets with 5, 10, 15, and 20 fields and 64K rules. As before, we generate 5 classifiers for each parameter file yielding a total of 20 rulesets per parameter file or 240 rulesets in total.

2) Tuple Space Search : Our TSS implementation is the one from GitHub described in Pfaff *et al.* [3]. Specifically, we use their hash table implementation in the TSS scheme. We use their Priority TSS implementation that searches tuples in decreasing order of maximum priority.

3) SmartSplit: We use an implementation that is written by the authors of the paper which is hosted on GitHub.[2]

4) PartitionSort : For all comparisons, we run PartitionSort with a fully online partitioning scheme unless explicitly stated otherwise; that is, the classifiers are generated by starting with an empty classifier and then repeatedly inserting rules. We perform internal comparisons between online and offline partitioning strategies.

5) Caching: We also compare PartitionSort and TSS with flow caching focusing on the microflow and megaflow caching used in Open vSwitch [3].

In Open vSwitch, a microflow is an exact-match rule using all packet header bits of an actual packet and the corresponding action that was applied to that packet. The idea is that subsequent packets of a flow can be immediately classified using one hash table lookup. The limitation is that microflows can only recognize packets that have already been seen. In contrast, a megaflow is a wild-card rule where specific bits of a complete packet header are replaced with wildcards; in OVS, these wildcard bits often include all the bits from some fields such as port fields. The intuition is that one megaflow with many wildcard bits can match flows we have not yet seen.

OVS uses two-level flow caching as follows. When processing a packet, OVS first checks the microflow cache, then the megaflow cache, and finally the OpenFlow tables. The megaflow cache is implemented as a TSS classifier without priority since OVS ensures all megaflows in the cache are non-overlapping. Thus, OVS can terminate once a match is found. OVS needs a table for each combination of wildcard bits used in the megaflow classifier. The key advantage is that the megaflow classifier has many fewer tables than the full OpenFlow classifier.

OVS generates megaflows on-the-fly using bit-tracking. When classifying a packet p, OVS identifies which bits are not needed to classify p. OVS can generate any megaflow where any combination of unneeded bits are replaced with wildcards. The challenge is determining which such bits to wildcard while ensuring the megaflow classifier has a small number of tables.

We implement TSS with two-level caching and PartitionSort with one-level caching (e.g. only microflow cache). We use the OVS github algorithms to implmement both the megaflow and microwflow caches. We set the microflow cache size to 1024 entries. We do not limit the size of the megaflow cache to give TSS every possible advantage.

6) Evaluation Metrics: We use four standard metrics: classification time, update time, space, and construction time. Space is simply the memory required by the classifier. We measure classification time as the time required to classify 1,000,000 packets generated by ClassBench when it constructs the corresponding classifier. In most experiments, we omit caching and consider only slow path packet classification of the first packet from each flow. For some experiments, we do compare TSS with two-level caching to PartitionSort with one-level caching.

We measure update time as the time required to perform one rule insertion or deletion. We perform 1,000,000 updates for each classifier by beginning with each classifier having half the rules. We then perform a sequence of 500,000 insertions intermixed with 500,000 deletions choosing the rule to insert or delete uniformly at random from the currently eligible rules. We report all data by averaging over our rulesets.

When we compare with TSS, for each unique ruleset size and each number of fields and every metric, we average all matching rulesets of the same type: ACL, FW, and IPC.

When we compare with SmartSplit, for each unique ruleset size and each number of fields and every metric, we average together all matching rulesets. We report[3] the average metric value, and, in some cases, a box plot.

7) Machine Environment and Correctness Test: All experiments are run on a machine with Intel i7-4790k CPU@ 4.00GHz, 4 cores, 32 KB L1, 256KB L2, 8MB L3 cache respectively, and 32GB of DRAM. Most of the experiments were run on Windows 7. The exception is the SmartSplit test; both SmartSplit and PartitionSort were run on Ubuntu 14.04.

To ensure correctness, we generate a stream of 1 million packets per ruleset and verified that all of the classifiers agree on how to classify all packets.

B. PartitionSort Versus TSS Without Caching

We first compare PartitionSort with TSS. We compare these algorithms using the metrics of classification time, update time, construction time, and memory usage. We note that both algorithms have extremely fast construction times taking less than a second to construct all rulesets.

1) Classification Time: Our experimental results show that, on average, PartitionSort classifies packets 7.2 times faster than TSS for all types and sizes of rulesets. That is, for each of the 420 rulesets, we computed the ratio of TSS classification time divided by PartitionSort classification time and then averaged these 420 ratios. When restricted to the 60 size 64k rulesets, PartitionSort classifies packets 6.7 times faster than TSS with a maximum ratio of 20.1 and a minimum ratio of 2.6. Figure 9 shows the average classification times for both PartitionSort and TSS across all ruleset types and sizes. If we invert these classification times to get throughputs, PartitionSort achieves an average throughput of 4.5 Mpps (million packets per second) whereas TSS achieves an average throughput of 0.79 Mpps. We note 4.5 divided by 0.79 is less than 7.2; this is because taking an average of ratios is different than a ratio of averages.

[2]https://github.com/xnhp0320/SmartSplit

[3]To minimize side-effects from hardware and the operating system, we run 10 trials for each of classification/update time and report the average.

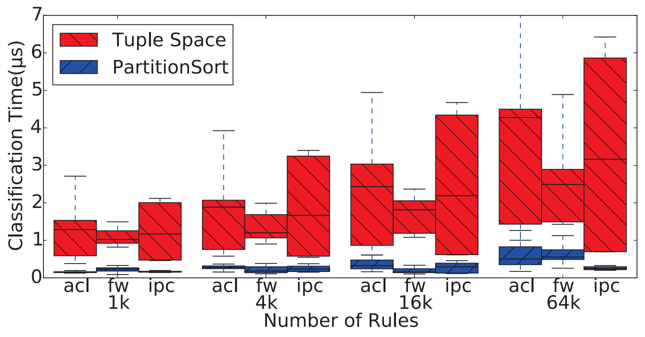

Fig. 9. Classification time by ruleset type and size.

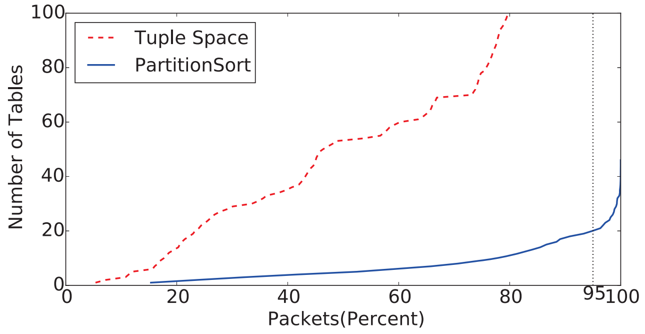

Fig. 10. Partitions/Tuples Queried. A CDF distribution of number of partitions required to classify packets with priority optimization (cf. Section VI-B).

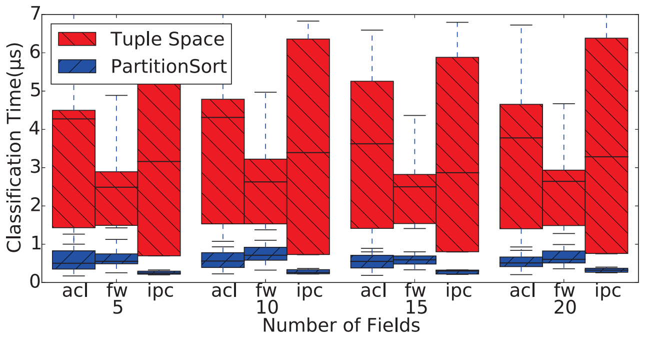

Fig. 11. Classification time on 64K rules.

Besides being much faster than TSS, PartitionSort is also much more consistent than TSS. Specifically, the quartiles and extremes are much tighter for PartitionSort than for TSS where the higher extremes do not even fit on the scale in Figure 9.

The reason for these factors is that PartitionSort requires querying significantly fewer tables than TSS. As seen in Figure 10, PartitionSort needs to query fewer than 20 tables 95% of the time, while TSS needs to only 23% of the time. Since search times should be roughly proportional to the number of tables queried, search times should be similar to the area under their respective curves. Since TSS's area is significantly larger, so too should their search times.

Both methods are relatively invariant in the number of fields, as seen in Figure 11. Because there are more field permutations available, PartitionSort is able to pick the best one for slightly bigger partitions. This results in a slight decrease in lookup times.

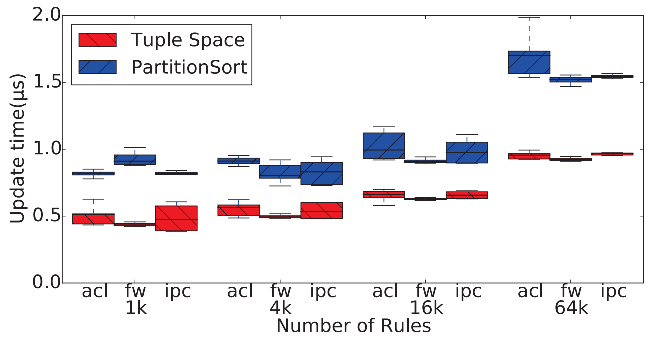

Fig. 12. Update time vs tuple space search.

2) Update Time: Our experimental results show that PartitionSort achieves very fast update times with an average update time of 0.65 μs and a maximum update time of 2.1 μs. This should be fast enough for most applications. For example, PartitionSort can handle several hundred thousand updates per second. Figure 12 shows the average update times across all ruleset types and sizes for both PartitionSort and TSS.

As expected, TSS is faster at updating than PartitionSort, but the difference is not large. Similar to classification time, we compute the ratio of PartitionSort's update time divided by TSS's update time for each ruleset and then compute the average of these ratios. On average, PartitionSort's update time is only 1.7 times larger than TSS's update time. Restricted to the 64k rulesets, this average ratio is also 1.7. The data from Figure 12 does show that PartitionSort has an $O(\log n)$ update time.

3) Construction Time: Our experimental results show that PartitionSort has very fast construction times, with an average construction time of 83 ms for the largest classifiers. This is small enough that even significant changes to the ruleset cause only minor disruption to the packet flow. As expected, TSS builds faster than PartitionSort, but the difference is not large. On average, PartitionSort's construction time is only 1.9 times larger than TSS's construction time. Restricted to the 64k rulesets, this average ratio decreases to only 1.4.

4) Memory Usage: Our experimental results show that our PartitionSort requires less space than TSS. Both are space-efficient, requiring $O(n)$ memory, with PartitionSort requiring slightly less space than TSS.

C. PartitionSort Versus TSS With Caching

Our classification time results with and without caching, depicted in Figures 13 and 9, respectively, show that caching significantly improves the performance of both TSS and PartitionSort with more improvement for TSS; however, PartitionSort still classifies packets 1.844 times faster than TSS across all ruleset sizes. The ratio is 1.06 times faster for the 1k rulesets and grows to 2.5 times faster for the 64k rulesets.

These new classification time results can be explained by looking at hit ratios for both algorithms in Figure 14. The hit ratio for TSS with microflow and megaflow caches ranges from 76% for the 64K classifiers to 99% for the 1K classifiers

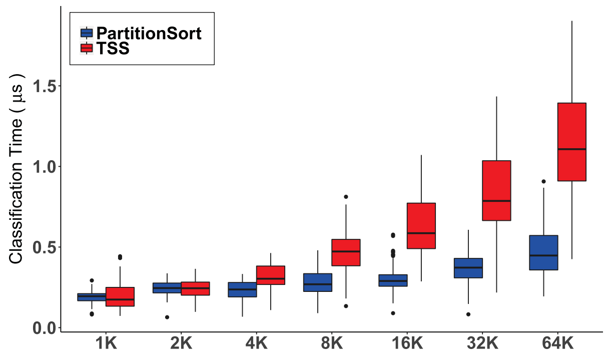

Fig. 13. Classification time with caching by ruleset size.

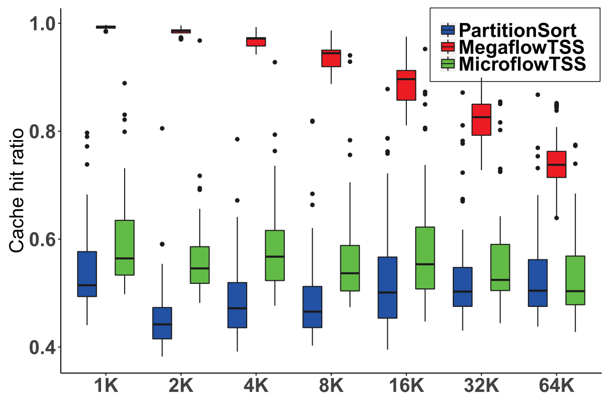

Fig. 14. Cache-hit ratio by ruleset size.

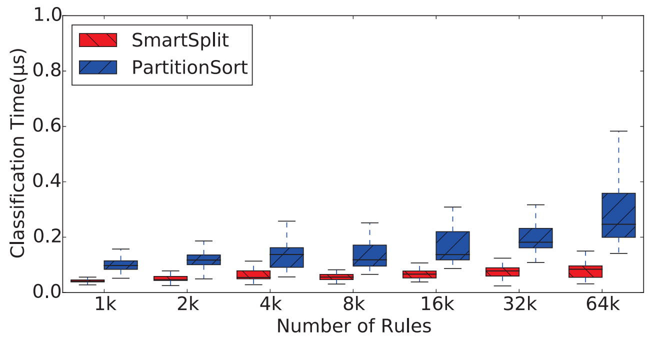

Fig. 15. Classification time vs SmartSplit.

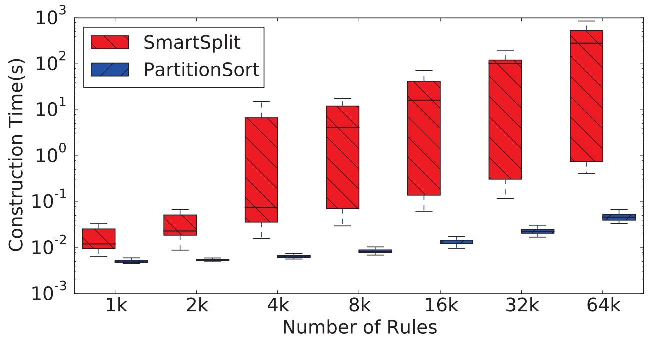

Fig. 16. Construction time in logarithmic scale.

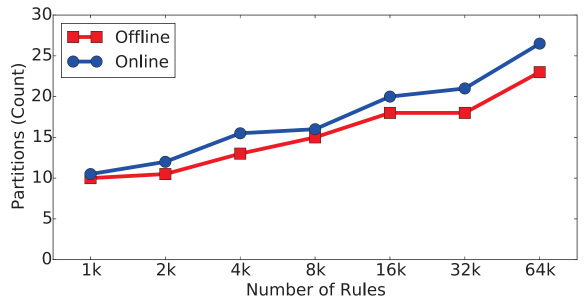

Fig. 17. Number of partitions for offline and online PartitionSort.

whereas the hit ratio for PartitionSort is roughly 54% for all ruleset sizes. Thus, as classifier size increases, TSS must classify more packets.

An interesting observation is the hit ratio for microflow cache in TSS is slightly better than that in PartitionSort because the eviction for TSS is based on the megaflow classifier, which contains more specific rules, whereas the eviction for PartitionSort is based on the entire ruleset.

D. Comparison With SmartSplit

We now compare PartitionSort with SmartSplit. We compare these algorithms using the metrics of classification time and construction time. We do not compare update time because SmartSplit does not support update. We briefly discuss memory.

1) *Classification Time: Our experimental results show that, on average, PartitionSort takes 2.7 times longer than SmartSplit to classify packets.* This can be seen in Figure 15. There are two factors leading to this result. First, SmartSplit uses fewer trees than PartitionSort (the same advantage that PartitionSort has over TSS). Second, the wider branching provided by the HyperCuts trees used by SmartSplit reduces the overall tree height. While both classifiers have logarithmic search times, these two factors make SmartSplit's constant factor smaller.

2) *Construction Time: PartitionSort can be built faster than SmartSplit by several orders of magnitude.* This becomes increasingly more pronounced as the number of rules increases, where PartitionSort takes less than a second but SmartSplit requires almost 10 minutes. This can be seen in Figure 16. Additionally, SmartSplit does not support updating. Together, this means that SmartSplit is not appropriate for cases where the ruleset is updated frequently.

3) *Memory Usage: Our experimental results (not shown) demonstrate that PartitionSort requires less space than SmartSplit.* For some classifiers, SmartSplit requires much more memory than PartitionSort; for others, they are much closer. SmartSplit's memory usage is inconsistent since it depends on whether it makes one tree or several, and whether it makes HyperCuts or HyperSplit trees. As rulesets increase in size, SmartSplit uses multiple HyperSplit trees to try and limit memory usage.

E. Comparison of Partition Algorithms

We now compare our offline (cf. section V) and online (cf. section VI) partitioning algorithms. Figure 17 shows the number of partitions required by each version.

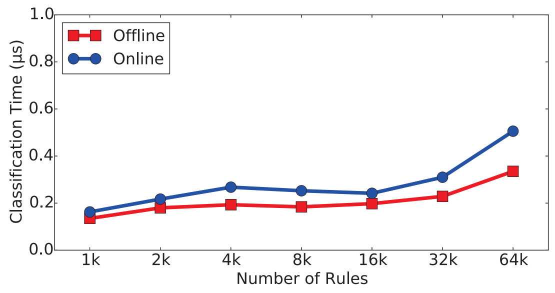

Fig. 18. Classification time for offline and online PartitionSort.

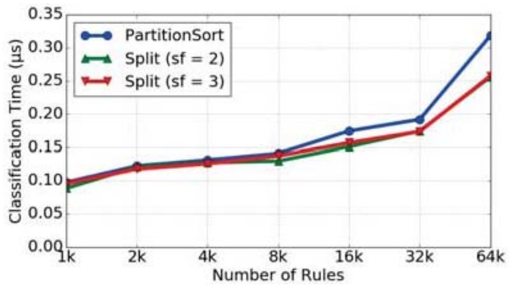

Fig. 19. Classification time for splitting experiments.

Online partitioning performs almost as well as offline partitioning. Online partitioning requires only 17% more trees than offline partitioning. This translates into 31% larger classification times as seen in Figure 18. We conclude that a single offline construction at the beginning with online updates should be more than sufficient for most purposes.

F. Comparison of Split Factors

We now compare offline partitioning with various amounts of splitting. We try several different split factors sf. If there are n rules, up to $sf \cdot n$ rule fragments are allowed. This determines the threshold for how many rules are selected for the table. We performed splitting experiments with sf up to 5, but since this had no further improvement to classification time, we only show $sf = 2$ and $sf = 3$.

1) Classification Time: Splitting significantly improves the search times of PartitionSort. Even allowing only twice as much memory achieves a 30% improvement in classification time. Increasing the split factor sometimes improves the classification time, but most of the time the effects are negligible. This can be seen in Figure 19.

The improvement in classification time is at least partially caused by a reduction in the number of tables required, as seen in Figure 20. This strictly goes down as the split factor increases. However, the classification time does not decrease by the same amount. This occurs for two reasons. First, since we are increasing the number of rules, the search times are going to be higher than the number of tables will imply. Second, later tables are usually smaller and queried less often, so they have a smaller effect on the classification time.

2) Drawbacks of Splitting: We now discuss two drawbacks of splitting. First, *splitting significantly increases the construction time required.* Setting $sf = 2$ increases construction time

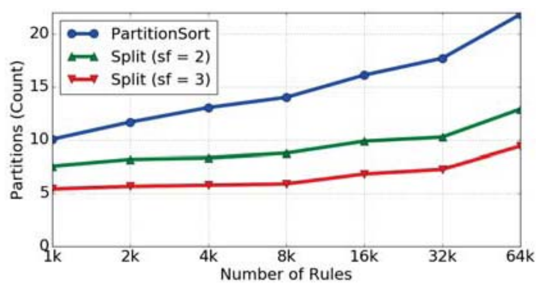

Fig. 20. Number of partitions for splitting experiments.

400 times, but setting $sf = 5$ has very little further effect, increasing construction time on large classifiers by only 11% and actually decreasing construction time for small classifiers. Second, *splitting increases the memory usage, but less than the total amount allowed.* Setting $sf = 2$ raises memory usage by 59% and $sf = 5$ by 359%. This shows that splitting is able to use more memory to increase classification speed and that it stays within the given memory limits.

X. Conclusions

We provide the following contributions. We introduce ruleset sortability that allows us to combine the benefits of fast classification time from decision tree methods with fast update from TSS. We then develop an efficient data structure MITree for sortable rulesets that supports fast classification and update time achieving $O(d + \log n)$ running time. We then give an online ruleset partitioning algorithm that effectively produces few partitions which naturally handles dynamism of SDN packet classification. Taken together, we provide PartitionSort which satisfies the SDN requirements of high-speed, fast-update and dimensionally scalable packet classification.

References

[1] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.
[2] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
[3] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *Proc. 12th USENIX Conf. Netw. Syst. Design Implement.*, 2015, pp. 117–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789779
[4] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Int. Conf. Passive Act. Netw. Meas.* Cham, Switzerland: Springer, 2015, pp. 347–359, doi: 10.1007/978-3-319-15509-8_26
[5] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *Proc. 22nd Int. Conf. Netw. Protocols (ICNP)*, Washington, DC, USA, Oct. 2014, pp. 308–319.
[6] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 1999, pp. 135–146.
[7] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2003, pp. 213–224.
[8] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2010, pp. 207–218.
[9] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 648–656.

[10] Y. Giora and H. Kaplan, "Optimal dynamic vertical ray shooting in rectilinear planar subdivisions," *ACM Trans. Algorithms*, vol. 5, no. 3, Jul. 2009, Art. no. 28. [Online]. Available: http://doi.acm.org/10.1145/1541885.1541889

[11] P. K. Agarwal, L. Arge, and K. Yi, "An optimal dynamic interval stabbing-max data structure?" in *Proc. 16th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2005, pp. 803–812. [Online]. Available: http://dl.acm.org/citation.cfm?id=1070432.1070546

[12] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Santa Clara, CA, USA: Springer-Verlag, 2008.

[13] P. Afshani, L. Arge, and K. G. Larsen, "Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model," in *Proc. 28th Annu. Symp. Comput. Geometry*, 2012, pp. 323–332. [Online]. Available: http://doi.acm.org/10.1145/2261250.2261299

[14] S. Rahul, "Improved bounds for orthogonal point enclosure query and point location in orthogonal subdivisions in R^3," in *Proc. 26th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2015, pp. 200–211. [Online]. Available: http://dl.acm.org/citation.cfm?id=2722129.2722144

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

[16] H. B. Acharya, S. Kumar, M. Wadhwa, and A. Shah, "Rules in play: On the complexity of routing tables and firewalls," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.

[17] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," in *Proc. IEEE INFOCOM*, Mar. 2005, pp. 2068–2079.

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

[19] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and eXpressive PAcket classification)," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2014, pp. 15–26.

[20] X. Sun, S. K. Sahni, and Y. Q. Zhao, "Packet classification consuming small amount of memory," *IEEE/ACM Trans. Netw.*, vol. 13, no. 5, pp. 1135–1145, Oct. 2005.

[21] H. Liu, "Efficient mapping of range classifier into ternary-CAM," in *Proc. 10th Symp. High Perform. Interconnects (HOTI)*, Aug. 2002, pp. 95–100.

[22] J. V. Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 4, pp. 560–571, May 2003.

[23] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2005, pp. 193–204.

[24] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary CAMs can be smaller," in *Proc. Int. Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, New York, NY, USA, 2006, pp. 311–322.

[25] D. Pao, Y. K. Li, and P. Zhou, "Efficient packet classification using TCAMs," *Comput. Netw.*, vol. 50, no. 18, pp. 3523–3535, Dec. 2006.

[26] A. Bremler-Barr and D. Hendler, "Space-efficient TCAM-based classification using gray coding," in *Proc. 26th IEEE Int. Conf. Comput. Commun. (INFOCOM)*, May 2007, pp. 1388–1396.

[27] H. Che, Z. Wang, K. Zheng, and B. Liu, "DRES: Dynamic range encoding scheme for TCAM coprocessors," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 902–915, Jul. 2008.

[28] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010.

[29] S. W. Bent, D. D. Sleator, and R. E. Tarjan, "Biased search trees," *SIAM J. Comput.*, vol. 14, no. 3, pp. 545–568, 1985.

[30] V. K. Vaishnavi, "Multidimensional balanced binary trees," *IEEE Trans. Comput.*, vol. 38, no. 7, pp. 968–985, Jul. 1989.

[31] J. Y. Hsiao, C. Y. Tang, and R. S. Chang, "An efficient algorithm for finding a maximum weight 2-independent set on interval graphs," *Inf. Process. Lett.*, vol. 43, no. 5, pp. 229–235, Oct. 1992.

[32] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.

Sorrachai Yingchareonthawornchai received the B.Eng. and master's degrees in computer engineering from Chulalongkorn University in 2012 and 2013, respectively. He is currently pursuing the Ph.D. student in computer science with Michigan State University. His research interests are algorithms, networking, and distributed systems.

James Daly received the B.S. degree in engineering and computer science from the Hope College in 2008. He is currently pursuing the Ph.D. degree in computer science with Michigan State University. His research interests are algorithms, networking, and security.

Alex X. Liu received the Ph.D. degree in computer science from The University of Texas at Austin in 2006. His research interests focus on networking and security. He received the IEEE & IFIP William C. Carter Award in 2004, the National Science Foundation CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He is an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING and the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING and an Area Editor of *Computer Communications*. He received Best Paper Awards from ICNP 2012, SRDS 2012, and LISA 2010.

Eric Torng received the Ph.D. degree in computer science from Stanford University in 1994. He is currently an Associate Professor and the Graduate Director with the Department of Computer Science and Engineering, Michigan State University. His research interests include algorithms, scheduling, and networking. He received the NSF CAREER Award in 1997.