A SORTED PARTITIONING APPROACH TO FAST AND SCALABLE DYNAMIC PACKET CLASSIFICATION

By

Sorrachai Yingchareonthawornchai

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science – Master of Science

2019

ABSTRACT

A SORTED PARTITIONING APPROACH TO FAST AND SCALABLE DYNAMIC PACKET CLASSIFICATION

By

Sorrachai Yingchareonthawornchai

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. 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 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 PartitionSort is an order of magnitude faster than TSS in classifying packets while achieving comparable update time. 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, PartitionSort is scalable to an arbitrary number of fields and requires only linear space.

ACKNOWLEDGEMENTS

I would like to express my great appreciation to Dr.Eric Torng, my research advisor, for his patient guidance, insightful critiques of my research. In particular, I am grateful for his flexibilty in that I can collaborate with researchers both inside and outside Michigan State University for diverse projects. I would like to thank Dr.Alex Liu for his support and guidance through this project. I appreciate Dr.James Daly for his time collaborating with me on experimental analysis. I would like to thank Jacob Marcus, a professorial assistant, for moving projects towards open-source community.

Finally, I am grateful for financial support from the department of Computer Science and Engineering at MSU. This work is partially supported by the National Science Foundation under Grant Numbers CNS-1318563, CNS-1524698, and CNS-1421407, and the National Natural Science Foundation of China under Grant Numbers 61472184 and 61321491, and the Jiangsu High-level Innovation and Entrepreneurship (Shuangchuang) Program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

KEY TO ABBREVIATIONS

ACL Access Control List

FW Firewall

GPU Graphics Processing Unit

IPC Internet Protocol Chains

OPEQ Orthogonal Point Enclosure Query

RIT Rank-Interval Tree

SDN Software-Defined Networking

TSS Tuple Space Search

TCAM Ternary Content Addressable Memory

VPP Vector Packet Processor

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

Packet classification is a fundamental building block for network architecture. A packet classifier provides an ability to distinguish packets into different flows for further action; such an ability is one of the core mechanisms of networking services and applications such as network security policy, network monitoring, routing protocols, quality of service measurement, and so on. As a result, if the packet classifier is not fast enough, then (millions of) incoming packets must be buffered and delayed. Extended packet buffering may disrupt and slow down the entire networking system.

Early packet classification research focused on static setting where preprocessing is permitted and updates are rarely required [1,2]. In this setting, we are only concerned with packet classification speed. In the preprocessing stage, we construct a data structure that represents a set of rules $\{r_j\}$ where each rule of a fixed dimension d can be represented as a catesian product of intervals $[a_i, b_i]^d$ for $i \leq d$, $0 \leq a_i \leq b_i$ and $a_i, b_i \in \mathbb{Z}_{\geq 0}$. A common setting is the task of classifiying a standard IP 5-tuple. A packet classifier extracts from the packet headers 5 fields: source IP, destination IP, source port, destination port, and protocol. (We can think of the values of 5 fields as a 5-tuple of non-negative integers $q = (q_1, q_2, \ldots, q_d) \in \mathbb{Z}^d_{\geq 0}$ in $d = 5$ dimensional space as a *query point*.) Given a query point q, the packet classifier finds a matching rule r_j such that $q_i \in [a_i, b_i]$, for all $i \in \{1, 2, \ldots, d\}$. If there is more than one matching rule, then we return the highest priority matching rule.

Many techniques have been introduced to improve packet classification in static setting. Given n rules, we can trivially scan rules in sequential order of priority to classify a packet in $O(dn)$ time using $O(n)$ space. Early research focused on getting to $O(\log n)$ classification time, assuming no memory restriction [3–5]. This naturally motivates a decision tree-based approach where rules are subdivided recursively based on some heuristics. The advantage is that the tree height is usually

much shorter than $\Theta(n)$. However, this approach usually necesssitates rule replications, requiring non-trivial amount of extraspace $O(n^d)$ in the worst case. Subsequent work focused on minimizing memory consumption using various techniques [6, 7]. These heuristics are experimentally shown to drastically reduce memory consumption with only moderate slow down in classification time. More recently, SAX-PAC [8] has shown a partitioning algorithm that can partition rules into a small number of sets of non-intersecting rules. The non-intersecting rules are a simpler instance of packet classification. This reduction is generic, and more importantly, does not incur extra space.

Software-Defined Networking (SDN) has shifted packet classification research by introducing new two unique challenges to packet classification: dynamicity and dimensionality. 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 [9, 10]. The advent of SDN leads to new challenges, dynamicity and dimensionality, which make rule update have similar importance to packet classification. Specifically, before SDN, classification rules were relatively static with infrequent updates. With SDN, rules are frequently inserted, modified or deleted by the controller to meet an applications' requirements such as establishing a new network path that satisfies a given quality of service. Thus, update time in addition to classification time must be minimized. For example, Open vSwitch [11] uses 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 [12]; 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. Furthermore, current software Open vSwitch implementations can handle up to $\sim 42,000$ rule updates/s [12], which requires the classifier's worst update time to be less than 10 μs, even for large rulesets. This rules out any classifier that has even millisecond update time.

For non-trivial dimensions $d = \Omega(\log n)$, no existing solutions can handle dynamicity and

fast classification at the same time. From the theory community, *prior algorithms on packet classification (also known as rectangle-stabbing query) and its priority variant are essentially linear in classification time*.

Kaplan *et al.* [13] presented a dynamic data structure for packet classification that uses poly-logarithmic query and update time while using $O(n\text{polylog})$ space. Afshani *et al.* [14] showed that rectangle-stabbing query cannot be dimensionally scalable with linear space; it requires $\Omega(\log n(\log n/\log d)^{d-2} + k)$ time to process a query. From the network community, *prior packet classification algorithms cannot satisfy both the dynamicity and scalability requirements*. Decision tree methods, such as HiCuts [3] and HyperCuts [4] are not scalable with the number of fields d and do not support fast update. In a decision tree, each branch or split point partitions the packet space with the goal to find a split where the rules are divided evenly. However, perfect splits are hard to find and any rule that crosses the splits must be copied into all partitions it intersects. This leads to inefficient updates as the insertion or deletion of a single rule, at a minimum, requires adding or deleting all copies of the affected rules. Furthermore, this may lead to inefficient splits compromising classification speed. Rule replication also lead to non-linear memory usage. Tuple Space Search (TSS) [15], an old packet classification algorithm, has been adopted in Open vSwitch [11] because it supports fast updates and its performance is relatively insensitive to the number of dimensions d. For each rule, TSS creates a d-tuple where each value represents the number of bits used in the corresponding field. TSS works by creating a hash table for each d-tuple. Unfortunately, while TSS has an update complexity of $O(d)$, its classification time is much slower than other existing packet classification methods because the number of hash tables is too large.

Current practices for SDN packet classification rely on caching, a technique that improves packet classification speed. Cache is a fast memory with limited capacity (typically SRAM) located near processing unit with orders of magnitude faster than the main memory storage (typically, DRAM). We can implement a simple standard cache by memorizing the signature of the packet query so that we can return the correct flow of the future packet that have the same signature. The effectiveness of caching depends heavily on locality of the reference, or patterns of flows. The lookup time can

significantly reduce if majority of the packet classification results are found in the cache.

We argue that it is not sufficient to rely on caching in the long run. In other words, we should treat locality of reference as a bonus for packet classifier whenever traffic has high locality. Otherwise, the packet classifier performance will go back to the full look-up mode in the worst case. Locality of reference is more challenging to preserve when there are many flows. Unlike IP lookup, packet classification cache size must scale with the number of flows (i.e., number of rules) to maintain high locality [1,16]. More complex network applications also contribute to a variety of traffic flows, and hence less locality for the cache to exploit. One (expensive) solution is to use more hardware to maintain high locality; this is clearly not sustainable in the long run because of the cost.

Our goal is to develop an efficient dynamic packet classifier that works well even with high dimensionality. More specifically, a packet classifier must provide high-speed packet classification and fast-rule-update at the same time. In other words, the packet classifier cannot assume offline construction and preprocessing. At the same time, the packet classifier must scale well with high dimensionality and does not rely on cache. Finally, we must use only linear space with respect to the number of rules.

1.2 Proposed Approach: an Overview

A key element of our approach is to a partition an arbitrary ruleset into *sortable* rulesets. Partitioning to sortable rulesets, which can be stored in binary search trees and thus provide both fast classification and fast update, separates our approach from previous partitioning strategies that partition to decision trees, which do not support fast update, or non-overlapping rulesets, which do not have a natural data structure that guarantees fast classification or update.

The intuition is that sorting one-dimensional rules is simple since they are only intervals in one dimension. The basic idea is that we choose a field order (permutation) $\vec{f}$ and then compare two rules by comparing their projections in the fields (*i.e.*, dimensions) sequentially. For example, in Figure 1.1, if we choose field order XY, then $R_1 \leq_{XY} R_4$ because R_1's projection in field X, which is $[1,3]$, is smaller than R_4's projection in field X, which is $[5,7]$; but if we choose field order YX,

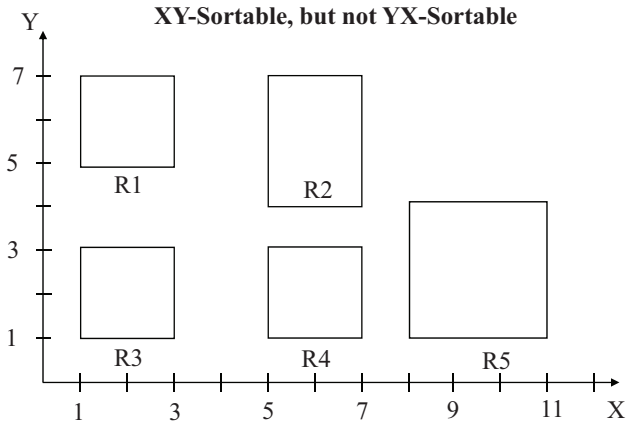

Figure 1.1: Sortable ruleset with $\leq_{xy}$

then $R_4 \leq_{XY} R_1$ because R_4's projection in field Y, which is $[1,3]$, is smaller than R_1's projection in field Y, which is $[5,7]$. As another example, $R_4 \leq_{XY} R_2$ because although the first field X cannot separate them apart as their projections in field X are both $[5,7]$, field Y can separate them apart as R_4's projection in field X, which is $[1,3]$, is smaller than R_2's projection in field X, which is $[5,7]$.

1.2.1 Sortable Ruleset Partitioning

Given an arbitrary ruleset, we partition it into a collection of sortable rulesets. Note the field orders for each partition might differ. We note many previous packet processing approaches such as EffiCuts [6] and SmartSplit [7] have also partitioned rulesets to try and deal with the replication of rules that is inherent in the popular decision tree data structures used by many packet processing solutions. While their partitioning has reduced rule replication, the main benefit has been to reduce the memory explosion that occurs without partitioning. Because the final data structure is a decision tree, the resulting partitions still do not support fast update as there is still rule replication (even if reduced); their solutions are not appropriate for dynamic SDN security policies. Alternatively, SAX-PAC [8] partitions rulesets into nonoverlapping rulesets. Unfortunately, there is no natural data structure for storing nonoverlapping rulesets so that packet classification can be performed quickly. In essence, SAX-PAC can guarantee fast classification for only two dimensional rules because they have not targeted an appropriate data structure to perform fast classification. Because we partition

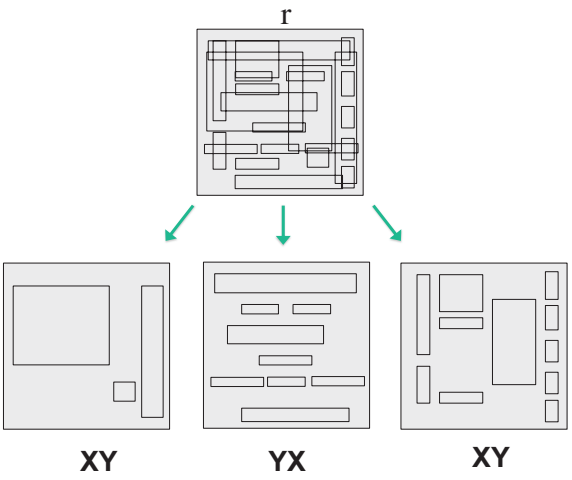

Figure 1.2: Examples of sortable ruleset partitioning with field order XY, YX, and XY.

to sortable rulesets, we overcome the limitations of prior art in that we can classify packets and update rules in the resulting data structure in $O(\log n)$ time using binary search trees. An example of sortable partitioning can be seen in Figure 1.2. Our goal in partitioning is to minimize the number of resulting partitions because we must search each partition in the worst case. We can view partitioning from both an online and an offline perspective. In the online perspective, we have an existing partitioning of the rules and must add or remove a rule from the existing partitions. In the offline perspective, we have a set of rules that must be partitioned as we see fit. The offline problem is important even for SDN as the system may periodically repartition the ruleset to improve performance.

1.2.2 Rank-Interval Tree (RIT)

Next, we describe an efficient data structure that potentially supports $O(d+\log n)$ rule insertion, rule deletion, and packet search. Because comparing two rules requires $O(d)$ time, a standard balanced binary search tree would only guarantee $O(d \log n)$ time rather than $O(d + \log n)$ time. There are trees that support these operations in $O(d+\log n)$ time [17–20]. Our contribution is that we propose a new tree balancing technique for RIT that yields the same optimal running time.

We propose a candidate data structure called Rank-Interval Tree RIT where nodes are standard

binary search tree nodes augmented with a "next field" pointer. RIT works with the unique intervals in each field. For example, consider field $\vec{f}_1$ and the corresponding n intervals of the rules in sorted order. Suppose the first $n/2$ intervals are identical and the remaining $n/2$ intervals are unique. Then this rule set has $n/2 + 1$ unique intervals in field $\vec{f}_1$; the first interval has cardinality $n/2$ and the remaining $n/2$ intervals have cardinality 1. This example is depicted in Figure 1.3.

How should RIT store these unique intervals? The key observation is that it must take into account both cardinality and balancing. If the cardinality of a node is sufficiently high, it must be high in the tree. Consider our running example in Figure 1.3; if it only balances and ignores cardinality, the first interval with cardinality $n/2$ will be at depth $\log n/2 + 1$; if this pattern repeats in later fields, this produces a tree with depth $\Theta(d \log n)$. If there are no high cardinality nodes, then it should be a standard balanced binary search tree, but we achieve this by combining nodes and focusing just on cardinality.

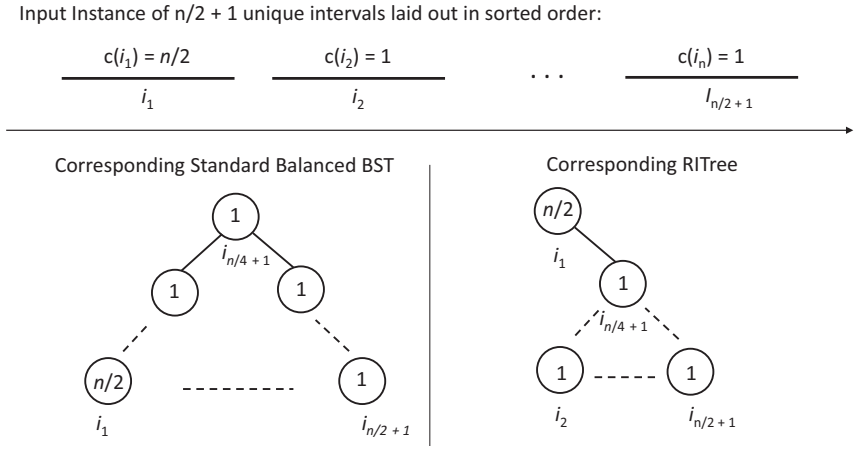

Figure 1.3: Example illustrating need for RIT. Input instance consists of $n/2 + 1$ unique intervals where smallest interval in sorted order has cardinality $n/2$. The corresponding standard balanced binary search tree has high cardinality interval at depth $\log n/2 + 1$ whereas RIT has high cardinality interval at depth 1. Note the actual RITree would have spanning nodes (defined in Section 4.2.1); we omit here to highlight intuition.

1.3 Highlights of Results

We compare PartitionSort to TSS because it offers the best combination of classification time and update time and thus is used by Open vSwitch [11]. We also compare with SmartSplit as a

state-of-the-art traditional packet classifier that ignores update.

Experimental Setup: We generate rulesets using ClassBench [21] which mimics three ruleset types: access control lists (ACL), firewalls (FW), and IP Chains (IPC). We generated rulesets with 1k, 2k, 4k, ..., 64k rules; here we show results using the 1k, 4k, 16k and 64k rulesets to improve readability. For each size and seed, we generated 20 different rulesets and report the average classification time and rule update time.

Metrics: We compared these algorithms using four metrics: classification time, update time, space, and construction time. We studied the scalability of the algorithms on these metrics with respect to the number of rules n and the number of fields d.

Summary: *Our results show that PartitionSort is uniquely competitive with both TSS in update time and SmartSplit in classification time. In particular, PartitionSort performs updates in 0.65 µs on average which is comparable to that of TSS while SmartSplit requires 100s of seconds to reconstruct some classifiers. At the same time, PartitionSort classifies packets in 0.29 µs on average which is comparable to that of SmartSplit while TSS requires an order of magnitude more time.* We show below a comparison of PartitionSort and TSS focusing on classification time and update time versus the number of rules.

Number of Colors: We first compare the number of colors required by PartitionSort and PTSS as this helps determine the number of partitions searched and largely explains why PartitionSort outperforms PTSS. *PartitionSort uses an order of magnitude fewer colors than PTSS.* For ACL, FW, and IPC rulesets with 32k rules, the number of colors used by PartitionSort and PTSS are on average 16.3 vs. 241.4, 22.3 vs 99.2, and 9.5 vs. 170.9, respectively. This suggests PartitionSort will classify rules an order of magnitude faster than PTSS.

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 1.4 (a) shows the average classification times for both PartitionSort and TSS across all ruleset types and sizes. *Besides being much faster than TSS, PartitionSort is also much more consistent than TSS.*

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.* On average, PartitionSort's update time is only 1.7 times larger than TSS's update time. Figure 1.4 (b) shows the average update times across all ruleset types and sizes for both PartitionSort and TSS. This result is consistent with the fact that TSS is optimized for fast update.

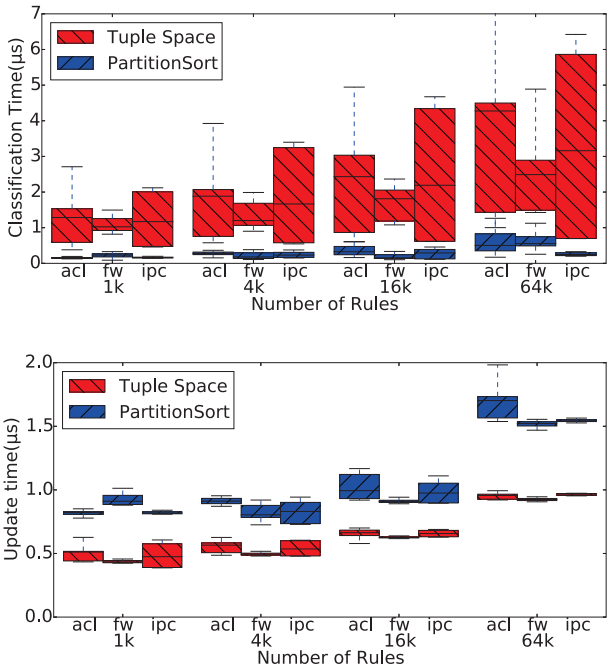

Figure 1.4: PartitionSort and PTSS classification times and update times.

1.4 Recent Development

After PartitionSort, there are a few more packet classification algorithms that focus on update time. TupleMerge [22] is a new partitioning-based packet classifier that improves TSS by merging compatible tuples in TSS in a less restrictive way than that in TSS. This approach reduces a

significant number of tuples, and so improves the overall classification and update speed. In comparison to PartitionSort, TupleMerge may have more partitions than that in PartitionSort, but each partition is represented as a hash table, which is faster than a binary search tree. Therefore, it is possible to have more hash tables, and still result in slightly faster overall classification time. CutSplit [23] is a new decision-tree packet classifier that improves previous decision tree techniques by exploting the cutting and splitting techniques adaptively. The key development is that CutSplit manages to dynamize decision trees that are known to be hard to handle rule dynamicity. Being unaware of our work, Yang et al [24] propose an improved version of aggregated bit vector scheme where they replace bit vectors and make the bitwise 'AND' operation of Bloom filters with the goal to improve both classification, and update speed. However, there is no comparison with PartitionSort and TupleMerge in their work.

1.5 Future Work

There are two main directions for future work. First direction is to extend our algorithm to work with batched processing. Second is to integrate our framework into open-source OpenFlow switches.

1.5.1 Batched Processing

Given a batch of packets, we can classify them using the basic idea of mergesort. Namely we do rounds of sorting packets and then merging them with sorted rule. With batched processing, we can achieve packet classification using amortized O(1) time per packet if we use a linear time sorting algorithm such as radix sort. The algorithm is fully compatible with our sorted-partitioning approach which supports both single-query and batched query classification.

1.5.2 Integration with OpenFlow switches

We will integrate our ideas into two open source OpenFlow switches: Lagopus and Open vSwitch. We have already done a preliminary integration of 1 and 2 with Lagopus. However, Open vSwitch is

more widely used. For this project, we propose to integrate all of our key ideas into Open vSwitch. There are two key challenges that we must overcome to successfully integrate our ideas into Open vSwitch. First, we must generalize PartitionSort to handle rules with arbitrary bitmasks. Second, we must develop caching schemes for PartitionSort similar to the megaflow caching scheme that Open vSwitch currently uses with its Tuple Space Search packet classifier.

CHAPTER 2

RELATED WORK

From the theory community, prior work is mostly on Orthogonal Point Enclosure Query. From the networking community, prior work falls into two main categories: software based and hardware based.

2.1 Orthogonal Point Enclosure Query

From computational geometry, we do not have any known results that give us dynamic d-scalable algorithms for any OPEQ variants. We do know that OPEQ with $d = 1$, which is also known as the interval stabbing query problem, can be solved in $O(\log n)$ time using linear space in the dynamic setting. For reporting and counting variants, there are several ways to achieve this result [25, 26]. For the priority variant, optimal $O(\log n)$ dynamic solutions have been given in [27, 28]. We can extend d-dimensional solutions to $(d + 1)$-dimensional solutions using segment trees at a cost of a multiplicative factor of $O(\log n)$ in both space and time [29]. If we restrict boxes to be disjoint or nested, Kaplan *et al.* [30] eliminate the multiplicative factor of $O(\log n)$ for queries; however, space and updates still require the multiplicative factor of $O(\log n)$. Finally, the general technique for handling the dynamic setting is to start with a good solution for the static case and then use general transformation techniques with additional cost of space or time [31–33].

2.2 Software Based Packet Classification

Most prior software based packet classification algorithms are based on decision trees; they differ in how to construct the tree. Representative ones are HiCuts [3], HyperCuts [4], and HyperSplit [5]. These methods generally produce $O(\log n)$ classification times at the potential cost of superlinear space due to rule replication. HiCuts and HyperCuts partition packet space with multiple cuts at each tree node with the goal to reduce tree height; however, this may lead to greater rule replication. HyperSplit splits packet space with just one cut at each tree node leading to less rule replication at

the potential cost of greater tree height. However, HyperSplit still suffers from rule replication and thus requires superlinear space. To mitigate the effects of rule replication in decision tree methods, EffiCuts [6] and SmartSplit [7] use multiple decision trees. They categorize rules by whether they are "small" or "large" in each field and label rules as conflicting if their vectors of small and large are not identical. For EffiCuts, there would be 2^d possible decision trees where each tree is a HyperCuts tree. For SmartSplit, there are only 4 possible decision trees as they specialize to traditional 5-field classification and only apply this categorization to the source and destination IP fields. SmartSplit analyzes the resulting rules and then chooses between HyperSplit and HyperCuts. Although EffiCuts and SmartSplit reduce rule replication and thus limit the super-linear memory usage, they do not eliminate replication and thus rule updates are still problematic; furthermore, they do not scale effectively to $d > 5$. Kogan *et al.* proposed SAX-PAC [8] where they partition a ruleset into sets of non-intersecting rules, which means that each partition is essentially a PLOS instance, and PLOS is not known to be d-scalable. Such partition criteria are too relaxed resulting in partitions that do not guarantee fast classification or update. Because decision tree methods do not support fast updates, Open vSwitch [11] uses Tuple Space Search (TSS) [15] for packet classification. TSS achieves fast updates because it can quickly identify which tuple corresponds to a given rule. Each partition is then implemented using a hash table on the tuple meaning rule insertion and deletion takes $O(d)$ expected time; the time is $O(d)$ rather than $O(1)$ because all relevant fields must be extracted and hashed. The limitation is that the definition of tuples is too strict resulting in too many partitions. Some techniques have been proposed to improve performance such as performing a search on specific fields in $O(\log n)$ time to identify eligible tuples [15]. However, these techniques complicate rule update, and Open vSwitch has not implemented these optimizations.

2.3 Hardware Based Packet Classification

Most prevalent hardware based packet classification solutions are based on Ternary Content Addressable Memory (TCAM). A TCAM chip takes as input a search key and then uses hardware circuits to compare the input search key with all of its occupied entries in parallel. It then uses a

priority encoder to identify the index (or contents if desired) of the first matching entry. This all takes place in constant time (*i.e.*, a few clock cycles). The CAM is ternary because each entry consists of an array of 0's, 1's, or *'s (*don't-care* values). A packet header (*i.e.*, a search key) matches a TCAM entry if their corresponding 0's and 1's match. TCAM has several limitations. First, TCAM chips have limited capacity. The largest available TCAM chip has a capacity of 72 megabits (Mb), while 2Mb and 1Mb chips are the most popular. Second, TCAM chips consume a large amount of *power* due to their parallel searching. The power consumed by a TCAM chip is about 1.85 Watts per megabit (Mb) [34], which is roughly 30 times larger than a comparably sized SRAM chip [35]. These problems are exacerbated when we consider OpenFlow packet classification with more fields and wider fields such as IPv6 IP addresses. As a result, while TCAM is used in some OpenFlow implementations, TCAM is not a scalable solution that truly meets the demands of OpenFlow packet classification. Kuźniar *et al.* also show how the data plane can lag behind the control plane by up to 400 ms in some TCAM implementations of OpenFlow [12]. Although TCAM space can be minimized using various TCAM optimization algorithms [36–59], once optimized, it is extremely inefficient (often in terms of minutes or hours) to perform updates as each update typically cause the whole TCAM table to be recomputed from scratch. Recently, GPU [60, 61] and FPGA [62, 63] based packet classification solutions have been proposed. These algorithms are not suitable for OpenFlow packet classification because they are very inefficient for updating.

2.4 Batch Processing

Batch processing has become commonplace in lower-level building blocks of networking stack. Netmap [64] is a framework for fast packet I/O, available in Linux, FreeBSD. Netmap reduces overhead from processing one packet at a time. The saving gained by a large batch comes from removing per-packet dynamic memory allocations, system call overheads, amortized over large batches. Intel's DPDK is an open-source project that aims for achieving high I/O performance, and high packet processing rates. The key advantage of DPDK is that it is a userspace application that interacts directly to the network hardware without passing through Linux kernel networking stacks.

In addition to the low-level networking stack, batch processing technique is brought to high-level networking functionality. Vector Packet Processor (VPP) [65] architecture provides layer 2 and 3 networking functionalities based on low-level building block such as DPDK. The unique feature for VPP is that VPP extends the batching from low-level building block to high-level packet processing functions. The network function can be represented as a tree. Instead of processing one packet at a time, VPP process a batch of at most 256 packets at once before moving on the next node.

We now turn attention to batching in packet classification. Given a batch of packets, we can process all packets the batch in parallel, and thus we can achieve significant performance speedup. A common massively-parallel processor is Graphics Processing Unit (GPU). PacketShader [66] exploits parallelism from GPU to speed up the software-router by introducing high-performance packet I/O engine at the operating system level and offloading packet batches to GPU. However, they show speed up over linear search classifier. They did not show if GPU can actually speed up over more advanced packet classifier. Recently, [67] design a new GPU-based algorithm that features improved memory access pattern to reduce latency to off-chip memory. Furthermore, they showed significant speedup over three types of packet classifiers: linear search, Bloom-filter, and Tuple Space search classifier.

One major drawback for batched processing is high latency. [66], and [67] reported latency in range between 200 microseconds to 1000 microseconds depending on packet size and classification algorithms. This high latency was because GPU requires large batches to maximize throughput. To obtain large batch, we need to wait for packet to queue up to the desired batch size. Moreover, with GPU we must deal with offloading packets from host CPU to GPU, this requires careful implementation of GPU using known techniques such as latency hiding.

CHAPTER 3

PARTITIONSORT

3.1 Introduction

3.1.1 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 [9, 10].

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 [11] 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 [12]; 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,

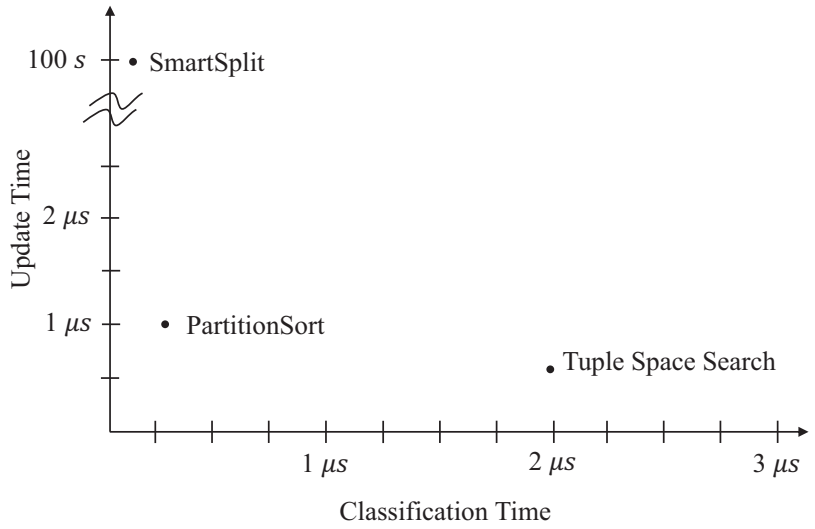

Figure 3.1: PartitionSort is uniquely competitive with prior art on both classification time and update time. Its classification time is almost as fast as SmartSplit and its update time is similar to Tuple Space Search.

the number of header field support is 45. Recently, *P*4 has been introduced to support programming protocol independent packet processing [10]. 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.

3.1.2 Limitations of Prior Art

One of the fundamental challenges in packet classification is to simultaneously achieve high-speed classification and fast update. Most existing methods such as SmartSplit [7] have focused on minimizing classification time while sacrificing update time and memory footprint while some methods like Tuple Space Search (TSS) [15] 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 3.1.

SmartSplit [7] is the state-of-the-art decision tree method as it builds upon the previous best methods HyperCuts [4], EffiCuts [6], and HyperSplit [5]. 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 [12].

TSS [15] used in Open vSwitch [11] 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.

3.1.3 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 3.1. 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.

3.1.4 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.

3.1.5 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 3.2. We then define ruleset sortability in Section 3.3. We then describe an Multi-dimensional Interval Tree in Section 3.4, an offline partitioning scheme in Section 3.5 and the full PartitionSort algorithm in Section 3.6. We provide theoretical analysis of PartitionSort in both worst-case and average case in Section 3.7. We then describe an optimization based on Rule-Splitting to group incompatible rules in the same partition in Section 3.8. We then give our experimental results in Section 3.9 and conclude the paper in Section 3.10.

3.2 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 [27, 28]. 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 [29]. Unfortunately, using linear space, any data structure requires $\Omega(\log n(\log n/\log d)^{d-2} + k)$ time to process a query [14]. 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 [68] 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 [2, 69]. In particular, if we restrict the structure of rulesets given by ClassBench [21], 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 [7], HiCuts [3], HyperCuts [4], and HyperSplit [5] 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 [15] and SAX-PAC [8] 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 [70], EffiCuts [6] and SmartSplit [7] 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 [38–42, 45, 71, 72]. 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.

3.3 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 3.2; 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.

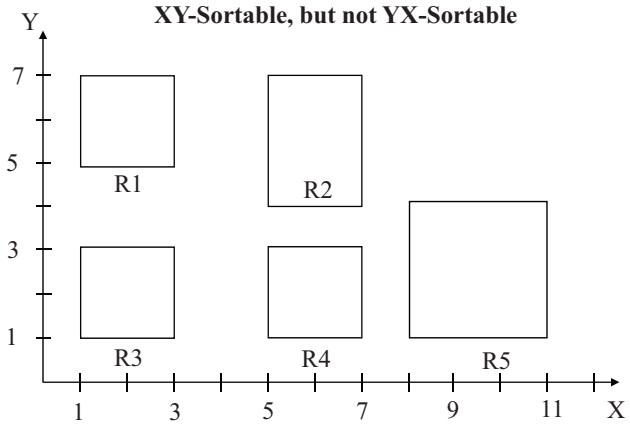

Figure 3.2: Example of ordered ruleset with $\leq_{xy}$

3.3.1 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 3.2 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 $\vec{f}$.*

3.3.2 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 3.3.1. *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 $p >_{\vec{f}} r_j$. □

3.4 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 3.3 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.

	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]

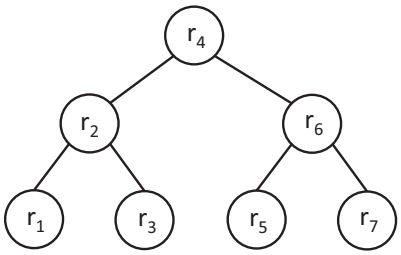

Figure 3.3: A sortable ruleset with 7 rules and 3 fields, field order $\vec{f} = XYZ$, and the corresponding simple binary search tree where each node contains d fields.

There exists a multi-key binary search tree supporting $O(d + \log n)$ dynamic time for the multi-key dictionary problem [73, 74].

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 3.4.1. *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 3.4 shows an MITree for the ruleset in Figure 3.3. 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 3.4. For the rest of this paper, we use MITree to refer to a compressed MITree.

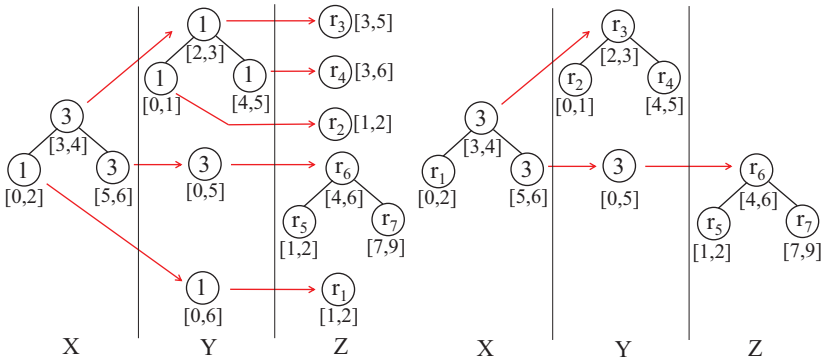

Figure 3.4: We depict the uncompressed (left) and compressed (right) MITree that correspond to the sortable ruleset from Figure 3.3. Regular left and right child pointers are black and do not cross field boundaries. Next field pointers are red and cross field boundaries. In the compressed MITree, nodes with weight 1 are labeled by the one matching rule.

3.5 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 $1 \leq i \leq \chi$ where $\bigcup_{1 \leq i \leq \chi} R_i = R$, $R_i \cap R_j = \phi$, 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 $1 \leq i \leq d!$, 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 $1 \leq j < k \leq n$. We then define the interval multi-graph $G = (V, \bigcup_{1 \leq i \leq d} E_i)$ where each edge is labeled with the field order that it corresponds to. Our

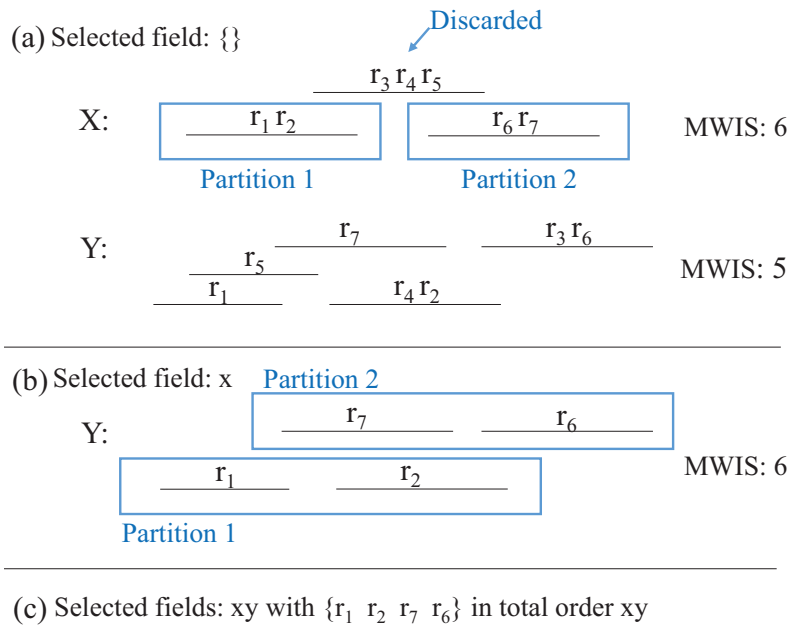

Figure 3.5: Example execution of GrInd.

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 $1 \leq i \leq d!$; 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 [75] 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 [75]) 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 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. The running time is $O(d^2 n \log n)$ as shown in Theorem 3.5.1.

We give an example of GrInd in action in Figure 3.5. The input set of boxes with two fields is shown in Figure 3.5 (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 3.5 (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 3.5.1. *GrInd computes a field order $\vec{f}(i)$ for $1 \leq i \leq d!$ 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 $1 \leq i \leq d$, 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. □

3.6 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.

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 $1 \leq i \leq j \leq t$, $pr(T_i) > pr(T_j)$. To maintain this sorted order, PartitionSort uses the following data structures. First, a lookup table $M : R \rightarrow \mathcal{T}$ 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.

3.6.1 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 $|T_i| \leq \tau$ for some small threshold such as $\tau \leq 10$, we run GrInd (cf. Section 3.5)

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 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.

3.6.2 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 $1 \leq i \leq t$ 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 [11] 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 3.1 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.

Table 3.1: 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		
	$\|G\|$	$\|O\|$	$\|T\|$	%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

3.7 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 3.2. 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, $n_i \geq n_j$ for $1 \leq i < j \leq b$, $\sum_i n_i = n$, and $\tilde{n} = \prod_{i=1}^{n} n_i^{1/b}$ is the geometric mean of n_i.

Table 3.2: 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	$O(d + \log n)$	$O(d + \log n)$	$O(d + \log n)$

3.7.1 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 3.7.1. *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 $i \neq 1$, then $\tilde{n} \leq n^{1/b}$ 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 3.7.1. *The search time and rule insertion time for PartitionSort is in the range $O(db + \log n)$ and $O(db + b\log n)$.*

3.7.2 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 $s_i \leq n/\gamma^i$ until $s_i \leq \tau$ where γ and τ are constants > 1.*

We show the r_i values for our 64k ACL, FW, and IPC rulesets in Figure 3.6. 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.

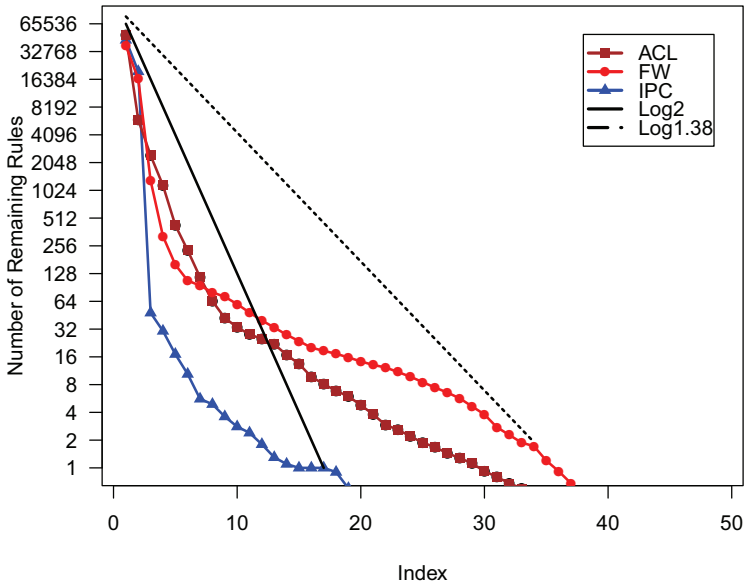

Figure 3.6: Average number of remaining rules (r_i) plotted in log scale with base 2 and 1.38. The average is across from partition at i index including those with zeros.

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

Lemma 3.7.1. *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 3.7.2. *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 3.7.1 and Lemma 3.7.1. Specifically, from Theorem 3.7.1, we get that the number of comparisons is $O(db + b \log \tilde{n})$. From Lemma 3.7.1, 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 [8]. Another option is to use a standard decision tree such as a SmartSplit tree. Finally, in Section 3.8, we propose a rule-spliting optimization that allows us to further reduce the number of sortable rulesets required.

3.7.3 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 3.6.2 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 3.7 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 3.1), 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 3.7.2. *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 $O(d + \log n)$.*

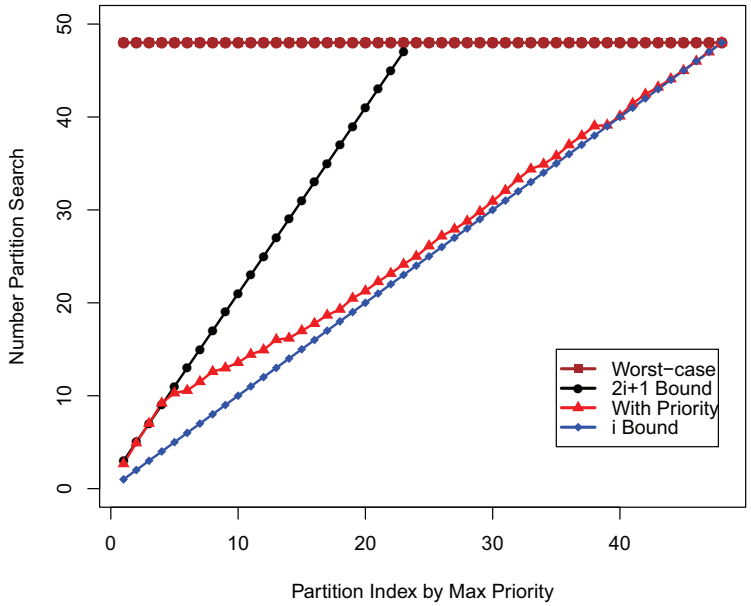

Figure 3.7: Plot of $\Delta(T_i)$ for our ACL, FW, and IPC 64k rulesets. The worst-case is the maxmimum number of partitions over all rulesets of size 64k. The average excludes missing values.

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. □

3.8 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.

For example, consider the rules in Figure 3.2. In YX order, only r_1, r_3, and r_4 are compatible.

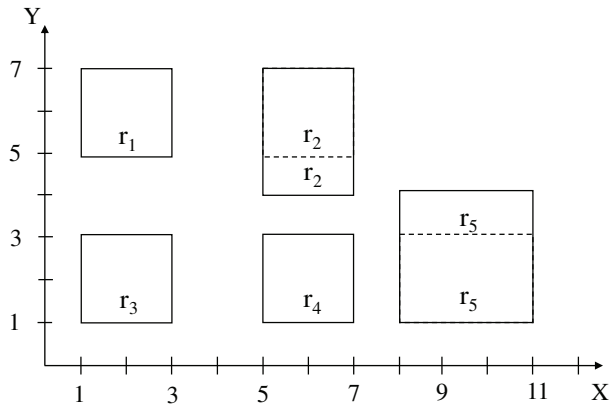

Figure 3.8: Splitting the rules from Figure 3.2 in YX order.

If we cut r_2 at $Y = 5$ and r_5 at $Y = 3$ as shown in Figure 3.8, 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).

3.8.1 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 3.2, 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 3.8 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.

3.8.2 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).

3.8.3 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.

3.9 Experimental Results

3.9.1 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.

3.9.1.1 Rulesets

We use the ClassBench utility [21] 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 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.

3.9.1.2 Tuple Space Search

Our TSS implementation is the one from GitHub described in Pfaff *et al.* [11]. 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.

3.9.1.3 SmartSplit

We use an implementation that is written by the authors of the paper which is hosted on GitHub[1].

3.9.1.4 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.

3.9.1.5 Caching

Most of our experiments are done without flow caching which can significantly speed up OpenFlow packet processing. We do some experiments where we compare PartitionSort and TSS with flow caching focusing on the microflow and megaflow caching used in Open vSwitch [11]. 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.

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

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.

3.9.1.6 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[2] the average metric value, and, in some cases, a box plot.

3.9.1.7 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.

3.9.2 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.

3.9.2.1 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 3.9 shows the

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

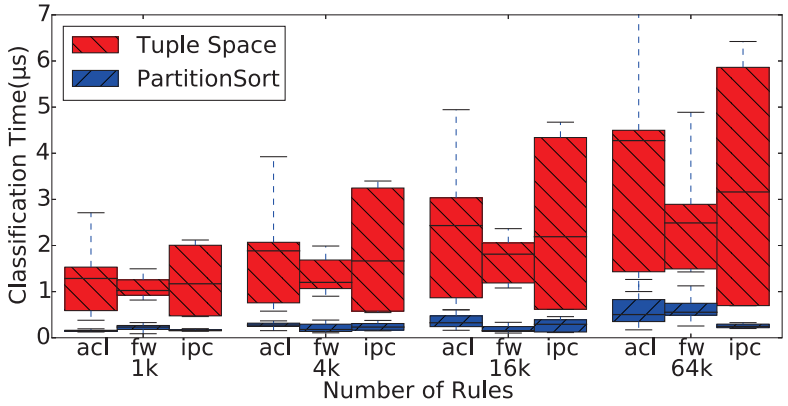

Figure 3.9: Classification Time by Ruleset Type and Size

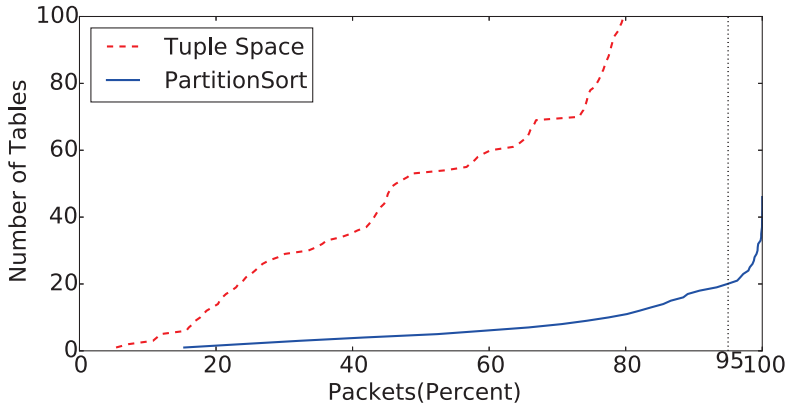

Figure 3.10: Partitions/Tuples Queried. A CDF distribution of number of partitions required to classify packets with priority optimization (cf. Section 3.6.2).

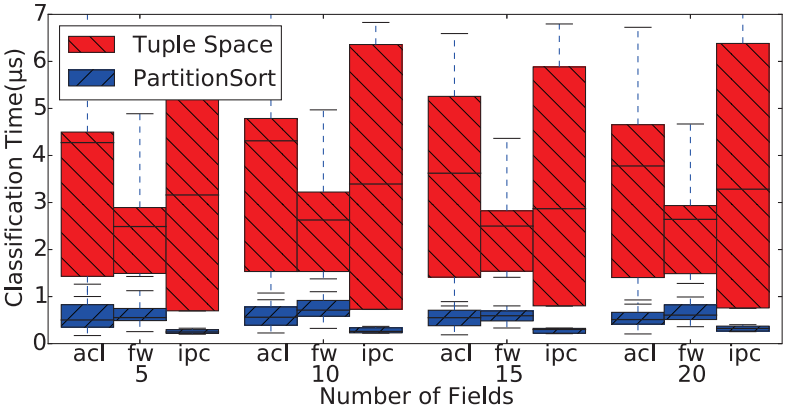

Figure 3.11: Classification Time on 64K Rules

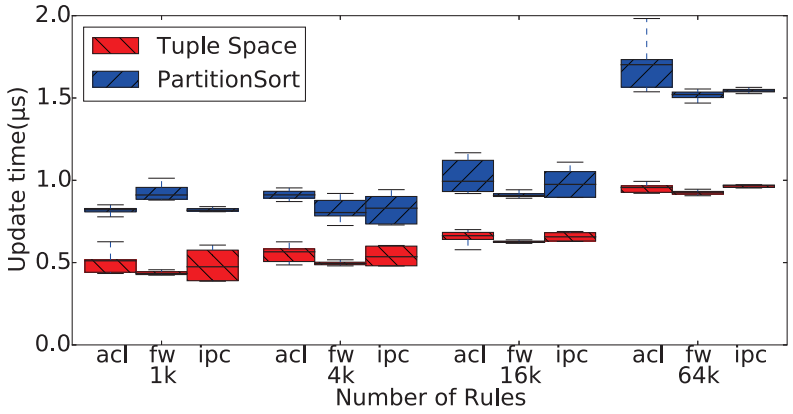

Figure 3.12: Update Time vs Tuple Space Search

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.

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 3.9.

The reason for these factors is that PartitionSort requires querying significantly fewer tables than TSS. As seen in Figure 3.10, 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 3.11. 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.

3.9.2.2 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 3.12 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 3.12 does show that PartitionSort has an $O(\log n)$ update time.

3.9.2.3 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.

3.9.2.4 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.

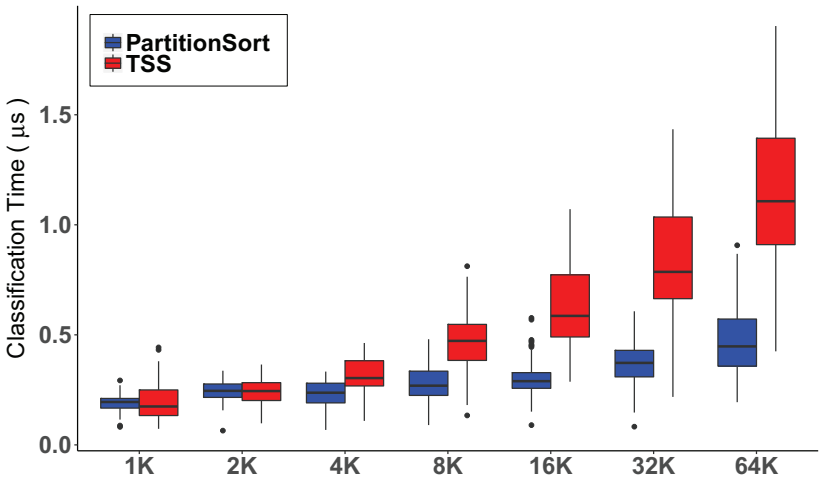

Figure 3.13: Classification Time with Caching by Ruleset Size

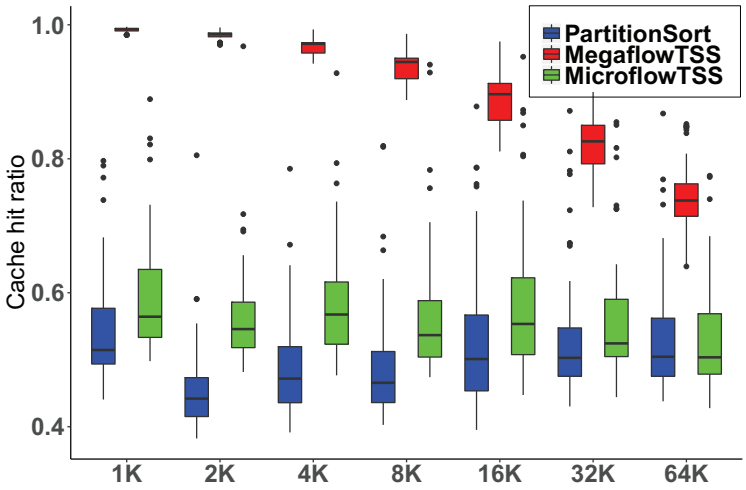

Figure 3.14: Cache-Hit Ratio by Ruleset Size

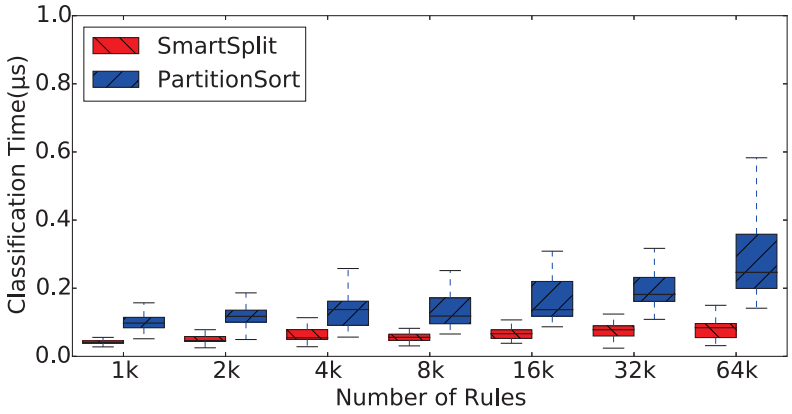

Figure 3.15: Classification Time vs SmartSplit

3.9.3 PartitionSort versus TSS with Caching

Our classification time results with and without caching, depicted in Figures 3.13 and 3.9, 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 (see Figure 3.14). The hit ratio for just the microflow cache for both methods is roughly 54% for all ruleset sizes whereas the hit ratio for TSS with microflow and megaflow caches ranges from 76% for the 64K classifiers to 99% for the 1K classifiers. Thus, as classifier size increases, TSS must classify more packets.

3.9.4 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.

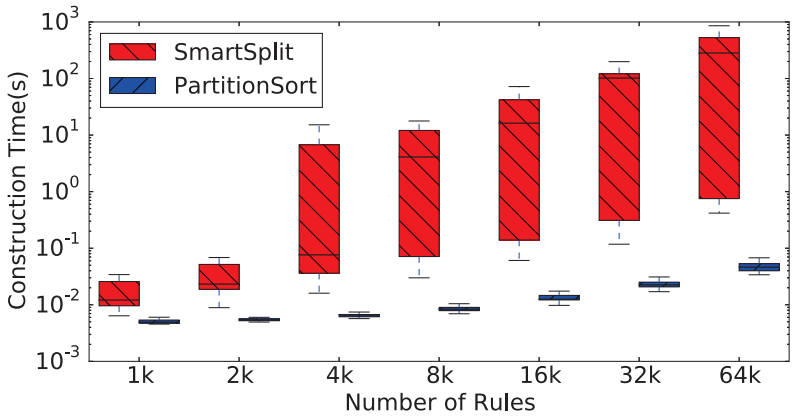

Figure 3.16: Construction Time in logarithmic scale.

3.9.4.1 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 3.15. 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.

3.9.4.2 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 3.16. Additionally, SmartSplit does not support updating. Together, this means that SmartSplit is not appropriate for cases where the ruleset is updated frequently.

3.9.4.3 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

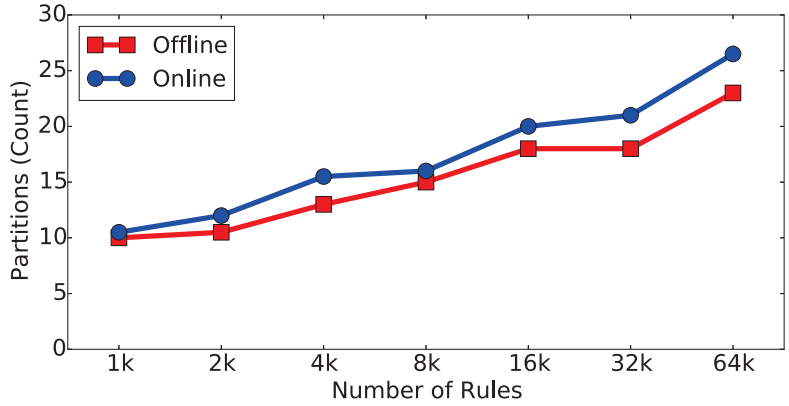

Figure 3.17: Number of Partitions for Offline and Online PartitionSort

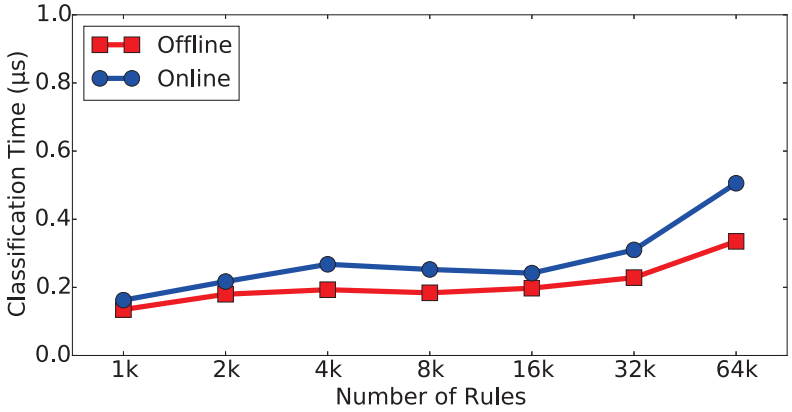

Figure 3.18: Classification Time for Offline and Online PartitionSort

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.

3.9.5 Comparison of Partition Algorithms

We now compare our offline (cf. section 3.5) and online (cf. section 3.6) partitioning algorithms. Figure 3.17 shows the number of partitions required by each version.

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

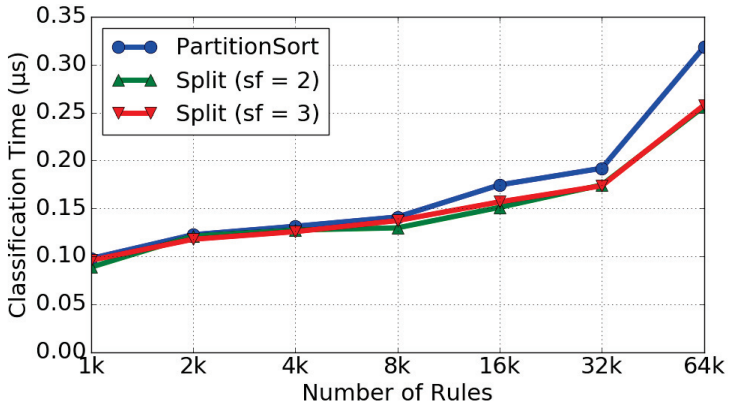

Figure 3.19: Classification Time for Splitting Experiments

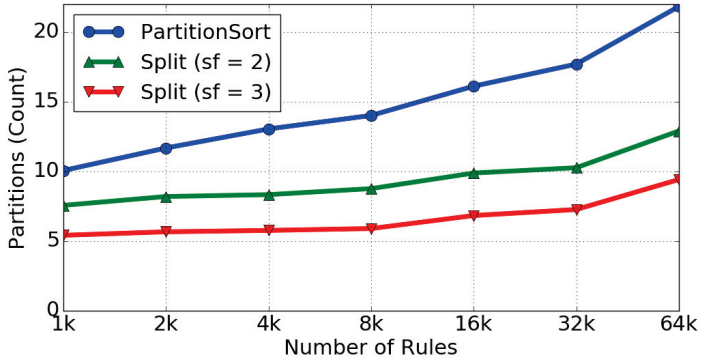

Figure 3.20: Number of Partitions for Splitting Experiments

as seen in Figure 3.18. We conclude that a single offline construction at the beginning with online updates should be more than sufficient for most purposes.

3.9.6 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 $sf = 3$.

3.9.6.1 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 3.19.

The improvement in classification time is at least partially caused by a reduction in the number of tables required, as seen in Figure 3.20. 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.

3.9.6.2 Drawbacks of Splitting

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

3.10 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.

CHAPTER 4

RANK-INTERVAL TREE

4.1 Background and Introduction

We define the problem of maintaining dynamic ordered set with lexicographic ordering. Each key is a tuple of d fields, $u = (u_1, u_2, \ldots, u_d)$ where u_i is from an ordered universe U. Two keys can be compared by lexicographically by field. That is, given, key u and v, $u < v$ if there exists a number $i \leq d$ such that $u_i < v_i$, but $u_j = v_j$ for $1 \leq j < i$. The goal is to design a data structure that supports search, insert, delete operations in $O(d + \log n)$ time where n is a number of keys. Although this problem is well-studied with numerous data structures with matching bounds have been proposed, we introduce an alternate data structure with the same matching bounds. Our key novelty is the new balancing technique that is different from known techniques. We start with basic idea for naive implementation of multidimensional binary search tree. Then, we propose a novel data structure that we call Rank-Interval Tree. We will describe definition of RIT and its static construction. Then, we will describe how to handle dynamic insertions and deletions in $O(d + \log n)$ time.

4.1.1 Naive Multi-dimensional Binary Search Tree

We start by defining a basic structure $T(n, d)$ which is d-dimensional binary search tree as follows. In the first dimension, we group distinct keys and construct a $d-1$-dimensional subtree recursively based on the next dimension. Hence, each distinct key represents a $(d-1)$-dimensional binary search tree. So, the tree $T(n, d)$ is a standard binary search tree where each node has a third pointer which points to $(d-1)$-dimensional subtree. The process repeat until the final dimension which is basically an ordinary binary search tree. Next, we describe how to search for a d-dimensional query point q. Starting with $i = 1$, we do a binary search using left and right child pointers for a node v such that $q_i = v_i$. If no such node v exists, then q is not matched. Otherwise, we follow the third pointer to the next field and repeat the binary search. If $i = d$ and we find a match, we report

the key that matches with the query q.

The problem with this construction is that the worst-case query time is $\Theta(d \log n)$ even if every subtree is balanced as shown in the following Proposition:

Proposition 4.1.1. *The naive binary search tree has $\Theta(d \log n)$ worst case query time given a d-dimensional input instance with n keys.*

Proof. The upper bound of $O(d \log n)$ follows from the fact that there are d fields and each field has a binary search tree which requires $O(\log n)$ time to process a query. We now show the lower bound. We start with field $\vec{f_1}$ and the corresponding n intervals of the boxes in sorted order. Suppose the first $n/2$ keys are identical, and the remaining $n/2$ keys are unique. Then for field $\vec{f_1}$, we have a binary search tree with $n/2 + 1$ unique keys, and the node v representing the key with cardinality $n/2$ will be at depth $\log n/2 + 1$. Focusing just on v, we can continue this pattern producing a tree with maximum depth $\Omega(d \log n)$ as the maximum depth of the tree in each level reduces by only 1. □

4.2 Rank-Interval Tree and Static Construction

In this section, we present an efficient search tree data structure for preprocessing lexicographic ordered set when the entire set is given. Our main result is a novel tree structure that uses $O(n)$ space and processes queries in $O(d + \log n)$ time.

The problem with the native multidimensional tree is that a large cardinality node v can be a leaf node for a given level of the naive tree. This means it can take $\Omega(\log n)$ time to get to v, and the subsequent tree on the next level still has $\Omega(n)$ keys. Hence, only local tree re-balancing is not sufficient to get $O(d + \log n)$ search time.

4.2.1 Rank-Interval Tree Definition

We provide a different structure of multidimensional binary search tree which we call a Rank-Interval Tree (RIT) that achieves an optimal worst case search time of $\Theta(d + \log n)$ by ensuring that

high cardinality nodes are higher in the tree. Such a tree is possible because the sum of cardinalities is n, so there cannot be too many high cardinality nodes.

When describing RIT, we focus on one field at a time. The slight difference is now we consider that every node contains an interval rather than a key. In particular, each key u_i can be considered as an interval with identical lower and upper bounds $[u_i, u_i]$. At field i, q_i matches an interval $[a, b]$ if $u \in [a, b]$. If we find a match, we use the third pointer to proceed to the next field if $i < d$ or return the matching box if $i = d$.

We now define some notation. Suppose we have n keys (possibly non-distinct) and d dimensions for a subproblem $T(n, d)$. Let $I = \langle i_1, i_2, \ldots, i_q \rangle$ be the *sorted* list of unique intervals $\vec{f}_1(b)$ for the n keys b. Let $c(i)$ be the cardinality of the node corresponding to interval i. Each interval $i \in I$ corresponds to $c(i)$ keys and $\sum_{i \in I} c(i) = n$.

Definition 5. *For any interval i, we define* $rank(i) = \lfloor \log n \rfloor - \lfloor \log c(i) \rfloor$.

We now prove the following two simple propositions.

Proposition 4.2.1. *For any two intervals* i_j *and* i_l *in I, we can determine the exact value of* $rank(i_j) - rank(i_l)$ *given only* $c(i_j)$ *and* $c(i_l)$ *without knowing n.*

Proof. This follows from the definition of rank and the fact that the $\lfloor \log n \rfloor$ terms will cancel out. □

Proposition 4.2.1 is useful in the dynamic setting where we must insert and delete keys.

Proposition 4.2.2. *For any interval* $i \in I$, $rank(i) = k$ *implies* $c(i) < n/2^{k-1}$ *for* $k \geq 0$.

Proof. The proof follows from the definition of rank. If $rank(i) = k$ for $k \geq 0$, then $\lfloor \log n \rfloor - \lfloor \log c(i) \rfloor = k$ which can be rewritten as $\lfloor \log c(i) \rfloor = \lfloor \log n \rfloor - k$. We know that $\log c(i) < \lfloor \log c(i) \rfloor + 1$ and $\lfloor \log n \rfloor \leq \log n$, so we can rewrite the equation above as $\log c(i) < \log n - (k - 1)$, and the result follows. □

Proposition 4.2.2 is used in the proof of Theorem 4.2.1.

Besides the q actual intervals from I, we also use spanning intervals which span a set of consecutive intervals. For example, the spanning interval $I(3,5) = [\min(i_3), \max(i_5)]$ spans intervals i_3, i_4, and i_5 (plus any gaps contained within them). We define the cardinality $c(I(j,l)) = c(i_j) + c(i_l)$, and $rank(I(j,l))$ is defined as with actual intervals. We form these spanning intervals with the aid of the following definition. In a sorted list of intervals I which may include spanning intervals replacing real intervals, two intervals are k-adjacent if they both have rank k and the intervals between them have rank strictly higher than k. Basically, we form a spanning interval of rank $k-1$ by combining all the intervals spanned by a pair of k-adjacent intervals.

We now prove the following key property about spanning intervals that we use extensively.

Lemma 4.2.1. *If i_j and i_l are k-adjacent intervals, thenk $I(j,l)$ has rank $k-1$.*

Proof. This follows from the definition of rank and spanning intervals. Specifically, if $rank(i_j) = rank(i_l) = k$, then $\lfloor \log c(i_j) \rfloor = \lfloor \log c(i_l) \rfloor$. By the definition of $I(j,l)$, $c(I(j,l)) = c(i_j) + c(i_l)$. It follows that $\lfloor \log c(I(j,l)) \rfloor = \lfloor \log c(i_j) + 1 \rfloor$ which means that $rank(I(j,l)) = k-1$. □

We create an RIT from I using two types of tree nodes: "real" nodes v which correspond to real intervals $I(v)$ in I and spanning nodes v which correspond to spanning intervals $I(v)$ constructed from I. All the nodes are organized as a binary search tree, but the spanning nodes use the third pointer to point to the root of an embedded RIT. Intuitively, when we come to a spanning node v with a query point q, we see if $q < I(v)$, $q \in I(v)$, or $q > I(v)$. In the first and third case, we go to the left or right child, respectively. In the second case where $q \in I(v)$, we proceed to the third pointer which points to an embedded RIT that contains all the real intervals spanned by $I(v)$. For a real node, the second case means we have found a match and we either return the matching box or use the third pointer to proceed to the next field. To illustrate, consider Figure 4.1 (e). The spanning nodes are denoted as red nodes. Each spanning node's third pointer, denoted with a red arrow, points to another RIT.

4.2.2 RIT Properties

RITs satisfy two invariants. The first invariant IV_1 is that any root-to-leaf path has monotonically increasing rank with at most two nodes (spanning or real nodes) of the same rank. The second invariant IV_2 is that two consecutive nodes v and $parent(v)$ have the same rank if and only if $parent^2(v)$ is a direct spanning node of $v, parent(v)$. We now prove the key property of RITs.

Lemma 4.2.2. *The depth of any rank k node in a RIT is at most* $2k + 1$.

Proof. By invariant IV_1, if we follow any path in T from the root node r to any node v of rank k, the ranks of the nodes on the path are monotonically increasing and we encounter at most two nodes of rank k' for $1 \leq k' \leq k$. There is at most one rank 0 node in any RIT. Thus, the total length of the path is at most $2k + 1$. □

This leads to the following theorem.

Theorem 4.2.1. *Consider a d-dimensional keys J and query point q. Let T be an RIT for J. The worst case search time for query q in T is* $2 \log n + 3d$ *which is* $\Theta(d + \log n)$.

Proof. We assume without loss of generality that we reach the RIT for field d during our search for q. Let v_i be the final node visited in field $\vec{f_i}$ and $k_i = rank(v_i)$ for $1 \leq i \leq d$. Let $T(n', d')$ denote the total number of nodes visited in T given we have n' total boxes remaining and d' total fields including the current field remaning. Then $T(n, d)$ is the quantity we wish to compute.

We can bound the number of elements of node of rank k by $n/2^{k-1}$ using Proposition 4.2.2. We then apply Lemma 4.2.2 d times to get the following inequalities.

$$\begin{aligned}
T(n,d) &\leq 2k_1 + 1 + T(n/2^{k_1-1}, d-1); k_1 \leq \log n \\
T(n/2^{k_1-1}, d-1) &\leq 2k_2 + 1 + T(n/2^{k_1+k_2-2}, d-2); k_2 \leq \log(n/2^{k_1-1}) \\
T(n/2^{k_1+k_2-2}, d-2) &\leq 2k_3 + 1 + T(n/2^{k_1+k_2+k_3-3}, d-3); k_3 \leq \log(n/2^{k_1+k_2-2}) \\
&\dots \\
T(n/2^{\sum_i^{d-1} k_i - d + 1}, 1) &\leq 2k_d + 1; k_d \leq \log(n/2^{\sum_i^{d-1} k_i - d + 1})
\end{aligned}$$

Substituting each term into the first equation yields $T(n,d) \leq 2(\sum_i^d k_i) + d$. The result follows from $k_d \leq \log(n/2^{\sum_i^{d-1} k_i - d + 1}) = \log n - \log(2^{\sum_{i=1}^{d-1} k_i - d + 1}) = \log n - (\sum_{i=1}^{d-1} k_i - d + 1)$ which implies $\sum_i^d k_i \leq \log n + d$. □

4.2.3 RIT Construction

We describe how to construct a RIT and show that the construction method maintains the invariants. We work with the set of real intervals I defined above. We first describe how we convert I into a set of real intervals and spanning intervals. We start from $k = \log n$. We find the leftmost pair of k-adjacent intervals i_a and i_b to form a spanning interval, replacing all the enclosed intervals by this spanning interval $I(a,b)$. We define $span(I(a,b))$ to be the set of intervals replaced by spanning interval $I(a,b)$; for later values of k, $span(I(a,b))$ may include a spanning interval $I(c,d)$. In this case, $I(c,d) \in span(I(a,b))$, but intervals within $span(I(c,d))$ are not in $span(I(a,b))$. By the definition of rank and the definition of $c(i)$ for a spanning interval, the new spanning interval $I(a,b)$ has a rank $k-1$ since we combine two nodes of rank k. We continue until there are no k-adjacent intervals remaining. We then decrement k and repeat the process until $k = 0$.

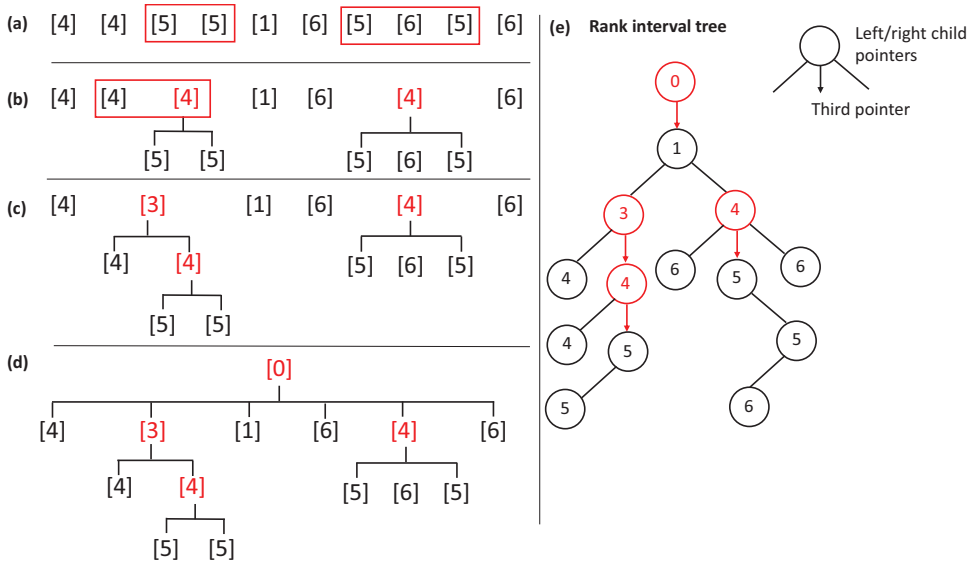

Figure 4.1: An example of RIT construction.

We now convert this set of intervals into a RIT T. We create a node for each interval: a real

node for real intervals and a spanning node for spanning intervals. By our construction, we do not have any k-adjacent intervals for any $k \geq 0$. We form a binary search tree by inserting the nodes into the tree in order of rank starting with the lowest rank node. If all nodes are real nodes, we are done. For any spanning nodes v, we recursively apply the same operation form a RIT $T(span(v))$ for the nodes in $span(v)$ and we have the third pointer of v point to the root of $T(span(v))$. The third pointer of all real nodes v' points to the root node of a RIT created for the n' boxes whose current field in $\vec{f}$ matches $I(v')$ using the next field in $\vec{f}$.

We illustrate the construction of a RIT in Figure 4.1. We start with the ten unique and sorted intervals depicted in Figure 4.1 (a) where an interval $[k]$ has rank k. Intervals in black are real intervals; intervals in red are spanning intervals. We use node and interval interchangeably. We start with the lowest rank 6, but there are no 6-adjacent intervals. For rank 5, there are two pairs of 5-adjacent intervals as denoted in the red rectangles. Figure 4.1 (b) shows the two resulting spanning intervals of rank 4 and intervals replaced by the spanning intervals. There is one pair of 4-adjacent intervals which is denoted by the red rectangle in Figure 4.1 (b). Figure 4.1 (c) shows the resulting spanning interval of rank 3 and the two intervals replaced by the spanning intervals. We then find there are no more k-adjacent intervals for any value of k, so we create a final spanning interval of rank 0 as depicted in Figure 4.1 (d). Finally, in figure 4.1 (e), we show the resulting RIT where real nodes are denoted in black, spanning nodes are denoted in red, the third pointer of spanning nodes are denoted with red arrows, and the number inside each node is its rank.

We show that after construction the tree has invariants.

Lemma 4.2.3. *The RIT satisfies the invariants IV_1 and IV_2 after static construction.*

Proof. First, in a RIT T, if node p is the parent of node v and points to v using the third pointer, then $rank(v) < rank(p)$. Since node p uses its third pointer in this field of $\vec{f}$, p must be a spanning node. By our construction process, this implies the following. Node v plus another node v' must be k-adjacent where $rank(v) = rank(v') = k$. The node p contains union of two disjoint nodes v and v'. By Lemma 4.2.1, $rank(p) = k - 1$. Second, in a RIT T, if node v is the left or right child of node p, then $rank(p) \leq rank(v)$. This follows trivially from our node insertion process. Third,

in a RIT T, if node v is the left or right child of node p and $k = rank(p) = rank(v)$, then nodes p and v were k-adjacent nodes that were combined to form node r whose child is node p pointed to by the third pointer of r. Suppose this is not true. By our construction process, nodes p and v must have rank at least as low as all nodes that lie between then in T. This means nodes p and v would be k-adjacent. Putting together the three properties, the invariant IV_1 is satisfied since if we follow any path in T from the root node r to any node v of rank k, the ranks of the nodes on the path are monotonically increasing and we encounter at most two nodes of rank k' for $1 \leq k' \leq k$. Invariant IV_2 is satisfied by the first and third properties.

□

4.3 Dynamic RITs

We now show that RITs also support insertions and deletions in $O(d + \log n)$ time for dynamic instances.

To show this running time, it is sufficient to show that if a box i is in a node of rank k in the search path of field j, the insertion/deletion of key i in the RIT of field j can be done in $O(k)$ time. The crucial idea is when we insert/delete an element from a RIT, we perform a constant amount of work per node on the search path to ensure the invariants IV_1 and IV_2 are still maintained. This is similar to the approach of many balanced binary search trees such as red-black trees.

Proposition 4.2.1 is critical for handling insertions and deletions. Specifically, it allows us to compare ranks for two nodes j and l given $c(j)$ and $c(l)$ even if we do not know n.

We first address insertions; we then handle deletions. We will show that the difficult case for both insertions and deletions is when we must change spanning nodes on the search path.

One simplification we make is to no longer record the actual cardinality $c(v)$ for spanning nodes v in our RIT. Instead, spanning nodes record $\lfloor \log c(v) \rfloor$. The key observation is that $\lfloor \log c(v) \rfloor$ can only change when $c(v')$ changes where v' is one of the two nodes merged to form v.

We make a second minor notational change to simplify our discussion. If node sp is a spanning node that merges nodes w and w', we refer to w and w' as siblings even though one is technically

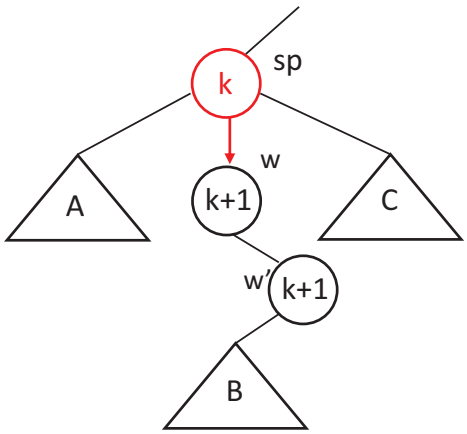

Figure 4.2: Illustration of left/right siblings.

the parent of the other node. For example, consider Figure 4.2. Here we say that w is a left sibling of w' and w' is a right sibling of w. There is a symmetric case where w' is pointed to by the third pointer of sp and w would be accessed via the left pointer of w'.

4.3.1 Insertion in $O(d + \log n)$ time

We assume that we are inserting a key b. We separate insertion into two phases: simple insertion and path repair. In the simple insertion phase, we first search for b in the RIT. If we are processing field i, when we come to node v, we compare $\vec{f_i}(b)$ with $\vec{f_i}(I(v))$. To simplify notation, we drop $\vec{f_i}$ and say we just compare b with $I(v)$. If $b < I(v)$, we proceed to the left child of v and recurse. If $b > I(v)$, we proceed to the right child of v and recurse. If $b = I(v)$ and v is a real node, we increment $c(v)$ and follow the third pointer to the next field and recurse. We note there must be some next field since we assume that insertion is possible which means that at some field i, b will have a unique interval. If $b = I(v)$ and v is a spanning node, we follow the third pointer to the root of the v's subtree and repeat the process recursively. If a null child is found, which must happen eventually, we create a new real node v with $c(v) = 1$, and we create singleton node trees in the subsequent fields which do not require path repair since $c(v) = 1$. This completes the simple insertion phase which clearly runs in $O(d + \log n)$ time given this is essentially identical to search.

We now proceed to path repair. We focus on a specific field i. Let b denote $\vec{f_i}(b)$, T be the RIT that we searched in field i when inserting b, and let v be the node that was modified in T by the insertion of v. Note that v is either a new node with $c(v) = 1$ or v is an existing node where we incremented $c(v)$ by one. We use $c'(v)$ to denote the old value of $c(v)$ including denoting $c'(v)$ to be 0 if v is a new node. Let the root-to-leaf path $\mathcal{P}$ be the path from r, the root of T, to v. After simple insertion, the following is true: invariants IV_1 and IV_2 hold everywhere in T except for a possible violation between v and $parent(v)$ since $c(v)$ increased by 1 or v was created with $c(v) = 1$ which means $rank(v)$ may have changed by exactly 1.

In path repair, this will be our steady state. Specifically, invariants IV_1 and IV_2 will hold everywhere in T except for a possible violation between a specified node v and $parent(v)$ due to $rank(v)$ possibly decreasing by 1. We will repair the potential violation between v and $p(v)$ in such a way that invariants IV_1 and IV_2 will still hold everywhere in T except for a possible violation between a new node v' and $parent(v')$ where v' is some node above v on $\mathcal{P}$. Note that v' may not have originally been on $\mathcal{P}$.

There are some cases where no path repair is needed. First, if $\lfloor \log c(v) \rfloor = \lfloor \log c'(v) \rfloor$, no repair is needed as node v's rank has not changed. This also implies that the rank of any spanning nodes on $\mathcal{P}$ have not changed as they can only change rank if some node on $\mathcal{P}$ changes rank. Thus, we are done. A second case where no path repair is needed is if $\lfloor \log c(v) \rfloor > \lfloor \log c'(v) \rfloor$ but $\lfloor \log c(v) \rfloor < \lfloor \log c(parent(v)) \rfloor$. In this case, v has changed rank, but its new rank is still strictly smaller than its parent's rank, so IV_1 and IV_2 hold for all of T. This case can only occur if v has no left or right sibling.

The remaining cases are (1) v has no left or right sibling and $\lfloor \log c(v) \rfloor = \lfloor \log c(parent(v)) \rfloor$ and (2) v has a left or right sibling and $\lfloor \log c(v) \rfloor > \lfloor \log c'(v) \rfloor$.

We address case (1) using the following merge operation depicted in Figure 4.3. We form a new spanning node v' with rank $k - 1$ by merging v and $parent(v)$. This new spanning node v' takes the place of $parent(v)$ on $\mathcal{P}$ where we set the left child pointer of v' to point to the root of subtree A. We note there is a symmetric case where v is the right child of $parent(v)$, and a symmetric merge

operation handles that case.

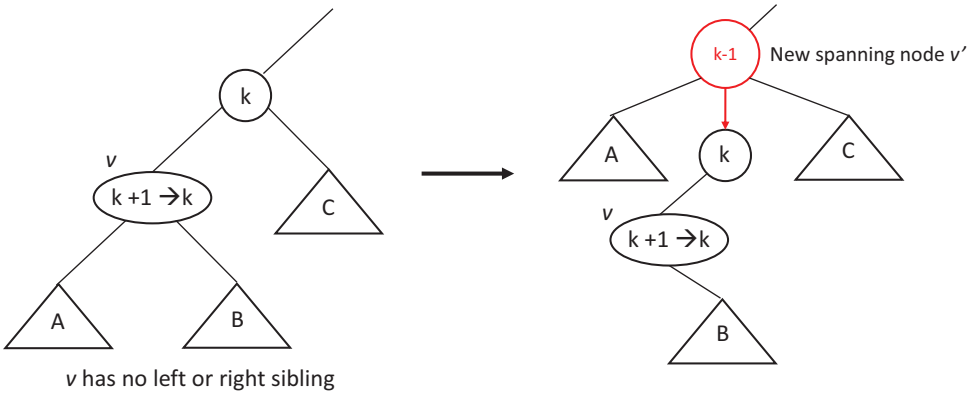

Figure 4.3: Merge Operation.

Lemma 4.3.1. *If the only violation of IV_1 and IV_2 occurs at node v where v has no left or right sibling and $rank(v) = rank(parent(v))$, we can eliminate this violation merging v and $parent(v)$. After the merge and replacing $parent(v)$ with the new spanning node v', the only possible violation of IV_1 and IV_2 in T can occur between v' and $parent(v')$.*

Proof. If we assume that IV_1 and IV_2 hold for T before $rank(v)$ decreases by 1 from $k+1$ to k, then it follows immediately that IV_1 and IV_2 hold for the subtree rooted at v'. Furthermore, the only possible violation of IV_1 and IV_2 occurs between v' and $parent(v')$ since $rank(v') = k-1$ whereas $rank(parent(v)) = k$ and v' has replaced $parent(v)$. □

We address case (2) as follows.

Lemma 4.3.2. *If the only violation of IV_1 and IV_2 occurs at node v where v has a left or sibling and $\lfloor \log c(v) \rfloor > \lfloor \log c'(v) \rfloor$, we can fix the violation between v and $parent(v)$ and advance the next point of the violation q steps where $q \geq 1$ doing $O(q)$ work.*

Proof. The situation must look like that of Figure 4.2 or the symmetric variant where w' is the child of sp rather than w and v must be w or w'. We break case (2) into several cases. The simplest is if v is w'. We fix this by a simple rotation making w' the child of sp and w the left child of w'

and r_B the right child of w. This brings us to the symmetric case of $v = w$. We have advanced the point of violation by 1 step doing constant work, so this case is complete.

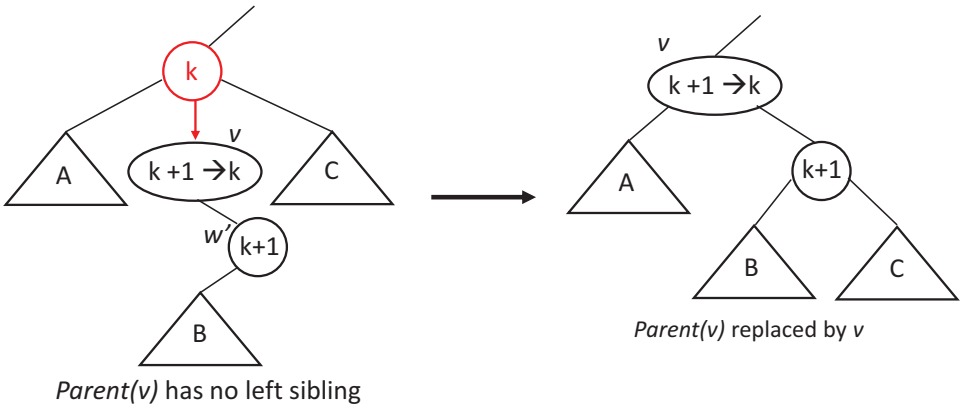

Figure 4.4: Fixing case 2.1: Replace $parent(v)$ with v

We now consider the case where $v = w$. We break this case into two subcases. In case (2.1), $parent(v)$ has no left or right sibling. We then fix the violation using the following steps as depicted in Figure 4.4. We replace $parent(v)$ with v and perform appropriate pointer rewiring. Namely, we set v's left child to be $parent(v)$'s left child, and we set the right child of w' to be $root(C)$. Note that since w' was v's right sibling, it did not have a right child before. Furthermore, in this case, we actually end the path repair as v has taken $parent(v)$'s place on $\mathcal{P}$ with the same rank as $parent(v)$, so IV_1 and IV_2 must hold throughout T. We have eliminated all violations doing only constant work, so this case is complete.

In case (2.2), $parent(v)$ has a left or right sibling. Without loss of generality, we may assume that the situation is as depicted in Figure 4.5 (a) or (b). If not, we simply perform one rotation between $sp = parent(v)$ and sp's sibling to make sp the direct child of its spanning node. We note there are the symmetric cases where sp has a left sibling and a spanning node parent.

If the configuration is that depicted in Figure 4.5 (a), we replace sp with v breaking apart v and w'. We then make y the new right child of v, merge y and v, and make sp^2 the resulting spanning node. Note that because w' was v's right sibling, $I(w')$ lies between $I(v)$ and $I(y)$ which means that $I(sp^2)$ is unaffected by these changes. We then do some pointer manipulation. Specifically, we

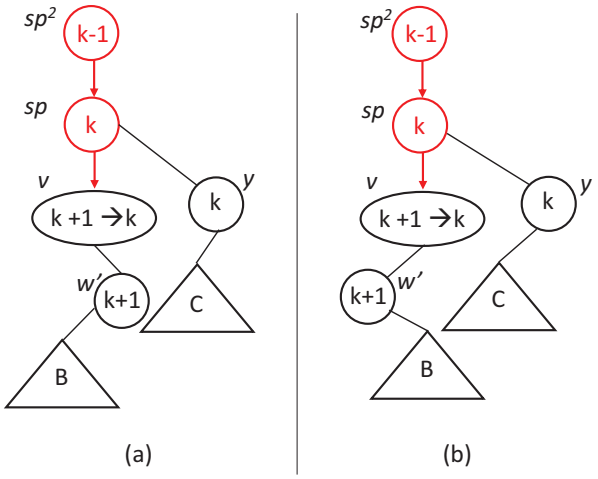

Figure 4.5: Case (2.2): If sp has a right sibling and a spanning node parent.

make $root(C)$ the right child of w' and we make w' the left child of y. The only possible issue is if $root(C)$ has rank $k+1$ in which case $root(C)$ and w' violate the invariants. In this case, we merge $root(C)$ with w', and then we merge the new spanning node v' that has rank k with y. Finally we rotate this second new spanning node that has rank $k-1$ with v to make this new spanning node the direct child of sp^2. This returns us to case (2) where this new spanning node is the new v and sp^2 is the new $parent(v)$. Note that if $root(c)$ has rank strictly greater than $k+1$, no additional work would be required and tree T would now satisfy invariants IV_1 and IV_2. In this case, we have advanced the point of violation by at least one step while doing constant work.

The final case we must deal with is if the configuration is that depicted in Figure 4.5 (b). This is the most involved case. Similar to case (2.2a), we again replace sp with v breaking apart v and w'. We then make y the new right child of v, merge y and v, and make sp^2 the resulting spanning node. However, unlike case (2.2a), $I(w')$ is smaller than $I(v)$ which means that $I(sp^2)$ no longer contains $I(w')$. We need to find a place to reinsert w' into T that lies "to the left" of sp^2.

We illustrate how we find the place to reinsert w' in Figure 4.6. We work our way up $\mathcal{P}$ starting with sp^2 which has rank $k-1$ until we find a node x that does not have a right sibling. Note that we must eventually find such a node as $root(T)$ has no right sibling. For each node w that we encounter during this search that does have a right sibling, we assume without loss of generality that w's parent is the spanning node formed by merging w and w's right sibling. The crucial observation is

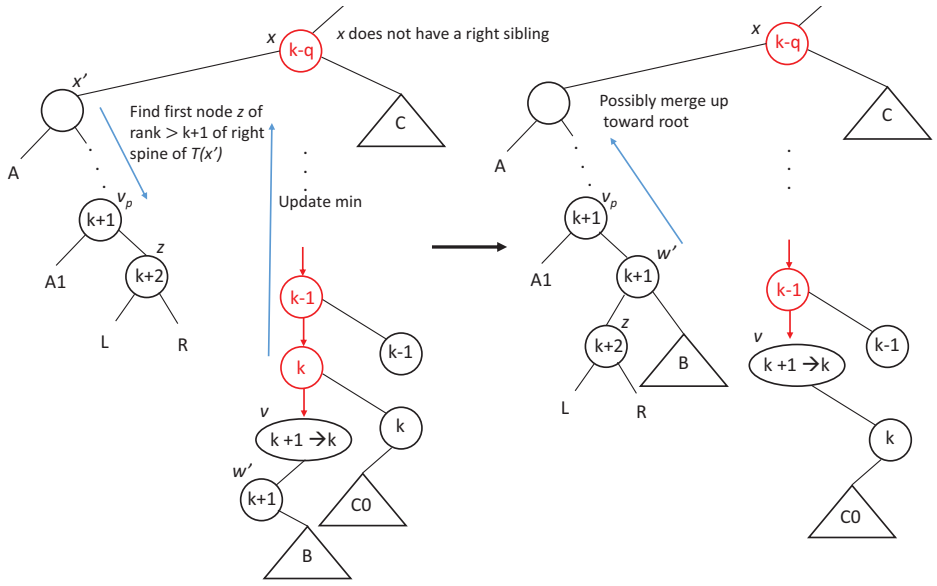

Figure 4.6: Fixing case (2.2b). Note that $q \geq 1$.

that if w has a right sibling, then w's left child pointer must be null. We update $\min(I(w))$ to be $\min(I(v))$. When we find this x, it could be the case that x has a left sibling or x is not a spanning node. Let the rank of x be $k - q$ where $q \geq 1$.

Regardless of which case applies for x, the crucial observation is that the left child pointer of x no longer needs to be null. If x's left child pointer is null, we simply set w' to be the left child of x and we are done. Tree T has no violations of any invariant. If x's left child pointer is not null, let x' denote the left child of x. We then find the place to insert w' by searching in the right spine of the subtree rooted at x'. We proceed until we find a node z with rank greater than $k + 1$ or we find the end of the right spine of $T(x')$, whichever comes first. Let v_p be the parent of z or the last node on the right spine of x'; note x could be v_p. We then reset v_p's pointer to point to w' instead of z. If we found a node z, we then set the left child pointer of w' to point to z as shown in Figure 4.6.

Finally, if v_p has rank $k + 1$, we then have an invariant violation between w' and v_p. However, the crucial observation is that all the nodes on the right spine of $T(x')$ have no left or right siblings by definition of the right spine since we never traverse a third pointer. We resolve the invariant between w' and v_p by merging w' and v_p. This may lead to another invariant violation between the resulting spanning node v' and the parent of v_p. Because of our crucial observation, we can resolve

this and any future violations by repeatedly merging the nodes until we eventually merge with x'. If x' is the left sibling of x, then we have returned to case (2) where this new node that replaces x' is v and is in violation with its parent x which is also its right sibling. The crucial observation is that we have advanced the point of violation at least q levels while doing $O(q)$ work. In summary, we have shown that we can handle all subcases within case (2) where we advance at least q steps towards $root(T)$ while doing at most $O(q)$ work. □

Theorem 4.3.1. *Inserting a d-dimensional key b into a RIT maintaining IV_1 and IV_2 can be done in $O(d + \log n)$ time.*

Proof. We observe the simple insertion phase takes at most $O(d + \log n)$ time resulting in at most one violation per RIT per field. If the affected node is at rank k in tree T in field i, then Lemmas 4.3.1 and 4.3.2 imply that the path repair phase takes at most $O(k)$ work for tree T. Given this is proportional to the search time in tree T, this implies the total path repair time is $O(d + \log n)$ and the result follows. □

4.3.2 Deletion in $O(d + \log n)$ time

Before we start, we first introduce a rank rotation operation that we will need for some cases during path repair. Rank rotation is a basic left or right tree rotation possibly followed by two merges. Consider Figure 4.7 (and its symmetric counterpart). Assume that subtrees A, B, and C satisfy IV_1 and IV_2 and their roots have ranks $\geq k$, $\geq k$ and $\geq k+1$, respectively. We perform basic tree rotation for node v' and v. Let r_b be a root node of subtree B. If $rank(r_b) > k$, we are done since subtree rooted at v' after rotation satisfies IV_1 and IV_2. Otherwise, $rank(r_b) = k$. In this case, we merge r_b and v to obtain a new spanning node sp of rank $k-1$ that replaces v. Next, we perform another merge between sp and v' to obtain another new spanning node sp' of rank $k-2$ that replaces v'. The subtree rooted at vsp' now satisfies IV_1 and IV_2. The net result is that we fix the violation between v and its left child where the new node in the position of v may have a rank that is 1 or 2 smaller.

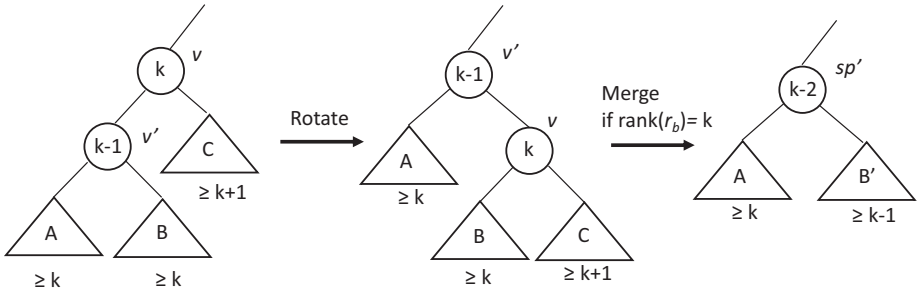

Figure 4.7: Rank Rotation operation.

We assume that we are deleting a key b. We separate deletion into two phases: simple deletion and path repair. In the simple deletion phase, we first search for b in the RIT. If we are processing field i, when we come to node v, we compare b with $I(v)$ (we again drop $\vec{f_i}$). If $b < I(v)$, we proceed to the left child of v and recurse. If $b > I(v)$, we proceed to the right child of v and recurse. If $b = I(v)$ and v is a real node, we maintain a pointer to $c(v)$ and follow the third pointer to the next field and recurse. If $b = I(v)$ and v is a spanning node, we follow the third pointer to the root of the v's subtree and repeat the process recursively. If a null child is found, this means the search failed and no deletion occurs. If box b is found, we then decrement $c(v)$ in each final node for each field and proceed to path repair in each tree. Note that in at least the dth field, $c(v)$ must be 0. If $c(v) = 0$, we must delete v from the tree completely, but we delay this deletion to the path repair stage. This completes the simple deletion phase which clearly runs in $O(d + \log n)$ time given this is essentially identical to search.

We now proceed to path repair. We focus on a specific field i. Let b denote $\vec{f_i}(b)$, T be the RIT that we searched in field i when deleting b, and let v be the node that was modified in T by the deletion of v. We use $c'(v)$ to denote the old value of $c(v)$. Let the root-to-leaf path $\mathcal{P}$ be the path from r, the root of T, to v. After simple deletion, the following is true: invariants IV_1 and IV_2 hold everywhere in T except for the following possible violations. Case (1): if v has no left or right sibling, we may have a violation between v and either or both of its binary search tree children since $c(v)$ decreased by 1. Case (2): if v has a left or right sibling, we may have a violation between v and its sibling and their spanning node parent if $rank(v)$ no longer matches the rank of its sibling. We also note that if $c(v) = 0$, we do need to delete node v completely. In this case, we can only have

the second type of violation as a node with $c(v) = 1$ will have no children other than a left or right sibling.

As with insertion, there are some cases where no path repair is needed. First, if $\lfloor \log c(v) \rfloor = \lfloor \log c'(v) \rfloor$, no repair is needed as node v's rank has not changed. This also implies that the rank of any spanning nodes on $\mathcal{P}$ have not changed as they can only change rank if some node on $\mathcal{P}$ changes rank. Thus, we are done. A second case where no path repair is needed is if $\lfloor \log c(v) \rfloor < \lfloor \log c'(v) \rfloor$ but $\lfloor \log c(v) \rfloor > \lfloor \log c(child(v)) \rfloor$ for both possible children of v. In this case, v has changed rank, but its new rank is still strictly larger than the rank of its possibly two children, so IV_1 and IV_2 hold for all of T. This case can only occur if v has no left or right sibling.

We now deal with our two possible invariant violations. We first deal with case (1) which turns out to be a very easy case.

Lemma 4.3.3. *If the only violation of IV_1 and IV_2 occurs at node v where v has no left or right sibling and $rank(v) = rank(child(v))$, we can eliminate this violation by merging v and $child(v)$. After the merge, there will be no violations of IV_1 and IV_2 in T.*

Proof. First, we observe again that this case cannot occur if $c(v) = 0$ as v must have been a singleton node before the deletion and a singleton node cannot have a child that is not a left or right sibling. Without loss of generality, we assume that the child with equal rank is its left child lc. When we merge lc and v to form v', we replace v with v'. We make v' point to v and lc is still the left child of v. We make the old right child of v, if it exists, the right child of v'. Since v' will have the rank that v used to have, we have eliminated all possible invariant violations. □

We now deal with the second case where we have a single node v whose rank has changed and v has a sibling.

Lemma 4.3.4. *If the only violation of IV_1 and IV_2 occurs at node v where v has a left or sibling and $\lfloor \log c(v) \rfloor < \lfloor \log c'(v) \rfloor$, we can fix the violation among v, its sibling, and its spanning node parent and advance the next point of the violation q steps where $q \geq 1$ doing $O(q)$ work.*

Proof. The situation must look like that of Figure 4.2 or the symmetric variant where w' is the child of sp rather than w and v must be w or w'. If v is w, we do a simple rotation between w and w' making w' the direct child of sp. Thus, without loss of generality, we assume v is w'.

If the rank of the root r_B of subtree B is $k+1$, we merge r_B and v and replace v with the resulting spanning node v'. It is easy to see that this deals with all invariant violations and we are done.

If $rank(r_B) > k+1$ or B is empty, then we split case (2) into two subcases similar to what we did with insertions. The first subcase (2.1) is that sp has no left or right sibling. In this case, we split the merge between v and w as illustrated in Figure 4.8. That is, we promote w to replace sp and we make v the left child of w. In the process, we now make r_A, the root of subtree A, the left child of v and r_C, the root of subtree C, the right child of w. This intermediate state is illustrated in the middle of Figure 4.8.

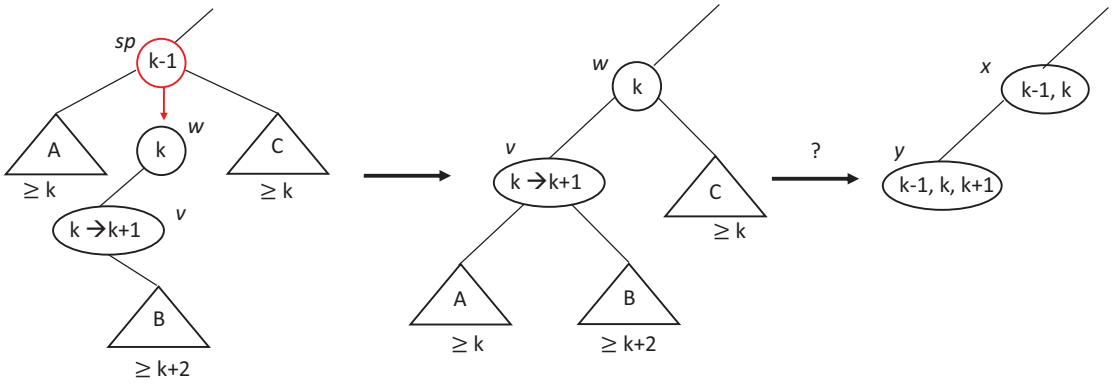

Figure 4.8: Split Merge operation.

We could have violations between r_A and v and r_C and w. We address these as follows with the result illustrated in the rightmost tree of Figure 4.8 where we only focus on the node x which ultimately replaces w and the node y which ultimately replaces v. We start with the relationship between w and r_C and the resulting node x. If $rank(r_C) = k$, we merge r_C and w to form a new spanning node x with rank $k-1$. Otherwise, there is no violation between w and r_C and x is just w with rank $k-1$. Thus, node x has rank $k-1$ or k.

We now consider r_A and v and the resulting node y. If $rank(r_A) \geq k+2$, there is no violation and y is just v with rank $k+1$. If $rank(r_A) = k+1$, we merge r_A and v to make a rank k spanning

node y. The last possibility is that $rank(r_A) = k$. In this case, we need to perform a rank rotation operation between r_A and v. This results in node y having rank k or $k-1$. Considering all the possibilities, node y has rank $k-1$, k, or $k+1$.

We now consider the possibilities between node y and node x. We may need to merge y and x if they share the same rank resulting in a root node with rank $k-2$ or $k-1$. We may have to perform a rank rotation between y and x resulting in a root node with rank $k-2$ or $k-1$. Finally, we may not need to do anything resulting in a root node with rank $k-1$ or k. Thus, the root node will have rank k, $k-1$, or $k-2$.

If the root node has rank $k-1$ or k, we are done as we initially assumed that sp had no left or right sibling and we have not decreased the rank of this subtree. Thus, no invariants are violated and this case is complete. If the root node has rank $k-2$, we return to the case of insertion path repair as we have actually decreased the rank of this node, and we leverage our insertion path repair results to complete this case. Thus, subcase (2.1) is complete.

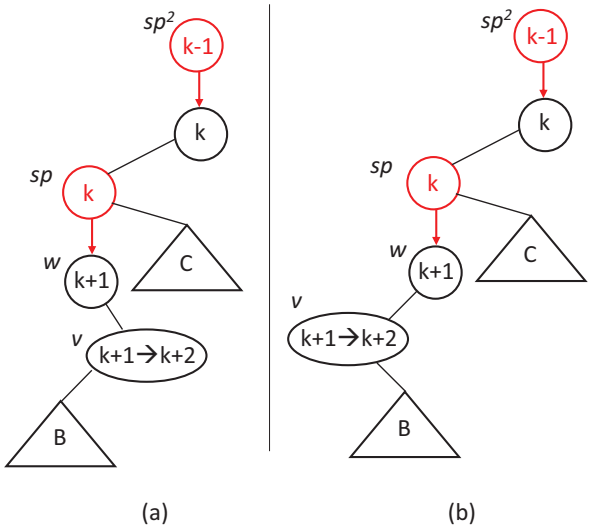

Figure 4.9: Case (2.2): If sp has a right sibling.

We now consider subcase (2.2) where sp has a left or right sibling. Similar to insertion, we consider two subcases illustrated by Figure 4.9. The other possibilities are symmetric to these cases and so the same analysis applies to them. For subcase (2.2a), depicted in Figure 4.9 (a), we replace sp with w and move subtree C to be the right child of v. We consider the structure of subtree C. Let

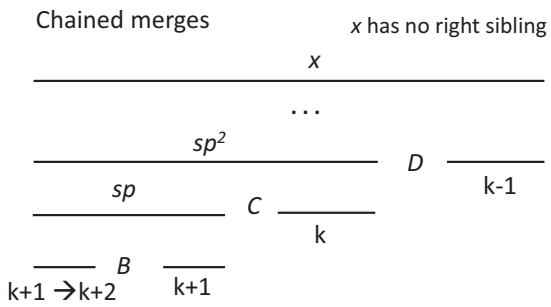

Figure 4.10: Alternate representation of Figure 4.9 (b).

r_C be the root of subtree C. If $rank(r_C) > k + 2$ or r_C does not exist, then we now have moved the invariant violation up one level to node w. This completes this case as we have done $O(1)$ work.

If $rank(r_C) = k + 2$, we merge r_C and v to make a new spanning node v' with rank $k + 1$ to replace v, and this v' can merge with w to create a new spanning node x with rank k to replace w. This completes this case as T now satisfies IV_1 and IV_2.

Otherwise, $rank(r_C) = k + 1$. Let r_L and r_R be the roots of the left and right subtrees of r_C. We first rotate r_C with v. This means r_L is the right child of v and a possible violation. We now merge r_C and w to make a node v' with rank k to replace w. If $rank(r_L) > k + 2$, we have no violation between r_L and v and this case is complete as T has no violations of IV_1 or IV_2. Otherwise $rank(r_L) = k + 2$, so we merge r_L and v to make a new node z with rank $k + 1$ that is a left child of r_C. This returns us to one of the insertion path repair violations as z has the same rank as its parent r_C. We leverage our insertion path repair results to complete this case. Thus subcase (2.2a) is complete.

Now we consider subcase (2.2b) depicted in Figure 4.9 (b). This situation is more complicated and is similar to the complicated case from insertion. We start again by replacing sp with w. If $rank(r_C) = k + 1$, then we merge r_C and w to create a new spanning node v' with rank k that replaces w. This now creates a situation that is identical to the insertion case (2.2b) from Lemma 4.3.2 as we must insert v back into T and we must find a place to perform this insertion. We leverage our insertion path repair results to complete this case.

We now consider the case where $rank(r_C) > k + 1$. In this case, w does not have sufficient rank

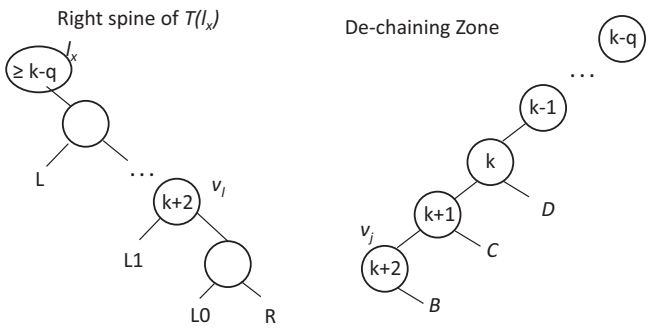

Figure 4.11: Merge-zip operation

to maintain the merge between sp and its right sibling. Thus, this leads to splitting not only sp but also sp^2 and potentially more spanning nodes. We illustrate this ripple effect in Figure 4.10.

We deal with this case as follows. We convert the chained merges in Figure 4.10 into a binary search tree. That is, we start from node v and we move up path $\mathcal{P}$ until we find a node x with rank $k - q$ that has no right sibling; this is similar to the case for insertion. Along this path, we basically replace each spanning node encountered with its right sibling child. The resultant tree is illustrated in the De-chanining Zone in Figure 4.11. We note that there are no gaps in rank from $k + 2$ up to $k - q - 1$ which is the right sibling child of x.

We now consider node x which may have a left sibling or a left child. The key point is that the left child pointer need not be null since x does not have a right sibling. We consider the right spine of the subtree rooted at l_x, the left child of x. This right spine is also depicted in Figure 4.11. We note that $rank(l_x) \leq k - q$ since it might be a left sibling of x.

We now basically merge the right spine with the De-chaining Zone. There may be gaps in rank in the right spine, but there are no gaps in rank in the De-chaining zone.

We progress up the two chains using pointers as follows. For the De-chaining zone, we start with v_j pointing to v, the minimum node in the De-chaining zone. Note that v_j has a null left child pointer. For the right spine, we set v_l to point to the node that has the largest rank at most $rank(v_j)$. Thus, $rank(v_l) \leq rank(v_j) < rank(rightchild(v_l))$ if v_l has a right child.

We now merge the two chains as follows. We move the right child of v_l to be the left child of

v_j which is possible since v_j has a null left child pointer. We then compare $rank(v_j)$ and $rank(v_l)$. Note that $rank(v_l) \leq k - q$. If $rank(v_j) < rank(v_l)$, we progress up the De-chaining Zone until either we find the node with $rank(v_l)$ or we move all the way to the top of the De-chaining Zone. In either case, we reset v_j to point to this node and make it the right child of v_l. If we moved to the top of the De-chaining Zone, this means $rank(l_x) \geq k - q - 1$. Thus, we either merge v_l and v_j creating a new node with rank $k - q$ to replace x and thus eliminating all invariant violations, or we replace x with v_l which is l_x and was a left sibling of x. In this case, we have moved the invariant violation up $q + 3$ levels doing $O(q)$ work which completes this case.

If we have not moved to the top of the De-chaining Zone, this means $rank(v_j) = rank(v_l)$. This is our steady state condition. In this case, we then merge v_j and v_l to create a new node that replaces v_l and we reset v_l to be this new node. Note that v_l has a null right child pointer. This may then lead to a series of merges in the right spine and we update v_l to continue to point the newly formed spanning node which continues to have a null right child pointer. This either continues all the way to l_x or there is a gap in the right spine and the merge stops where $rank(v_l) \leq k - q - 1$. The rank cannot be $k - q$ because v_l could merge with l_x in that scenario. We then reset v_j to be the node that has the same rank as v_l and we have returned to our steady state condition. Once in this steady state, we must continue this merging until we have exhausted the right spine which includes merging with l_x. We cannot stop beforehand because there is always a node in the De-chaining zone that can merge with the current v_l until it finally merges with l_x. There may be nodes left in the De-chaining zone if there no gaps at the top of the right spine and the final v_l has rank less than $k - q - 1$. If there are leftover nodes from the De-Chaining zone, we make the top node the right child of the final v_l. Finally, we replace x with this final v_l which must have rank $k - q - 1$ or $k - q$. If it has rank $k - q$, we are completely done if x had no left sibling. Otherwise, we have moved the invariant violation up $q + 3$ levels doing $O(q)$ work. If the final rank is $k - q - 1$, then we have advanced the invariant up $q + 3$ levels doing $O(q)$ work and we have returned to an insertion case as $rank(v_l) = rank(x) - 1$. This completes this final case. □

CHAPTER 5

CONCLUSION AND FUTURE WORK

We have designed a new packet classification algorithm (PartitionSort) that is dynamic and dimensionally scalable using linear space, and we also design a new data structure (RITs) supporting efficient insertion and deletion of rules. We now describe possible extension of our work.

5.1 Batched Processing.

Here is the key idea: given a batch of packets, we can classify them using the basic idea of mergesort. Namely we do rounds of sorting packets and then merging them with sorted rule. With batched processing, we can achieve packet classification using amortized O(1) time per packet if we use a linear time sorting algorithm such as radix sort.

For this work, the basic ideas have been fleshed out. I mainly need to carefully write these up and then perform experiments that validate the approach in practice.

Now we provide more details about how we hope to achieve batch packet processing using sortable rulesets. We first illustrate our approach for the one-dimensional case. We then describe the challenges we must overcome moving to multiple dimensions.

One-dimensional Batch Packet Processing. We first assume that our m one-dimensional packets are sorted. If not, we simply sort them. Note that with $d = 1$, packets are just numbers. Our n one-dimensional rules are stored in an appropriate data structure such as a balanced binary search

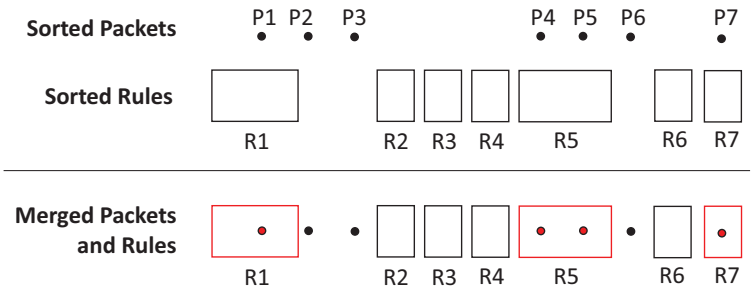

Figure 5.1: An example of 1D packet-rule merge

tree or a B-height tree. The key is that we can output the rules in sorted order by doing a simple traversal of the tree. We now process our packets using a merge operation similar to the merge operation from merge sort. Figure 5.1 illustrates the basic idea. We store the results of the merge in an array indexed by packet which guides what happens with subsequent partitions. We initialize the result array to be empty. We compare the current packet p to the current rule r. Initially, p and r are the first packet and the first rule in sorted order, respectively. In our merge, we have two modes. We either compare p to $\min(r)$ or to $\max(r)$. We start in mode $\min(r)$. In the $\min(r)$ mode, if $p < \min(r)$, then p does not match any rule and its result is already correct. We reset our current packet p to be the next packet staying in $\min(r)$ mode. If $p \geq \min(r)$, we then switch to $\max(r)$ mode. In the $\max(r)$ mode, if $p < \max(r)$, then p matches r. We set the result for p to be r, and we reset our current packet p to be the next packet staying in $\max(r)$ mode. If $p > \max(r)$, we reset to $\min(r)$ mode and reset our current rule r to be the next rule. We continue in this fashion until either all packets or all rules are finished. When we move to the next sortable ruleset, we continue in this fashion except with a few small changes. First, we initialize the result array to be the array that carries over from the previous ruleset. We note a few positive features of this merge process. First, the total number of comparisons is at most $m + 2n$ per tree and thus $mT + 2n$ where T is the number of partitions. This can be seen where we charge each comparison to p if p is smaller than the endpoint of r it is compared to or to r if the endpoint of r is smaller than p. Each packet p is charged at most once per partition and each endpoint of r is charged at most once, so the result follows. Second, it follows that the resulting number of comparisons is $O(1)$ per packet per partition as long as we ensure that $m = \Omega(n)$. Finally, similar to merge sort and sequential search, the caching behavior of the merge is good as packets and rules are accessed sequentially.

Multi-dimensional Batch Packet Processing. We will investigate how to batch process packets in the multi-dimensional case. We first note that a simple extension of the one-dimensional algorithm will not work, even when extending to $d = 2$.

For example, consider three points $P4$, $P6$, and $P7$ and box $R5$ from Figure 5.2. In XY order, $P6 <_{XY} P4 <_{XY} P7$. If we perform the simple merge routine, we first find that $P6$ matches

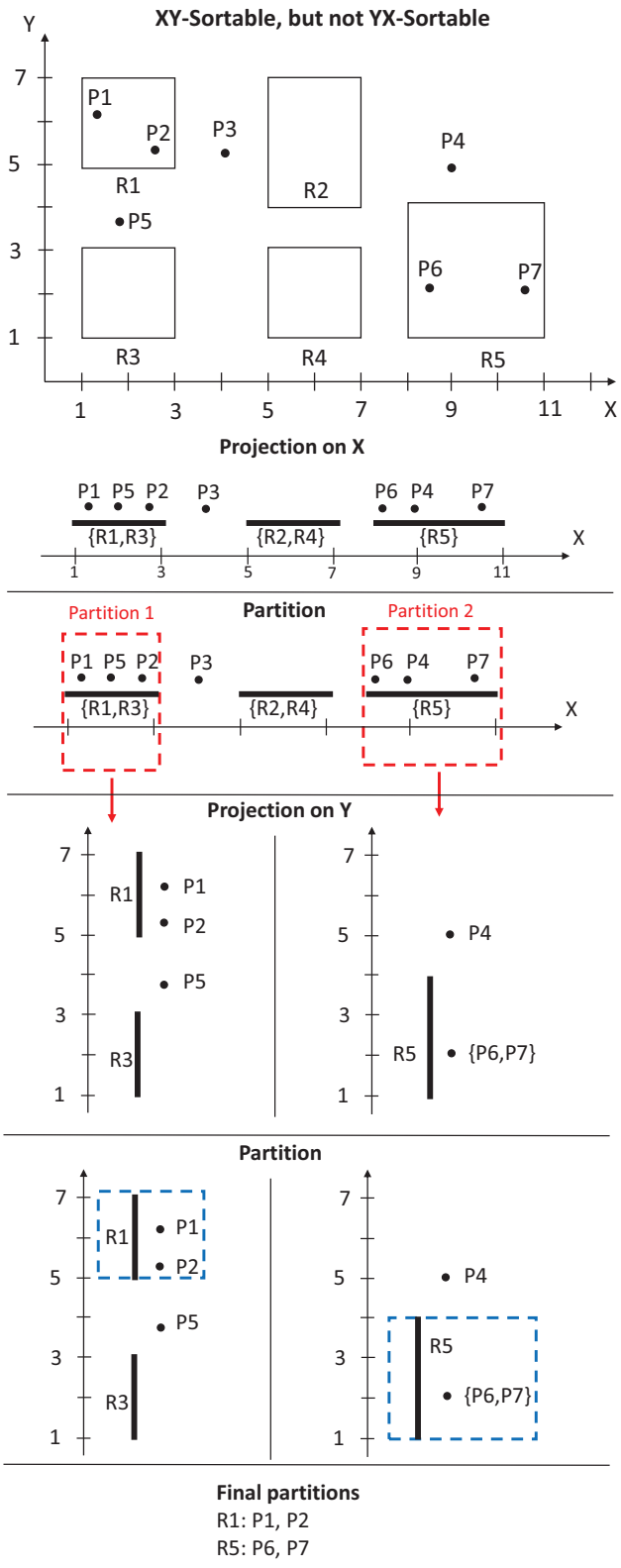

Figure 5.2: Example 2-D Partition Projection Algorithm

$R5$. We then find that $P4$ is too large for $R5$ in Y. Using the logic of the basic merge, we would then not compare $P7$ to $R5$, but $P7$ is contained within $R5$. To solve this problem, we propose an algorithm that we call Partition Projection Algorithm which solves this problem by performing the merge one field at a time using the given field permutation $\vec{f}$. For now, we assume that we have sorted the m query points in each field ahead of time deferring how we sort to the next task. The key idea behind Partition Projection Algorithm is that before we process a dimension i, we have partitioned the packets p and rules r into partitions such that if p and r are in the same partition, then $\vec{f}^{i-1}(p) \in \vec{f}^{i-1}(r)$. That is, the projection of p in the first $i-1$ fields of $\vec{f}$ lies within the projection of r in the first $i-1$ field of $\vec{f}$. We also have a special null partition that contains points p and rules r that match no rules or points, respectively for the first $i-1$ dimensions. When we process dimension i, we update the partitioning to reflect dimension i using the simple merge process so that p and r are in the same partition if $\vec{f}^{i}(p) \in \vec{f}^{i}(r)$.

Partition Projection Algorithm processes field $\vec{f}_i$ for $1 \leq i \leq d-1$ by running the basic merge modified as follows. For each active partition, we keep track of the mode and active rule. When we fetch the next point p, we first verify it is an active point. If it is in the null partition, we move on to the next point. If p is active, we get its current active partition and go to that partition's active rule r in that partition's active mode. If we determine a match between a point p and a rule r, then we place p and r into the same partition for the next field $i+1$. If we determine that p does not match any rule r in dimension i in its current partition, then we assign p to the null partition. Likewise, if we determine that no points from a rule r's current partition match r, then it goes to the null partition. When Partition Projection Algorithm processes field d, the only change is that we now match packets with rules rather than assigning them to partitions. Each packet will match only one rule since the rules are non-overlapping.

To illustrate, consider the example in Figure 5.2. The boxes are sortable in XY field order. We first begin with the X field. The first field is always the simplest as all packets and rules are in the same active partition. We perform the basic merge between our sorted list of points and the sorted intervals of rules. Because we have pairs of rules that have identical intervals in field X,

namely $X(R1) = X(R3) = [1,3]$ and $X(R2) = X(R4) = [5,7]$, when point $P1$ matches interval $[1,3]$, we then place $P1$ into the same partition with both rule $R1$ and $R3$. At the end of the merge, we have the following partitions: partition 1 with points $P1$ and $P5$ and rules $R1$ and $R3$, partition 2 with points $P3, P6, P7$ and rule $R5$, and a null partition with the remaining points and rules. We now process the Y field. We start with our assumed global sorting of all the points in the Y field: $P6, P7, P5, P4, P2, P3, P1$. We first process point $P6$ and see it belongs to partition 2 which is in $\min(r)$ mode. We get the first rule $R5$ from partition 2 and return that $P6$ matches $R5$. We then get point $P7$ and see it belongs to partition 2 which is now in $\max(r)$ mode. The active rule is still $R5$, and we return that $P7$ matches rule $R5$. We then get point $P5$ and see it belongs to partition 1 which is in $\min(r)$ mode. We get the first rule $R3$ from partition 1 and, when we finish with point $P5$, we return that both $R3$ and $P5$ belong to the null partition. We then get point $P4$ and see it belongs to partition 2 which is still in $\max(r)$ mode. The active rule is still $R5$, and we return that $P4$ belongs to the null partition. We then get point $P2$ and see it belongs to partition 1 which is in $\min(r)$ mode. We get the current rule $R1$ and return that $P2$ matches $R1$. We then get point $P3$ and skip it since it is already in the null partition. Finally, we get point $P1$ and see it belongs to partition 1 which is still in $\max(r)$ mode. We get the current rule $R1$ and return that $P1$ matches $R1$.

Theorem 5.1.1. *For two dimensions, the Partition Projection Algorithm runs in $O(m+n)$ time given a sortable ruleset instance with m points and n rules where we can access the points in sorted order in each field and we have a data structure that gives us the boxes in sorted order in each field. In addition, if $m = \Theta(n)$, then per point running time is amortized $O(1)$.*

We will explore several ways to further speed up sorting of packets. First, we observe that we only need to perform the sorting once for each set of packets and can reuse the sorting for each partition. This leads to an even faster amortized result. Second, we will look for ways to speed up the sorting, particularly as we drop packets as they hit the null partition. Third, while some fields such as IPv6 addresses may be very large, if not all the bits are really used, we may be able to reduce the number of passes for radix sort. We also observe that many of the proposed fields for SDN and OpenFlow have small ranges and thus would allow sorting to take place in only one or

a few passes. Fourth, we observe that by reducing packet processing to sorting, we can leverage any sorting optimizations developed by other researchers. For example, there is a lot of ongoing work in developing fast sorting algorithms that exploit GPUs or other multi-core processors. As appropriate, we can plug any of these solutions into our solution if it will improve throughput.

Finally, we will consider potential "smart buffer" solutions where we do some of the sorting of packets ahead of time while the packets are waiting in the buffer. For example, if we have multiple processors, we might use idle cores to sort packets in the buffer while other packets are being processed. In particular, if we are in a system where we offer different levels of quality of service, we might proactively push lower priority traffic into a lower priority buffer where we presort the packets and then later process these packets using batch mode while higher priority packets are processed in single query mode as they arrive.

5.2 Integration with OpenFlow switches

We will integrate our ideas into two open source OpenFlow switches: Lagopus and Open vSwitch. We have already done a preliminary integration of 1 and 2 with Lagopus. However, Open vSwitch is more widely used. For this project, we propose to integrate all of our key ideas into Open vSwitch. There are two key challenges that we must overcome to successfully integrate our ideas into Open vSwitch. First, we must generalize PartitionSort to handle rules with arbitrary bitmasks. Second, we must develop caching schemes for PartitionSort similar to the megaflow caching scheme that Open vSwitch currently uses with its Tuple Space Search packet classifier. In addition to algorithmic challenges, other challenges include implementation and deployment issues. In particular, Open vSwitch uses multi-threading programming with RCU (read-copy-update) based on mutual exclusion. This requires significant amount of efforts to implement. Furthermore, Open vSwitch uses pipelined OpenFlow tables which are a wrapper abstraction from a single flow table. Finally, we must also deploy the switch and test on realisitc dataset to demonstrate the observe speed-up over Tuple Space Search version.

We now briefly describe pipeline processing of OpenFlow tables in OpenFlow switches. Ac-

cording to OpenFlow specification 1.5, the switch contains at least one flow table. Each flow table contains multiple flow entries (or rules). One can think of a flow table as one packet classification instance where the priority is resolved within the flow table. Actions are cumulative collecting from the first flow table. The match in first flow table may contain an instruction to direct a packet to the table with larger table id, but not less. The process will be repeated at the next flow table. At any table, if the match does not specify next table id to go to, then the look-up process terminate, and all accumulated actions are applied. Each flow table contains a subset of match fields from all possible matching fields. There are 13 required fields. Each required field must be used in at least one flow table.

We plan to address each challenges as follows. First, we plan to separate rules into two types: prefix rules, and non-prefix rules. We use PartitionSort to implement prefix rules, and use Tuple Space Search for non-prefix rules. There is a study of statistics of non-prefix rules showing that roughly 90% of rules are prefix. So, this simple strategy will be likely to be effective. Second, we plan to develop a new megaflow generation algorithms for PartitionSort. This requires an additional metadata about bitmask used for each field ordering in each sortable partitions. We will experiment with partitioning algorithms that generate maximal flow caching for ranged-based rules. Third, PartitionSort is compatible with pipelined processing in OpenFlow switches. In particular, we believe there is a room for further optimization where existing matching results on one flow table can be transferred to another flow table using a standard technique such as fractional cascading from computation geometry. Finally, we plan to use the OpenFlow ClassBench, the new simulated rulesets to run experiments, and plan to collect rule update traffic and sequences from open source SDN projects.

We give a formal interpretation of megaflow generation problem in OVS paper. We first define packet classification problem in OVS, which is a more general case. A packet field f is a variable whose domain $D(f)$ is a set of all possible words of a fixed length $\ell(f)$ from the set of alphabet $\{0, 1, *\}$. A rule r over d fields is a triplet of (M, q, a) where $M = (s_1^r \in f_1 \wedge s_2^r \in f_2 \wedge \ldots \wedge s_d^r \in f_d)$ is match fields from packet fields $f_1, \ldots, f_d$, q is a unique number for priority, and a is an action

which we can consider as a number without loss of generality. A packet p is a list of bitstrings $(p_1, p_2, \ldots, p_d)$ ordered by field f_i where in each field the bitstring has the same length as field length, $|p_i| = \ell(f_i)$ for $1 \leq i \leq d$. A packet p matches a rule r if for each field $s_i^r \in f_i$ and for each alphabet b at position j in s_i^r if $b \neq *$, then the corresponding bit in the packet of the same field must be the same, i.e., p_i at position j is equal to b. Two rules r_1 and r_2 are overlapped if there exists a packet matching both r_1 and r_2. Otherwise, we say the two rules are non-overlapped. In packet classification, if there are multiple matches for a given packet, then an action from the matching rule with the highest priority must be returned. Let R be a set of rules, define $A_R(p)$ to be a packet classification function that maps a packet p to an action from the matching rule with highest priority. An action can be empty if there is no match. We define flow cache F from a ruleset R as a set of non-overlapping rules R_F such that for all packet p, $A_{R_F}(p) = A_R(p)$ whenever $A_{R_F}(p) \neq \emptyset$.

Next, we formulate an online megaflow generation according to OVS paper. Let n be number of rules in ruleset R. Online megaflow generation is to transform a regular packet classifier into a cache-aware packet classifier that can efficiently generate a new flow cache with broad coverage. More precisely, given a packet classifier T with classification time $T(n, d)$, we want to transform it into a new packet classifier that can generate a new rule r from any packet p with action $a = A_R(p)$ such that (1) r is a flow cache, (2) the overhead is a constant factor from usual classification time, i.e., extra running time is $O(T(n, d))$. The objective is to maximize number of wildcarded bits (i.e., $'*'$ bits) of the new rule for a given packet.

OVS uses TSS for a packet classifier. It has several optimizations for maximizing number of wildcarded bits. The main principle is by tracking bits of packets. In particular, if we know that certain bits are never used during packet classification process, then we can generate a new rule in which we can wildcard such bits, i.e., mark certain positions with alphabet $*$.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] David E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, 2005.

[2] P. Gupta and N. McKeown. Algorithms for packet classification. *Netwrk. Mag. of Global Internetwkg.*, 15(2):24–32, March 2001.

[3] Pankaj Gupta, , Pankaj Gupta, and Nick Mckeown. Packet classification using hierarchical intelligent cuttings. pages 34–41, 1999.

[4] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. pages 213–224, 2003.

[5] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. Packet classification algorithms: From theory to practice. pages 648–656, 2009.

[6] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. *SIGCOMM Comput. Commun. Rev.*, 40(4):207–218, August 2010.

[7] Peng He, Gaogang Xie, Kave Salamatian, and Laurent Mathy. Meta-algorithms for software-based packet classification. In *Proceedings of the 2014 IEEE International Conference on Network Protocols*, INCP '14, pages 308–319, 2014.

[8] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. Sax-pac (scalable and expressive packet classification). *SIGCOMM Comput. Commun. Rev.*, 44(4):15–26, August 2014.

[9] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[11] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. pages 117–130, 2015.

[12] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. *What You Need to Know About SDN Flow Tables*, pages 347–359. Springer International Publishing, Cham, 2015.

[13] Haim Kaplan, Eyal Molad, and Robert Endre Tarjan. Dynamic rectangular intersection with priorities. In *STOC*, pages 639–648. ACM, 2003.

[14] Peyman Afshani, Lars Arge, and Kasper Green Larsen. Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. pages 323–332, 2012.

[15] V. Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *Proceedings of the ACM SIGCOMM*, pages 135–146, 1999.

[16] Kang Li, Francis Chang, Damien Berger, and Wu-chang Feng. Architectures for packet classification caching. In *ICON*, pages 111–117. IEEE, 2003.

[17] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[18] Vijay K. Vaishnavi. Multidimensional balanced binary trees. *IEEE Trans. Comput.*, 38(7):968–985, July 1989.

[19] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[20] Roberto Grossi and Giuseppe F. Italiano. *Efficient Techniques for Maintaining Multidimensional Keys in Linked Data Structures (Extended Abstract)*, pages 372–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[21] David E. Taylor and Jonathan S. Turner. Classbench: A packet classification benchmark. In *Proceedings of the IEEE Infocom*, March 2005.

[22] James Daly and Eric Torng. Tuplemerge: Building online packet classifiers by omitting bits. In *ICCCN*, pages 1–10. IEEE, 2017.

[23] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *INFOCOM*, pages 2645–2653. IEEE, 2018.

[24] Tong Yang, Alex X. Liu, Yulong Shen, Qiaobin Fu, Dagang Li, and Xiaoming Li. Fast openflow table lookup with fast update. In *INFOCOM*, pages 2636–2644. IEEE, 2018.

[25] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[26] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[27] Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Trans. Algorithms*, 5(3):28:1–28:51, July 2009.

[28] Pankaj K. Agarwal, Lars Arge, and Ke Yi. An optimal dynamic interval stabbing-max data structure? pages 803–812, 2005.

[29] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[30] Haim Kaplan, Eyal Molad, and Robert E. Tarjan. Dynamic rectangular intersection with priorities. pages 639–648, 2003.

[31] Kurt Mehlhorn and Mark H. Overmars. Optimal dynamization of decomposable searching problems. *Information Processing Letters*, 12(2):93 – 98, 1981.

[32] M.H. Overmars and J. van Leeuwen. Two general methods for dynamizing decomposable searching problems. *Computing*, 26(2):155–166, 1981.

[33] Herbert Edelsbrunner and Mark H Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6(4):515 – 542, 1985.

[34] Banit Agrawal and Timothy Sherwood. Modeling TCAM power for next generation network devices. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 120– 129, 2006.

[35] Cristian Lambiri. Senior staff architect IDT, private communication. 2008.

[36] Alex X. Liu and Mohamed G. Gouda. Complete redundancy removal for packet classifiers in TCAMs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21(4):424–437, 2010.

[37] Alex X. Liu, Chad R. Meiners, and Yun Zhou. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proceedings of the 27th Annual IEEE Conference on Computer Communications (Infocom)*, pages 574–582, April 2008.

[38] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proceedings of the ACM SIGCOMM*, pages 193 – 204, August 2005.

[39] Qunfeng Dong, Suman Banerjee, Jia Wang, Dheeraj Agrawal, and Ashutsh Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proceedings of the ACM Sigmetrics*, pages 311–322, 2006.

[40] Huan Liu. Efficient mapping of range classifier into Ternary-CAM. In *Proceedings of the Hot Interconnects*, pages 95– 100, 2002.

[41] Jan van Lunteren and Ton Engbersen. Fast and scalable packet classification. *IEEE Journals on Selected Areas in Communications*, 21(4):560– 571, 2003.

[42] Derek Pao, Yiu Li, and Peng Zhou. Efficient packet classification using TCAMs. *Computer Networks*, 50(18):3523–3535, 2006.

[43] Derek Pao, Yiu Keung Li, and Peng Zhou. An encoding scheme for TCAM-based packet classification. In *Proceedings of the 8th IEEE International Conference on Advanced Communication Technology (ICACT)*, February 2006.

[44] Ed Spitznagel, David Taylor, and Jonathan Turner. Packet classification using extended TCAMs. In *Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP)*, pages 120– 131, November 2003.

[45] Anat Bremler-Barr and Danny Hendler. Space-efficient TCAM-based classification using gray coding. In *Proceedings of the 26th Annual IEEE Conference on Computer Communications (Infocom)*, May 2007.

[46] James Daly, Alex X. Liu, and Eric Torng. A difference resolution approach to compressing access control lists. *IEEE/ACM Transactions on Networking*, In press, 2015.

[47] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 18(2):490–500, 2010.

[48] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 20(2):488–500, April 2012.

[49] Chad R. Meiners, Alex X. Liu, and Eric Torng. Topological transformation approaches to tcam-based packet classification. *IEEE/ACM Transactions on Networking*, 19(1):237–250, February 2010.

[50] Eric Norige, Alex X. Liu, and Eric Torng. A ternary unification framework for optimizing tcam-based packet classification systems. *IEEE/ACM Transactions on Networking*, In press, 2015.

[51] Alex X. Liu, Eric Torng, and Chad R. Meiners. Compressing network access control lists. *IEEE Transactions on Parallel and Distributed Systems*, 22(12):1969 – 1977, December 2011.

[52] Chad R. Meiners, Alex X. Liu, and Eric Torng. Topological transformation approaches to optimizing tcam-based packet processing systems. In *Proceedings of the ACM SIGMETRICS*, August 2009.

[53] Chad R. Meiners, Alex X. Liu, and Eric Torng. Algorithmic approaches to redesigning tcam-based systems (extended abstract). In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Annapolis, Maryland, June 2008.

[54] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *Proceedings of the 17th IEEE Conference on Network Protocols (ICNP)*, October 2009.

[55] Chad R. Meiners, Alex X. Liu, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. In *Proceedings of the 15th IEEE Conference on Network Protocols (ICNP)*, pages 266–275, October 2007.

[56] James Daly, Alex X. Liu, and Eric Torng. A difference resolution approach to compressing access control lists. In *Proceedings of the 32th Annual IEEE Conference on Computer Communications (INFOCOM)*, Turin, Italy, April 2013.

[57] Alex X. Liu, Eric Torng, and Chad Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *Proceedings of the 27th Annual IEEE Conference on Computer Communications (Infocom)*, pages 691–699, Phoenix, Arizona, April 2008.

[58] Chad R. Meiners, Alex X. Liu, Eric Torng, and Jignesh Patel. Split: Optimizing space, power, and throughput for tcam-based classification. In *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 200–210, Brooklyn, New York, October 2011.

[59] Eric Norige, Alex X. Liu, and Eric Torng. A ternary unification framework for optimizing tcam-based packet classification systems. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, San Jose, California, October 2013.

[60] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[61] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using gpus in software packet processing. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 409–423, Berkeley, CA, USA, 2015. USENIX Association.

[62] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and T.V. Lakshman. Multi-layer packet classification with graphics processing units. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 109–120, New York, NY, USA, 2014. ACM.

[63] Yun R. Qu, Hao H. Zhang, Shijie Zhou, and Viktor K. Prasanna. Optimizing many-field packet classification on fpga, multi-core general purpose processor, and gpu. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 87–98, Washington, DC, USA, 2015. IEEE Computer Society.

[64] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference*, pages 101–112. USENIX Association, 2012.

[65] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, Dario Rossi, and Jerome Tollet. Batched packet processing for high-speed software data plane functions. In *INFOCOM Workshops*, pages 1–2. IEEE, 2018.

[66] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue B. Moon. Packetshader: a gpu-accelerated software router. In *SIGCOMM*, pages 195–206. ACM, 2010.

[67] Matteo Varvello, Rafael P. Laufer, Feixiong Zhang, and T. V. Lakshman. Multi-layer packet classification with graphics processing units. In *CoNEXT*, pages 109–120. ACM, 2014.

[68] Saladi Rahul. Improved bounds for orthogonal point enclosure query and point location in orthogonal subdivisions in r3. pages 200–211, 2015.

[69] H. B. Acharya, Satyam Kumar, Mohit Wadhwa, and Ayush Shah. Rules in play: On the complexity of routing tables and firewalls. In *ICNP*, pages 1–10. IEEE Computer Society, 2016.

[70] Xuehong Sun, Sartaj K. Sahni, and Yiqiang Q. Zhao. Packet classification consuming small amount of memory. *IEEE/ACM Trans. Netw.*, 13(5):1135–1145, October 2005.

[71] Hao Che, Zhijun Wang, Kai Zheng, and Bin Liu. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Transactions on Computers*, 57(7):902–915, July 2008.

[72] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking, in submission*.

[73] Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

[74] V. K. Vaishnavi. Multidimensional balanced binary trees. *IEEE Transactions on Computers*, 38(7):968–985, Jul 1989.

[75] Ju Yuan Hsiao, Chuan Yi Tang, and Ruay Shiung Chang. An efficient algorithm for finding a maximum weight 2-independent set on interval graphs. *Inf. Process. Lett.*, 43(5):229–235, October 1992.