

# Tradeoffs for Packet Classification

Anja Feldmann

S. Muthukrishnan

AT&T Labs–Research

Florham Park, NJ

{anja,muthu}@research.att.com

*Abstract*—We present an algorithmic framework for solving the packet classification problem that allows various access time vs. memory tradeoffs. It reduces the multi-dimensional packet classification problem to solving a few instances of the one-dimensional IP lookup problem. It gives the best known lookup performance with moderately large memory space. Furthermore, it efficiently supports a reasonable number of additions and deletions to the rulesets without degrading the lookup performance. We perform a thorough experimental study of the tradeoffs for the two-dimensional packet classification problem on rulesets derived from datasets collected from AT&T WorldNet, an Internet Service Provider.

## I. INTRODUCTION

While the current Internet offers best-effort service, future IP networks will provide enhanced services to its users. Such services may include differentiated services underwritten by service level agreements (SLA's), fine grain quality of service (QoS), Virtual Private Network (VPN) service, distributed firewalls, IP security gateways, traffic based billing, etc. All such enhancements need *packet classification*, that is, determining which flow a packet belongs to based on one or more fields in the packet header. Packet header fields that may be used for classification include the destination and source IP addresses, the protocol type, and the source and destination port numbers. Rules for classification are specified by specifying valid ranges for any of the header fields. Determining the most fitting rule for each packet is the packet classification problem.

**Examples of packet classification:** In today's IP networks, packet classification for routing is based purely on the destination IP address, and the rules are expressed as IP address prefixes. Packet classification rules for access lists and firewalls use two IP prefixes – source and destination IP address prefixes – and sometimes conditions on the port numbers and protocols. IP networks that use improved traffic engineering and support differentiated services may use one set of packet classification rules to determine which router to forward a packet (using MPLS tunnels) and another set of rules to determine which queue to use (for differentiated service) and yet another set of rules to decide whether to forward a packet (for VPNs and IP security gateways). Such rules are likely to be based on at least the source and destination IP addresses but may also include port numbers<sup>1</sup>.

**Requirements for packet classification:** They can vary widely depending on the application and where packet classification is performed in the network.

- *Resource limitations:* Packet classification solutions can trade-off time to perform the classification per packet vs. memory

used. At large corporate campuses, access speeds may range from medium speed of T3 and OC3, to top speeds of OC12 and above. At inter ISP boundaries, the access speeds will be OC12, OC48, and above. Residential customers have access speeds of T1 (DSL) or less. Solutions should achieve the required target access speed, while minimizing the amount of memory used.

- *Number of rules to be supported:* Packet classification applications differ in the number of rules that are specified. Today typical firewalls may specify a few hundred rules, while an access/backbone router may have a few hundreds of thousands of rules; these numbers are expected to scale up with enhanced services and router throughput and may reach millions of rules.

- *Number of fields used:* Packet classification applications differ on the number of fields (*dimensions*) of the IP header that is used for classification. Current routers use one field (destination IP address), but it is expected that emerging routers will use two-dimensional rules. Firewalls and other access list applications may use a few more fields [9].

- *Nature of rules:* Current routers use rules with a prefix mask on destination IP addresses. However, more general masks such as arbitrary ranges are expected to become permissible. Packet classification solutions need to accommodate such general specification.

- *Updating the set of rules:* The number of changes to the rules either due to a route or policy change is moderate to small compared to the number of packets that an application, e.g., a router, needs to classify in the same time period. Packet classification solutions must adapt gracefully and quickly to such updates without sacrificing the access performance. Rebuilding major parts of the data structure for every update is prohibitive.

- *Worst case vs. Average case:* There is a widely held view that for access time performance of packet classification, one must focus on worst case, rather than average case [11].

Many of these requirements have been articulated in the extensive collection of papers that have addressed the packet classification problem [1], [9], [8], [11], [12], [14], [15] and the references therein. Several solutions have been proposed; they are effective in meeting some of the criteria, but not all of them. Some of them do not allow range specification in the rules [8], others do not preserve the access time performance with guaranteed small update times [9], [11], [15], and yet others are not designed for large rulesets [12], [14]. It is desirable to have a suite of solutions with a range of tradeoffs that can be tuned to particular applications.

**Our contributions:** We present a novel algorithmic framework for solving the packet classification problem. It relies on insights from computational geometry, and has the following interesting

<sup>1</sup>Any communication that uses IP-Sec encryption will not expose port numbers. Therefore relying on the availability of port numbers in backbone IP networks may be problematic.

features. (1) It allows various access time vs. memory space tradeoffs, and can be engineered for different applications. In particular, it gives the currently best known access times for the packet classification problem with moderate amount of memory use. (2) It reduces the packet classification problem (with arbitrary ranges and  $d$ -dimensions,  $d \geq 1$ ) to a small number of invocations of a specific one-dimensional packet classification problem, namely, the *IP Lookup* problem in which all rules are prefix ranges. Many optimized software [5], [12], [17] and hardware solutions (e.g., [10]) are known for the IP Lookup problem. Using our framework, these are now applicable to the general packet classification problem. (3) It allows efficient updates to the ruleset without recomputing the full data structure. Moderate amount of updates (additions as well as deletions of rules) do not effect the access times significantly; this is the first such solution for the packet classification problem.

We test our two-dimensional algorithm on rulesets (source/destination ranges) derived from flow traces collected at multiple backbone routers of an ISP, AT&T WorldNet. The results show that our algorithm can perform packet classification for rulesets of size beyond  $10^6$  with at most 18 memory accesses (assuming that each memory access retrieves a full 32 bit cache-line [15]) in the worst case. The space to store the data structure is at most a factor of 5 bigger than the space needed to store the rules themselves. No previous experimental result considered real datasets of this magnitude. The number of memory accesses can be further reduced by using bigger cachelines or larger memory.

At the technical core, our algorithmic framework relies on data structural solutions [7] in which the access time is aggressively minimized (to roughly  $O(\log \log U)$  where  $U$  is the range of IP addresses). We modify their algorithm to eliminate many of the steps with inherently large constants. Consequently we have obtained a simple solution which we further extend to allow dynamic updates. The theoretical solution could use moderately large memory space (e.g.,  $O(n^{1+\epsilon})$  for  $1 > \epsilon > 0$  where  $n$  is the number of rules). However, our experiments show that the memory usage of our algorithm is quite reasonable on realistic datasets.

**Organization:** We define the packet classification problem in Section II and next present our algorithmic framework for static (Sections III – V), as well as dynamic cases (Section VI). We discuss engineering tradeoffs in Section VII. In Section VIII, we focus on the two-dimensional problem and present extensive experimental results. We review related work in Section IX. Concluding remarks and future work are in Section X.

## II. PROBLEM SPECIFICATION

Informally, the packet classification problem identifies the flow a packet belongs to, based on one or more fields in the packet header. Formally, the *d-dimensional packet classification problem* (denoted as PC problem) is as follows. We are given a set  $\mathcal{R} = \{r_1, \dots, r_n\}$  of rules over  $d$  fields (dimensions). Each rule consist of a set of ranges  $r_i = [F_1^i, \dots, F_d^i]$ , where  $F_j^i$  is a range (interval) of values the field  $j$  may take; each rule also has a *cost*. The set of rules may be preprocessed. Queries are presented on line. Each query is a packet  $p = [f_1, \dots, f_d]$ , where each  $f_i$  is a singleton value. A rule  $r_i$  applies to a packet  $p$

Reference	Space used	# of memory accesses
[5]	$O(n)$	$O(\log_2 U)$
[17]	$O(n \log n)$	$O(\log_2 \log_2 U)$
[12]	$RL_s(2n, U)$	$RL_t(2n, U)$

TABLE I  
PERFORMANCE BOUNDS FOR THE IPL PROBLEM

if for all dimensions  $k$ , the field value  $f_k$  of packet  $p$  lies in the range  $F_k^i$ . The problem is to determine the least cost rule that applies to the packet.<sup>2</sup> For example, in layer-four switching, the dimensions could consist of the source address, destination address, source port, and destination port. A rule such as [135.207.\*, 12.\*, 1024–65535, 20–23] may be used to allow IP addresses within AT&T Labs-Research to contact IP addresses within WorldNet either via ftp, ssh, or telnet. Hosts from AT&T Labs-Research are restricted to use any of the non-private ports. Depending on how costs are assigned to rules one can model different flavors of the PC problem (see [9]).

The rules have a natural geometric interpretation in  $d$  dimensions. Each rule  $r_i$  can be thought of as a “hyperrectangle” in  $d$  dimensions (called rectangles henceforth), obtained by the cross product of the intervals  $F_j^i$  along each of the dimensions  $j$ . Thus the set of rules  $\mathcal{R}$  now corresponds to a set of rectangles in  $d$  dimensions. Each packet  $p$  corresponds to a point in  $d$  dimensions, and the PC problem is identical to the *rectangle enclosure problem*, that is, given a set of rectangles  $\mathcal{R}$ , determine the least cost rectangle that encloses any query point  $p$ .

In solving the PC problem, the parameters of interest are storage space and the number of memory accesses performed per query (which is the dominating lookup cost).

## III. ONE-DIMENSIONAL CLASSIFICATION

The one-dimensional PC problem is: given a set of  $n$  rules – possibly overlapping intervals from  $[1 \dots U]$  – each with a cost, answer lookup queries for point  $q \in [1 \dots U]$  by identifying the smallest cost rule that contains  $q$ .

**Special case one:** The *IP Lookup* (IPL) problem is a subproblem of the general PC problem in which each range is a prefix of an IP address (IP addresses are in  $[1 \dots U]$ , with  $U = 2^{32}$  for IPv4). Each query  $q$  is an IP address. The task is to determine the least cost rule that is a prefix of  $q$ . The IPL problem is the classical PC problem based on the destination IP addresses. The worst case number of memory accesses needed to solve this problem is denoted by  $IPL_t(n, U)$  and the space used by  $IPL_s(n, U)$ . The best known performance bounds for the IPL problem are in Table I. (RL will be defined shortly.)

**Special case two:** The *Range Location* (RL) problem is another subproblem of the general PC problem in which the ranges are non-overlapping and completely cover the universe  $1 \dots U$ . The collection of intervals of RL can be specified as series of left end points of the intervals in the sorted order. Each query  $q$  is an integer, and the goal is to determine the interval that contains

<sup>2</sup>For debugging purposes, it may be useful to enumerate all rules that apply to a given packet and not merely return the one with the smallest cost. All our solutions can be extended to determine such a output, and we do not consider this version of the problem any further.

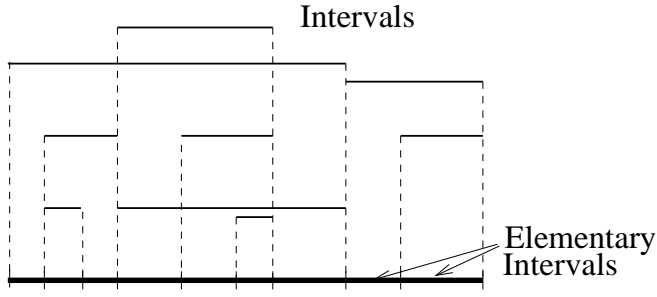


Fig. 1. Example: elementary intervals.

$q$ . The worst case number of memory accesses needed to solve this problem is  $RL_t(n, U)$  and the space used for this solution is  $RL_s(n, U)$ .

For us, the RL problem arises because our overall approach for solving the PC problem relies on reducing them to a few instances of the RL problem. For one-dimensions, this reduction is immediate. Given a set of intervals for the PC problem, consider the set of all the endpoints of intervals (and 1 and  $U$  are included by default). The region between any two such *consecutive* points in sorted order is called an *elementary interval*. Figure 1 illustrates an example. No two elementary intervals overlap. We can process each rule in the PC problem and retain the smallest cost rule that intersects each elementary interval. This reduces the PC problem on  $n$  rules to the RL problem with  $\leq 2n$  elementary intervals [12]. The universe size for both problems is  $U$ . Therefore it is sufficient to solve the RL problem to solve the PC and the IPL problem (Table I).

**Solving RL problem:** The best known theoretical result for solving the RL problem takes  $O(\log \log U)$  memory accesses using space  $O(n)$  with a large preprocessing time [18]; the constants involved are of moderate size. A different approach [12] is to do a multiway ( $k$ -way) search if  $k \log U$  bit operations can be performed with a single memory lookup, in which case at most  $\log_{k+1} n + 1$  memory accesses are needed. This is a very simple and memory efficient solution if  $n$  is not too large. Yet another approach that we explore here is to reduce the RL problem to the IPL problem.

*Theorem III.1:* Consider any instance  $I$  of the RL problem with  $N$  points in the range  $1 \cdots U$ . We can derive an instance  $I'$  of IPL with at most  $2N$  prefixes, each a string of length at most  $a = \lceil \log U \rceil$ . Each query  $i$  for the RL problem can be transformed into an IP address of length at most  $a$  for the IPL problem on set  $I'$ .

**Proof:** The reduction goes as follows. Say  $S_j$  is the bit representation of an integer  $j$  in  $I$ . Integer  $j$  and binary string  $S_j$  are treated interchangeably. We build a trie of the  $S_j$ 's. Let  $\sigma(u)$  denote the string labeling the path from the root to an internal node  $u$ . For each internal node  $u$  in the trie, we compute the smallest integer in  $I$  that is  $\geq \sigma(u) \parallel 0^{a-|\sigma(u)|}$  (denoted  $u_{\geq}$ ) and the largest integer in  $I$  that is  $\leq \sigma(u) \parallel 1^{(a-|\sigma(u)|)}$  (denoted  $u_{\leq}$ ). Here  $a \parallel b$  denotes string concatenation, and  $a^l$  represents the string obtained from  $a$  by repeating it  $l$  times. We generate set  $I'$  from  $I$  by generating two prefixes  $\sigma(u) \parallel 0$  and  $\sigma(u) \parallel 1$  for each internal node  $u$ . Since the number of internal nodes is at most  $N$ , the number of prefixes in  $I'$  is at most  $2N$ .

Figure 2 shows the trie (thick lines) for an example

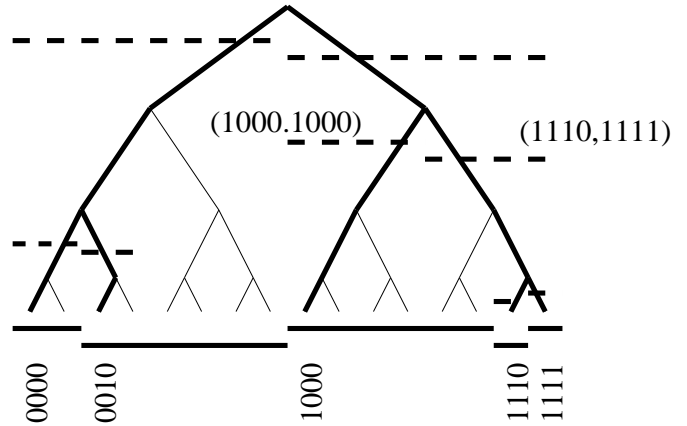


Fig. 2. Example: reduction of RL to IPL.

Reference	Space used	# of memory accesses
[12]	$2n$	$\log_{k+1} n + 1$ for $k$ -word cacheline
[18]	$O(n)$	$O(\log \log U)$
This paper	$IPL_s(2n, U)$	$IPL_t(2n, U)$

TABLE II

PERFORMANCE BOUNDS FOR THE RL PROBLEM

RL problem:  $\{0000, 0010, 1000, 1110, 1111\}$ . The resulting instance of the IPL problem consists of the prefixes  $\{000x, 001x, 0xxx, 1xxx, 10xx, 11xx, 1110, 1111\}$  (dashed lines). For the prefixes  $0xxx$  and  $1xxx$  the figure shows the two integers associated with each prefix.

Consider the query  $i$  in the RL problem. Let  $s(i)$  be the  $a$  bit string representation of  $i$ . Solve the IPL problem with  $s(i)$  on  $I'$  and say the solution is  $\sigma(v)$  for some internal node  $v$ . If  $i \leq v_{\geq}$ , then the solution to the RL problem is the interval ending at  $v_{\geq}$ ; otherwise,  $i \geq v_{\leq}$  in which case the solution is the interval starting at  $v_{\leq}$ . The proof is presented in the full version [6]. ■

The result is important to us since

- it gives a reduction from one-dimensional PC with arbitrary range rules to PC with only prefix rules (IPL) while increasing the number of rules by at most a factor of two. In contrast, the best previous reduction [15] uses a factor of  $O(\log U)$ .
- one can leverage off the best known solutions (hardware or software) for the IPL problem to solve the RL problem and the one-dimensional PC problem.

Table II summarize the performance bounds known for the RL problem; which of the results performs best in practice depends on the instance (relative values of  $n, U$  etc.). Combining this with results for the IPL problem, we can conclude that there are practical solutions for the one dimensional PC problem with  $O(\log \log U)$  memory accesses for each query; this is  $O(\log w)$  since  $w = \log U$  bit IP addresses are needed to specify numbers in the range  $1 \cdots U$ .

#### IV. TWO-DIMENSIONAL CLASSIFICATION

We view the two-dimensional PC problem in its geometric terms. Let  $\mathcal{R}$  be a set of  $n$  rectangles on a two-dimensional grid  $[1 \cdots U, 1 \cdots U]$  each with some cost. We want to preprocess  $\mathcal{R}$

such that we can efficiently answer the following query: Given a two-dimensional grid point  $q$ , find the smallest cost rectangle in  $\mathcal{R}$ , if any, that contains  $q$ . The basic data structure of our algorithm is a FIS tree.

**FIS trees:** Given a collection of segments our base data structure is a segment tree [4], but made “fat” to occupy a given number of levels  $\ell$  as in [7]. In addition, we invert it so directed paths go from leaves to the root which helps the search as described later. We call the data structure the *FIS tree* (*Fat, Inverted, Segment tree*) and describe it below.

Let  $\mathcal{S}$  be a set of  $m$  segments. The endpoints of the segments partition the universe into a number of elementary intervals as in Section III. Say  $t = m^{1/\ell}$ ,  $\ell > 1$ , and  $t, \ell$  and  $m$  are integers. The FIS tree is a balanced, inverted  $t$ -ary tree  $\mathcal{T}$  with  $\ell$  levels. Each node  $\nu$  has a pointer to its parent  $\text{parent}(\nu)$  and at most  $t$  incoming arcs. The leaves of the FIS tree correspond to the elementary intervals in order. An internal node  $\nu$  corresponds to the larger interval that is the union of the elementary intervals stored at its leaves. We denote this interval by  $I(\nu)$ . A segment  $s \in \mathcal{S}$  is stored at the node  $\nu$  if  $I(\nu) \subset s$  but  $I(\text{parent}(\nu)) \not\subset s$ . The set of segments stored with a node is called its *canonical set*.

*Property IV.1:* [7] A FIS-tree  $\mathcal{T}$  on  $m$  segments has the following properties. (1) The depth is  $\log m / \log t = \ell$ . (2) Each segment is stored in at most  $2t - 1$  nodes per level. (3) The collection of segments containing any point  $p$  is the union of  $\ell$  sets, namely the canonical subsets of the nodes on the search path of  $p$  in  $\mathcal{T}$ ; these sets are disjoint. ■

**Preprocessing:** Consider the rectangles of a two-dimensional PC problem and their projections along the  $x$  and  $y$  axes; we use these projected segments synonymously with the rectangles themselves. We build a FIS tree on the  $x$ -projections of the rectangles that we call the *x-FIS tree*. We call the canonical sets associated with the nodes of the  $x$ -FIS tree as the *x-canonical sets*. We also store the left most endpoints of the elementary intervals and a pointer to the leaf representing that elementary interval with each endpoint; we call this the *x-set*. We consider the  $y$ -projections of the rectangles of the  $x$ -canonical set of  $\nu$  and their elementary intervals. We store the left endpoints of each such interval together with the smallest cost rectangle that intersects each, in a set we call the *y-set* of  $\nu$ .

An example consisting of ten two-dimensional rules is shown in Figure 3. Each rule is represented as a rectangle (the rule number is shown in the upper left corner of the rectangle). The projections of the rectangles onto the  $x$ -axis (dotted lines) results in the set of elementary intervals shown in the figure. In this example we constructed an  $x$ -FIS tree with  $\ell = 3$  levels. Each leaf node of the tree corresponds to a single elementary interval, while the root of the tree corresponds to the full interval covered by all rectangles. Let us consider which rules will be inserted into the  $x$ -canonical sets of the second node from the left (labeled with  $\{3, 4\}$ ); say, this node is denoted  $u$ . Since rectangles 4 and 3 are fully contained in the interval covered by the parent of  $u$ , they are inserted into the  $x$ -canonical set of  $u$ . Since rectangle 1 contains the interval associated with the parent of  $u$ , 1 is not in the  $x$ -canonical set of  $u$ . Let us consider the second node on the second level of the FIS tree; say, that is denoted  $v$ .

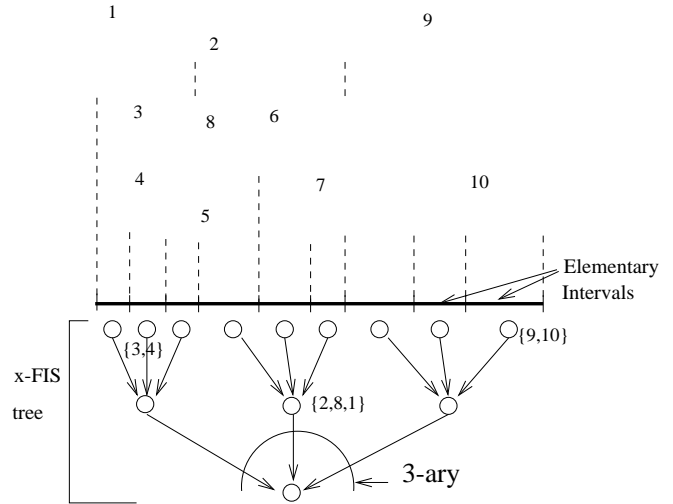


Fig. 3. Example: construction of a FIS tree.

Rectangles 1, 2 and 8 all contain the interval that corresponds to  $v$  but they do not contain the full interval (corresponding to its parent); hence, they are all in the  $x$ -canonical set of  $v$ . Rectangles 5, 7, 6, 8 contain subsets of the interval associated with  $v$ , and are therefore inserted into the  $x$ -canonical sets of a lower level. As described in the text,  $y$ -FIS trees are constructed on each of these  $x$ -canonical sets.

**Query Processing:** For a given query point  $p$  we have to search several FIS trees. Traditionally each such search is done by walking down the tree from the root, at each node using  $p$  to determine which child to pursue. If  $t$  is a small constant (such as for binary trees where  $t = 2$ ), this takes only  $O(1)$  time per node. However, for arbitrarily large  $t$ , we can no longer decide in  $O(1)$  time in which child of a given node  $\nu$  we have to continue the search. For this we need to perform a search among the intervals of the  $t$  children of  $\nu$ . This problem is precisely the RL problem on the left endpoints of the intervals represented by each child. Thus the problem of searching a  $\ell$  leveled FIS tree can be done using  $\ell$  instances of RL [7]. However, we take advantage of the inverted structure of the FIS tree to speed this search significantly; in fact, we reduce it to a *single* instance of the RL problem as explained below. Say the two-dimensional query point is  $q = (q_x, q_y)$ .

1. We solve the RL problem on the  $x$ -set with query  $q_x$ . This returns the pointer to the leaf  $L_x$  in the  $x$ -FIS tree representing the elementary interval containing  $q_x$ .
2. We consider all parents of  $L_x$  by following successive parent pointers. The total memory accesses is at most  $\text{RL}_t(2n, U)$  since the parent pointer location of a node can be computed arithmetically<sup>3</sup>.
3. We search the  $y$ -sets associated with each parent of  $L_x$  by solving the RL problem with  $q_y$  for each one. This determines the set of elementary interval that contain  $q$  from all  $y$ -sets. The smallest cost rectangle associated with these elementary intervals is returned as the solution. This may be thought of as solving the one-dimensional problem on the  $y$ -sets of the parents of  $L_x$ , using FIS trees of only one level.

<sup>3</sup>An additional  $\ell$  memory accesses may be needed when cascading many FIS trees.

Algorithm	Space Used	# of mem accesses
Grid of tries [15]	$O(nw)$	$O(\log w + w)$
Rectangular Search [14]	$O(n\sqrt{w})$	$O(w)$
Range Matching [11]	$O(n)$	$O(\log n + w)$
FIS trees $\ell = 3$	$O(n^{4/3})$	$4 \text{RL}_t(n, U) + 3$ $\sim 4 \log w$
cacheline $k, \ell = 3$	$O(n^{4/3})$	$4 \log_{k+1} n$
FIS tree, $\ell$	$O(n^{1+1/\ell})$	$(\ell + 1) \text{RL}_t(n, U)$ $\sim (\ell + 1) \log w$

TABLE III

TWO-DIMENSIONAL PC PROBLEM ( $w = \log U$ )

*Theorem IV.2:* Say the  $x$ -FIS tree has  $\ell$  levels. Our data structure for the two-dimensional PC problem uses  $O(\ell n^{1+1/\ell})$  space and takes at most  $(\ell + 1) \text{RL}_t(2n, U)$  memory accesses per query.

**Remark:** Say  $\ell = 2$ . Our data structure uses  $O(n^{3/2})$  space, worst case, and at most  $3 \text{RL}_t(2n, U)$  memory accesses. If  $\ell = 3$ , space used is  $O(n^{4/3})$  and number of memory accesses is at most  $4 \text{RL}_t(2n, U)$ . ( $2 \text{RL}_t(n, U)$  is essentially a lower bound for the two-dimensional PC problem since the RL problem has to be solved for each dimension at least once.)

**Comparison:** A comparison of our results with the previous ones is shown in Table III. All bounds are worst case bounds. While our solutions have the smallest number of memory accesses (order of  $\log w$  rather than order of  $w$ ), the worst case memory is moderately large.

## V. MULTI-DIMENSIONAL CLASSIFICATION

In the multi-dimensional PC problem, we are given a set  $\mathcal{R}$  of  $d$ -dimensional rectangles each with a cost. The query is a  $d$ -dimensional point  $q$  and the goal is to determine the rectangle of smallest cost that contains  $q$ . Our solution extends the two-dimensional approach. We first construct a FIS tree on the first dimension and then recursively construct our data structure on the remaining  $d - 1$  dimensions for each of the canonical sets in this FIS tree. The FIS tree for the last dimension will be of level one just as in the two-dimensional case. It follows,

*Theorem V.1:* If each FIS tree has  $\ell$  levels, our data structure for the multi-dimensional PC problem takes  $O(n(\ell n^{1/\ell} \log n)^{d-1})$  space and at most  $\ell^{d-1} \text{RL}_t(2n, U)$  memory accesses per query.

The known theoretical solutions would solve the RL problem  $\ell^d$  times in the worst case for each query. As in the two-dimensional case the inverted FIS tree lets us save  $\ell$  RL invocations for each FIS tree search. Our framework gives the best known results in terms of memory accesses for general rulesets. In the worst case the memory usage may be large. In practice, rulesets have a lot of structure [9], [14]. In particular many dimensions cluster naturally. Instead of using the full data structure with  $\ell$  levels for each dimension we can take advantage of the clustering and reduce the levels for those dimensions. In the dimensions that do not have a natural clustering, our framework provides an efficient way to structure the canonical sets and to explore the tradeoffs. Combined with other engineering insights this should lead to moderate memory usage and result in a small number of memory accesses.

## VI. DYNAMIC PC PROBLEM

In this section, we consider updates to the rulesets. If the only allowed updates are *inserts*, we refer to it as the *incremental* PC problem. If *deletes* are allowed as well, we refer to that as the *dynamic* PC problem. In this section we extend our data structures to allow for insertions and deletions, using a simple approach. We also explore how many updates can be accommodated without significantly degrading lookup performance. In general, it is more important to minimize lookup times rather than update times since IP routing protocols do not guarantee instantaneous convergence. Even within a single router they may not be instantaneous on all interfaces due to the distribution of information to the different linecards. Routing table updates do not have to be atomic but they should be incremental rather than requiring re-computation of the full data structure and they should preserve a consistent view for each rule and not introduce route flaps.

**Dynamic RL problem:** As in the static case, our approach involves reducing the problem to the RL problem. However, we now use the *dynamic* RL problem (DRL) defined as follows. Given a set of non-overlapping intervals that cover the domain  $1 \cdots U$ , the problem is to support operations: *split* an interval into two adjacent intervals, *merge* two contiguous intervals into one, and *lookup* a query point and return the interval that contains it. We solve the DRL problem using a  $B$ -tree with multi-key search on the left end points of the intervals in time  $O(\log_{k+1} n)$  where  $n$  is the number of rules and  $k$ -word cachelines are used. This is the most practical solution that we are aware of; using van Emde Boas trees [16] this problem can be solved with  $O(\log \log U)$  memory accesses but the constants are large. We estimate that our solution to the DRL problem will use at most twice the number of memory accesses as the solution to the static RL problem. (This penalty can be avoided by using a cacheline twice as wide.)

**Incremental classification:** We first consider the one-dimensional PC problem. In the static case, we could simply reduce this to the RL problem. The simple reduction of PC to RL is no longer sufficient in the incremental case since update times may be  $O(n)$ , which is prohibitive. The worst case update time arises if one inserts an interval which intersects most existing elementary intervals. Our solution is to use a variant of the FIS tree (*incremental FIS tree*) to reduce the update time.

Recall the definition of a FIS tree with indegree  $t$  and  $\ell$  levels on  $n$  segments. We define a FIS tree variant, called the *incremental FIS tree*, in which the internal nodes that the leaves connect to may have degree between  $t$  and  $ct$  for some suitable, small constant  $c$ ; all other internal nodes have in-degree  $t$  as in a standard FIS tree.

Say there is a collection of  $n$  rules at the beginning we start by building a FIS tree on the elementary intervals with one modification. We only store the least cost rule in each canonical set. The canonical sets themselves are not stored; we refer to the cost stored with a node  $\nu$  as its *canonical cost*  $C_\nu$ . Recall that the endpoints of the elementary intervals determined by the segments is called the  $x$ -set which is also maintained.

Suppose we insert a rule. This may generate new elementary intervals by splitting at most two existing elementary intervals. Splitting an elementary interval involves updating the

B-tree search structure on the  $x$ -set and the FIS tree. In the FIS tree we replace the corresponding leaf with two leaves of the same parent. Now it is straightforward to insert a rule into all appropriate canonical sets. Answering a lookup query with a point  $q$  is as before except that we solve the DRL problem on the  $x$ -set and return the minimum canonical cost of this leaf and that of any of its parents. As long as each insertion of a rule into the lowest level canonical set is atomic the lookups can proceed in parallel with the updating of the data structure. This implies that the time period during which the data structure needs to be locked is very small.

*Theorem VI.1:* Any  $O(n^{1/\ell})$  rules can be inserted such that an insert takes  $O(\ell n^{1/\ell})$  memory accesses in the worst case; a lookup query takes at most  $\text{DRL}_t(n, U) + \ell$  memory accesses. The space used is  $O(n)$ .

As an example, with  $\ell = 3$ ,  $O(n^{1/3})$  rules may be inserted; update time is  $O(n^{1/3})$  and the number of memory accesses is at most 3 more than that needed to solve the DRL problem.

We can extend this result to  $d$  dimensions by using incremental FIS trees instead of using standard FIS trees as described in Section V and by using an incremental FIS tree for the final dimension. For more details see the full version [6].

**Dynamic classification:** The data structures need to be more sophisticated in order to support deletions. It no longer suffices to maintain only the canonical cost for a node. If the corresponding rule is deleted, we need to quickly determine the rule with the smallest cost that overlaps the interval represented by that node. We explicitly maintain the entire canonical set at each node.

We define a FIS tree variant called the *dynamic FIS tree* to be a FIS tree with the following modifications. (1) The leaves connect to internal nodes that have indegree between  $t/c_1$  and  $c_2 t$  for suitable constants  $c_1, c_2 \geq 1$ . (2) Each node has pointers to a *base* canonical set and a *delta* canonical set; each canonical set is stored in a heap data structure. The delta canonical sets will be kept small, in particular, linear in the number of updates. The cost of the minimum cost rule from both two canonical sets of a node is its canonical cost, which we store at that node.

Say there is a collection of  $n$  one-dimensional rules at the beginning. We build a dynamic FIS tree on these segments by building a standard FIS tree and having the standard canonical set of a node be its base canonical set; the delta canonical sets of the nodes are empty. On inserting a rule we proceed to split elementary intervals as before. If a new interval  $v$  is created from interval  $u$  we initialize  $v$ 's base canonical set as the same as that of  $u$  and implement this as a pointer copy. We explicitly copy the delta canonical set of  $u$  into that of  $v$ . Again it is now straightforward to insert a rule into all appropriate canonical sets.

Deleting a rule works as follows. The rule is removed from the canonical sets of all nodes where it is stored, and the canonical costs are updated. We do not collapse the tree to remove any elementary intervals. Lookup on a query proceeds as before except that we find the minimum cost rule from both basic and delta canonical sets of each relevant node. That gives,

*Theorem VI.2:* Any  $O(n^{1/\ell})$  one-dimensional rules can be inserted or deleted such that each update takes  $O(\ell n^{1/\ell} \log n)$  memory accesses in the worst case; a lookup query takes at most  $\text{DRL}_t(n, U) + \ell$  memory accesses. The space used is  $O(n^{1+1/\ell})$ .

We can extend this to provide a dynamic  $d$ -dimensional classification just as before.

**Larger number of updates:** We have two suggestions to allow larger numbers of updates without sacrificing lookup performance significantly. The first is to relax the degree of the FIS tree at  $c$  levels. This involves splitting internal nodes of the FIS tree which can be done in a similar manner as the splitting of leaf nodes. The cost of lookup performance remains essential unchanged while the use of memory may increase by a factor of  $O(n^{c/\ell})$ . The second suggestion is to maintain the current data structure so that delta canonical sets are small. This involves copying FIS trees by collapsing basic and delta sets appropriately which may be done in the background since the number of lookup queries far outweighs the number of updates. If this copying is done carefully we can perform large number of updates while supporting lookups efficiently.

## VII. VARIOUS TRADEOFFS IN CLASSIFICATION

We have presented algorithmic solutions for the PC problem with provable performance guarantees. There are a number of ways to tailor them towards a particular application. We first summarize some of the issues with regards to the static version.

**Number of levels in the FIS trees.** The parameter  $\ell$  for a FIS tree in any of the dimensions must be chosen judiciously. The larger  $\ell$  is, the smaller the memory use and the larger the number of memory accesses will be. Appropriate choice of  $\ell$  will depend on the nested overlap structure of the intervals. If the overlap is large, a somewhat larger value of  $\ell$  is needed to decrease memory requirements.

**Choosing appropriate solutions for the subproblems.** The PC problem for  $d$  dimensions uses solutions to smaller dimensional PC problems on the canonical sets. Which solution to apply for the subproblems depends on the characteristics of the dataset. For example, if the canonical sets are moderate in size, the following *mapping approach* may prove efficient.

The *mapping approach* uses our solution after a reduction of the universe size. First we project all the endpoints of the  $x$ -projections of the rectangles to get at most  $2n$  elementary intervals; call this the  $x$ -set. We label the endpoints of these intervals using odd numbers, so the endpoints are now in the range  $1 \cdots 4n$ . We do likewise for the  $y$ -projections of all the rectangles and get the  $y$ -set. Now we can solve the PC problem with these labels on the rectangles which means that the solution to RL problems now involves only integers in  $1 \cdots 4n$ , and not in  $1 \cdots U$  as before. To perform a lookup query, we first solve the RL problem on the  $x$ -set with the given source IP address and assign it the odd number that falls in the interval it belongs; we do likewise for the destination IP address with the  $y$ -set. Following that, the query becomes a point in  $[1 \cdots 4n, 1 \cdots 4n]$  which can be solved using our methods and leads to the following theorem.

*Theorem VII.1:* Say the  $x$ -FIS tree has  $\ell$  levels. The space used by our data structure is  $O(\ell n^{1+1/\ell})$ . The number of memory accesses for each query is at most  $2 \text{RL}_t(n, U) + \ell \text{RL}_t(n, n)$ .

If  $n$  is much smaller than  $U$ , then  $\ell \text{RL}_t(n, n)$  may be smaller than  $(\ell - 1) \text{RL}_t(n, U)$ , and the mapping approach may outperform our basic approach.

**The order in which the dimensions must be considered.**

When there is more than one dimension involved, the order in which we consider them for building FIS trees may make a difference. For example, source and destination addresses may need to be considered prior to port numbers since there are likely to be only few port numbers for a particular source/destination combination, and hence the FIS trees for some dimensions may become trivial.

For the dynamic version, there are additional considerations. For example, the choice of the branching factor in the FIS tree at various levels governs the number of updates that can be performed without major maintenance of the tree. Also, in our approach, we can multiplex the updating of the tree with performing the lookups, although this requires careful implementation. Finally, one can batch updates and perform them more efficiently than doing each individual update separately. Which combination of these techniques to use in practice depends on the application, the nature of datasets, etc.

## VIII. EXPERIMENTAL STUDY

We focus only on the static version of the two-dimensional PC problem and present a thorough study of the tradeoffs for large rulesets. This special case offers significant insights into how to apply our algorithmic framework and the engineering issues involved.

### A. Datasets

Hardware restrictions and protocol availability limit the number of filters currently in use in IP backbone networks. Hence, finding good rulesets to test packet classification algorithms is difficult.

**Real Datasets:** Router vendors such as Cisco have augmented their traditional measurement capabilities to include usage based data which is needed to support traffic management and usage based billing. Cisco Netflow [13] measures statistics about flows at each enabled interface. A flow is a unidirectional sequence of packets between a given source and destination point that are close in time. Flow endpoints are identified by IP addresses, application port numbers, IP protocol type, type of service fields, and input/output interface identifier. The set of statistics that is collected on a per flow basis includes, but is not limited to, start time, end time, number of packets, number of bytes, bit mask used for IP lookup of the destination IP address, and bit mask used for IP lookup of the source IP address.

From the point of view of this paper, the most intriguing aspect is that this dataset includes not just the source and destination IP addresses but also the masks; therefore we know the network IP addresses<sup>4</sup> used in the lookup. In effect, if every packet had been classified by a (source, destination) rule, one possible ruleset would be given by the set of (source, destination) IP network pairs from the netflow data<sup>5</sup>. Our rulesets are obtained based on this premise.

We extracted 14 different rulesets from 7 flow datasets that were collected at 7 different routers within AT&T's WorldNet

<sup>4</sup>The IP network address is derived from the IP address  $a.b.c.d$  and the mask  $i$  by setting the  $i$ th most specific bits to 0, e.g.,  $a.b.c.d/24$  corresponds to network  $a.b.c.0/24$  and  $a.b.c.d/8$  corresponds to network  $a.0.0.0/8$ .

<sup>5</sup>If the packet was not netflow-switched but rather processor-switched, the mask bits are set to 0. Flows with masks bits 0 are eliminated from further analysis.

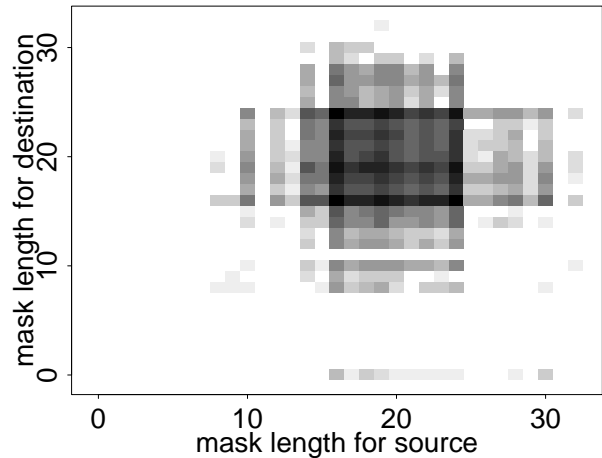


Fig. 4. Distribution of prefix length.

backbone on 21th of June, 1998 over a 24 hour period. The rulesets are derived from the netflow data by extracting the unique sets of (source, destination) IP networks, both from the whole ruleset and from a one hour subsample.

By design our rule sets do not contain any wild-card rules. We acknowledge that this is unrealistic and address this in a separate set of experiments where for a percentage of the rules we replace either the source or the destination network with a wildcard rule. In this way we decouple the evaluation of the influence of wildcards from the generation of the rulesets.

**Artificial ruleset:** We also constructed an artificial ruleset. We consider the forwarding tables from one of the routers. For each rule in the forwarding table, we randomly pick a source and a destination network from the networks that occur in the routing table. This ruleset is labeled ART.

**Ruleset characteristics:** Table IV summarizes some basic statistics of the rulesets. The first columns in the 24 hour and the one hour subcategories show the number of rules during the period. The second and third columns show the number of unique sources and unique destinations observed within the 24 hour trace. Since the routers have different numbers of interface cards and are at different locations in the network, the number of unique (source, destination) IP network pairs differs from ruleset to ruleset. The table reveals that the ART ruleset has more (source, destination) network IP addresses than the ruleset derived from the netflow data. We make two observations.

- The same IP networks are used in multiple rules — each source IP network is used on average between 19 and 35 times (36 to 45 times respectively) and the destination network is used on average between 11 and 14 times (64 to 97 times respectively) for the 1 hour (24 hour, respectively) rulesets.
- The distribution of bits used in the source and destination IP networks is shown in Figure 4. More precisely, the plot is an image map of the two dimensional histogram of the number of bits from the source and the destination networks from the ruleset  $R_1$ . A darker shading (log scale) indicates that more rules have the corresponding combination of bits in their network masks. As expected, the most common combinations include at least one class C network IP address. Somewhat surprisingly, network masks of 19 bits are rather popular. In the one hour rulesets, class B to class B routes are dominant.

## B. Algorithms tested

Our framework offers different ways of constructing algorithms for the two-dimensional PC problem based on (1) the different solutions for the RL (equivalently, one-dimensional PC) problem in one dimensions, and (2) the number of levels used in the FIS tree. Since we are mainly interested in exploring the memory vs. time tradeoffs from our two-dimensional framework, we *fix* the solution for the RL problem to be via multi-way search (this is also relevant in the dynamic case, but we do not explore that here.) With a cacheline of 32 bytes, we use multi-way search trees with a branching factor of 8. This enables us to solve the one-dimensional PC problem for all our rulesets with at most 6 memory accesses — at most 5 memory accesses for finding the correct elementary interval in a set of less than  $8^5 = 32768$  elementary intervals, and one additional memory access to identify the rule. We vary the number of levels in the FIS tree.

## C. Performance metrics

We are interested in two metrics: amount of memory used and the worst case number of memory accesses needed for a packet classification operation.

**Measuring the memory accesses:** The number of memory accesses is measured in terms of accesses to cachelines [15]. (A cacheline is assumed to have 32 bytes or 8 integers.) There are details in accounting for memory accesses. Say it takes  $a_1$  memory accesses in the worst case to solve the RL problem on the  $x$ -axis. For each level  $\ell$  of the  $\ell$ -level FIS tree, we compute the worst case number of memory accesses needed on the  $y$ -axis; let this be  $a_{2,\ell}$ . Then the total number of memory access is  $a_1 + \sum_{i=1..,\ell} a_{2,i} + \ell$ . The last  $\ell$  memory accesses are needed to find the minimum size rule among the  $\ell$  candidate rules identified by our algorithm.

**Measuring the memory usage:** We measure memory usage in two ways: absolute usage of memory in Mbytes and amount of memory used relative to the amount of memory used to store the input rules. (We assume it takes three integers or 12 bytes to store a rule.) In particular, the *memory factor*  $m_f$  is the ratio of the total amount of memory used to that needed to store the rules. A memory factor of 3 implies that the data structures for packet classification use three times the memory that is needed to store the rules.

## D. Experimental observations

**Basic performance bounds:** The results for the 8 rulesets are summarized in Table IV. Even rulesets with more than  $10^6$  number of rules can be searched with less than 22 (17, resp.) memory accesses using three (two, resp) levels of hierarchy; the memory factor is at most 4.1 (7, resp.). For rulesets of size 10,000 to 200,000, the number of memory accesses is sometimes fewer, and the memory requirements are below 12 (14) Mbytes using two (three) levels of hierarchy, respectively. Even though the rulesets are drawn from different physical locations (e.g., east coast, west coast, mid west) and are of widely varying sizes, the performance characteristics of the algorithm does not differ significantly. For the artificially derived ruleset, the performance is even better since the construction is less likely to create nested

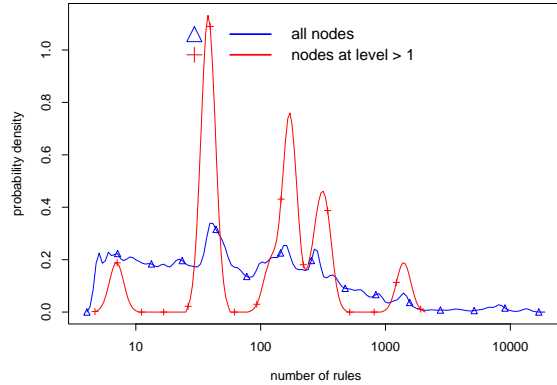


Fig. 5. Probability distribution of number of  $y$ -elementary intervals for each node in the  $x$ -hierarchy (adjusted for log scale).

rules. With just one level of the hierarchy, we observed that the memory factor is less than 8.5 for all rulesets, and at most 5 for 11 out of 16 rulesets.

**Effect of elementary intervals:** The memory used by our algorithm is dependent on the number of elementary intervals, more specifically, on the number of elementary intervals on the  $y$ -axis associated with the canonical sets of the nodes in the FIS tree. Figure 5 shows the distribution of number of elementary intervals for each node in the FIS tree for ruleset  $R_1$ . Note that the mean number of elementary intervals per node in the FIS tree is only 292 (90 percentile is only 407) and that the mean number of elementary intervals for the internal nodes of the FIS tree is even lower at 210. Indeed many internal nodes of the FIS tree have no elementary interval and therefore no rules associated with them. This implies that the memory requirements are significantly lower than our theoretical upper bounds from Section IV. Also consequently, the data structures associated with each of the FIS nodes can be searched much more efficiently than our upper bounds from Section IV indicate. This explains the gap between the theoretical number of memory access vs. the actual number of memory accesses for the ruleset. It also indicates that the average number of lookups is better than the worst case.

**Effect of wildcards in the ruleset:** The memory used by our algorithm depends on the extent to which the hierarchy of the FIS tree matches the inherent structure of the ruleset. Rules that span a large number of elementary intervals such as rules with wildcards ( $*$ 's match the entire domain of a field) may be associated with many nodes in the FIS tree, and result in memory wastage. This is indeed the case for a single level of hierarchy but as the number of hierarchy levels is increased, the negative influence of wildcard rules diminishes. Since none of our rulesets includes wildcard rules, we illustrate this point by starting with a ruleset ( $R_6$  being our example) and randomly replacing a percentage of the sources and destinations with wildcards. The results are shown in Table V. While the memory requirements increase significantly with just 2 levels of hierarchy, the increase is almost negligible for a 3 level hierarchy. The number of memory accesses increases from 14 to 19. However, this increase is an artifact of our prototype implementation: we have fixed the number of children at each node to be slightly larger than is required, for ease. If values were properly rounded and the boundary conditions were applied carefully, as it would be in

Rule-set	24 hour trace										1 hour trace							
	# unique			2 levels			3 levels				2 levels			3 levels				
	rules 10 <sup>6</sup>	src 10 <sup>3</sup>	dst 10 <sup>3</sup>	MB	$m_f$	$t_m$	$I_{elem}$ 10 <sup>6</sup>	MB	$m_f$	$t_m$	rules 10 <sup>3</sup>	MB	$m_f$	$t_m$	$I_{elem}$ 10 <sup>6</sup>	MB	$m_f$	$t_m$
$R_1$	1.09	13	26	65.5	4.0	16	4.36	55.5	3.2	21	150	9.4	4.2	16	0.67	8.5	3.7	17
$R_2$	0.46	6	12	28.0	4.1	15	1.84	24.3	3.5	18	67	4.2	4.2	14	0.31	3.9	3.8	15
$R_3$	1.15	18	25	85.3	5.2	16	4.94	63.0	3.6	18	149	10.3	4.7	14	0.64	8.1	3.5	16
$R_4$	0.54	8	13	30.1	3.6	16	1.86	25.2	2.9	20	78	4.8	4.1	13	0.30	3.9	3.1	16
$R_5$	1.31	13	29	88.8	4.6	17	5.70	72.3	3.6	22	212	13.7	4.4	16	0.92	11.6	3.6	21
$R_6$	0.20	3	5	9.3	2.9	15	0.62	8.7	2.6	18	34	1.5	2.8	13	0.11	1.5	2.7	14
$R_7$	1.18	16	26	101.2	6.2	16	5.86	71.7	4.1	18	135	12.8	6.9	16	0.72	8.7	4.3	17
ART	1.00	32	32	43.9	2.7	14	2.83	41.1	2.4	15	100	3.9	4.2	11	0.30	3.7	4.0	13

TABLE IV

PERFORMANCE FOR RULESETS  $I_{elem}$  is the number of elementary intervals.  $MB$  is total memory used, in MBytes.

% wild-cards	2 levels of hierarchy				3 levels of hierarchy			
	$I_{elem}$	MB	$m_f$	$t_m$	$I_{elem}$	MB	$m_f$	$t_m$
20	292K	3.1	6.8	14	150K	1.8	3.6	19
10	234K	2.7	5.6	14	147K	1.8	3.6	19
5	165K	2.1	4.1	14	119K	1.6	3.0	19
0	114K	1.5	2.8	13	107K	1.5	2.7	14

TABLE V

PERFORMANCE WITH WILDCARD RULES. Ruleset is  $R_6$  by randomly replacing src and dst with % of wildcards.

# levels	Ruleset $R_1$					ART		
	$I_{elem}$	MB	$m_f$	$t_m$		MB	$m_f$	$t_m$
1	6.17M	72.1	4.5	11		45.7	2.8	10
2	5.43M	65.5	4.0	16		43.9	2.7	14
4	3.68M	49.0	2.7	23		39.3	2.3	19
6	3.26M	45.2	2.4	28		38.2	2.2	24
10	2.83M	41.0	2.1	41		37.0	2.1	37
18	2.73M	39.9	2.0	66		37.0	2.1	55

TABLE VII

PERFORMANCE AS THE NUMBER OF LEVELS IS INCREASED.

# rules in ruleset	2 levels of hierarchy			3 levels of hierarchy		
	MB	$m_f$	$t_m$	MB	$m_f$	$t_m$
1093K	65.5	4.0	16	55.5	3.23	21
273K	17.6	4.4	15	15.5	3.73	20
68K	4.1	3.9	14	3.9	3.74	15
17K	1.1	4.3	12	1.0	3.86	14
4K	0.3	4.3	9	0.3	4.02	13
1K	0.1	4.5	8	0.1	4.75	10

TABLE VI

PERFORMANCE WITH INCREASING RULESET SIZE.

a production quality solution, the additional memory accesses can be eliminated. With three levels of hierarchy, the memory needed to support the 33,604 rules is still less than 2 Mbytes.

**Scaling the number of rules:** An important aspect is to understand how the performance scales as we increase the number of rules in the ruleset. See Table VI for the performance for rulesets of size between roughly  $10^3$  rules to  $10^6$  rules. We derived each ruleset by starting with a ruleset ( $R_1$  at the beginning) and retaining roughly one quarter of it by selecting each rule with probability  $1/4$ . From Table VI we can see that the memory factor stays between 3.9 and 4.5 for two levels of hierarchy, and between 4.8 and 3.2 for 3 levels of hierarchy. If the ruleset is too small, the overhead of an extra level of hierarchy can increase the memory requirements as seen with rulesets of less than 4K rules. With reasonable ruleset sizes, the memory factor does not seem to depend on the number of rules, and it seems to stay more or less constant. As expected, the number of memory accesses increases as the data sets grow because the number of distinct sources and destinations increases.

**Effect of the number of levels:** Our solution offers a memory vs. access time tradeoff depending on the number of levels in the FIS tree. Table VII shows the tradeoffs for two rulesets, flow

ruleset  $R_1$  and synthetic ruleset ART. On the total memory size, the impact of increasing the levels of the FIS tree beyond four or five is minimal.

**Summary:** Our exploration of the various tradeoffs lets us conclude that for small rulesets (up to a few K rules), one level FIS tree suffices. The space used is a few 100KBytes and the number of memory accesses is less than 10. For moderate sized rulesets (up to a few 10K rules), two level FIS trees suffice; space used is a few MBytes and the number of memory accesses is about 15. For very large data sets (order of  $10^6$  rules), two or three level FIS trees suffice; space used is up to 100 MBytes and the number of memory accesses below 18. Taking a very simplistic approach that disregards pipeline stalls and the complications of random access, one could expect that memory access speed of 8ns (SRAM) would translate into lookup times that are in the ballpark of OC48. A memory speed of 28ns, e.g., DRAMs, would translate to roughly OC12 and 110ns would translate to OC3.

As a side issue, the performance of our solution is significantly better for the artificially constructed ruleset ART, than for real data sets. This should not come as a surprise since the ART ruleset cannot capture the correlations in actual rulesets we obtained from various routers (as described earlier) illustrating some pitfalls of constructing random or arbitrary rulesets.

## IX. RELATED WORK

The PC problem has been well studied. Primarily the motivation has been to explore if software based solutions can perform lookups at high linespeed<sup>6</sup>. There are many approaches known,

<sup>6</sup>Hardware based approaches have also been explored, e.g., using Content Addressable Memories (CAMs), e.g., [10], high speed caches [8] etc. Our approach is akin in spirit to the software solutions, and hence we do not survey hardware based solutions to the PC problem. However, our solutions themselves can be

and we have already compared our results with the best known ones in Section III.

The work closest to ours is in [11]; their basic approach can be thought of using a one level FIS trees. As such, the memory requirement will be high. However, the authors used sophisticated compression techniques to decrease the space. Here, we use 2 or 3 level FIS trees which makes a substantial difference as our experiments indicate. There are solutions that use  $O(n \log^{d-1} n)$  space and  $O(\log^{d-1} n)$  memory accesses per query [3], [7]. Although these are space efficient, the lookups are rather slow. Our solution is based on the result in [7] that has  $O(\log \log n)$  memory accesses, but uses moderately large space. We have already described how our data structure extends the work in [7].

Recently, independent of our work, a dynamic algorithm was proposed for the PC problem [2]. This algorithm takes  $O(\alpha w)$  memory accesses for query processing and  $O(\alpha n^{1/\alpha})$  time for updates, for the two dimensional case; here  $\alpha$  is a tuneable parameter. Our algorithms were built on the premise that query times must not be sacrificed to accommodate updates, and furthermore, that updates are less frequent than queries. Hence, our algorithms were designed to support query processing in  $O(\log w)$  memory accesses. Our solutions do use more space than [2] in the worst case, but just as in the static case shown here experimentally, we expect the structure in the datasets to result in reasonable space usage.

In the dynamic case when rules are inserted, a query may have a *conflict*, that is, there are two (or more) least cost rectangles that contain it, but neither of these rectangles is contained in the other. Our solution will be able to detect the presence of conflicts when a query is presented. (In contrast, authors in [1] show how to detect all possible conflicts when a rule is inserted.) Solutions in [1] can be used to resolve these conflicts as we detect them.

## X. CONCLUSION

We have provided an algorithmic framework for solving the PC problem using simple, yet powerful ideas such as (1) using a “fat” hierarchy of canonical sets to decrease the number of sets to be searched per query [7], (2) locating the canonical sets to be searched by proceeding up from the leaves using the inverted edges of the FIS tree (Section III), (3) locating the leaves in FIS trees using the standard IPL problem, thereby leveraging off best known hardware and software solutions for it (Theorem III.1), (4) using FIS tree nodes with flexible degree to allow moderate number of updates without degrading the lookup performance significantly (Section VI, and (5) reducing the universe size using the IPL problem before applying our solutions thereby reducing the memory accesses for each consequent IPL solution. In our framework, these ideas can be combined to give space-time tradeoffs for PC. In particular, our experimental results based on rulesets derived from AT&T WorldNet traces indicate that using less than 80 Mbytes of 8ns SRAM memory one can perform static two-dimensional PC at roughly OC48 rates<sup>7</sup> (2.4Gbit/second) for rulesets with more than 1 million rules even

if all packets are only 40 bytes in size. This will be quite satisfactory in practice.

Our entire framework is based on decreasing the query lookup time aggressively; theoretically, this leads to the use of moderately large space. However, our extensive experimental study of traces from various WorldNet routers shows that in practice, the space usage is very reasonable, e.g., within a factor of 5 of the space used to store the rulesets themselves.

What remains to be explored is to apply the insights from our two-dimensional study to engineer the framework for specific applications. In particular, the dynamic case and the multidimensional case are of interest. A concern is the lack of large, realistic datasets for such cases.

## REFERENCES

- [1] H. Adishesu, S. Suri, and G. Parulkar. Packet filter management for layer 4 switching, 1999.
- [2] M. Buddhikot, S. Suri, and M. Waldvogel. Space decomposition techniques for fast layer-4 switching. In *Proceedings of the IFIP Sixth International Workshop on Protocols for High Speed Networks*, 1999.
- [3] B. Chazelle. Filtering search: a new approach to query-answering. *SIAM Journal Computing*, 15:703–724, 1986.
- [4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Theory and Applications*. Springer-Verlag, Heidelberg, 1997.
- [5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *ACM Computer Communication Review*, 27(4):3–15, 1997. ACM SIGCOMM’97, Sept. 1997.
- [6] A. Feldmann and S. Muthukrishnan. Tradeoffs for packet classification, 1999. AT&T Technical Report.
- [7] P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching. In *IEEE ACM Symp on Theory of Computing*, 1999.
- [8] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proc. IEEE INFOCOM*, page 1241, San Francisco, California, 1998.
- [9] P. Gupta and N. McKeown. Packet classification on multiple fields. *ACM Computer Communication Review*, 1999. to appear ACM SIGCOMM’99, Sept. 1999.
- [10] Quality Semiconductor Inc. Content addressible memory. <http://www.qualitysemi.com/news/qcam.html>.
- [11] T.V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *ACM Computer Communication Review*, 28(4):203–214, 1998. ACM SIGCOMM’98, Sept. 1998.
- [12] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. In *Proc. IEEE INFOCOM*, page 1248, San Francisco, California, 1998.
- [13] Cisco Netflow, 1999. [http://www.cisco.com/warp/public/cc/cisco/mkt/ios/netflow/tech/napps\\_wp.html](http://www.cisco.com/warp/public/cc/cisco/mkt/ios/netflow/tech/napps_wp.html).
- [14] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. *ACM Computer Communication Review*, 1999. ACM SIGCOMM’99, Sept. 1999.
- [15] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *ACM Computer Communication Review*, 28(4):191–202, 1998. ACM SIGCOMM’98, Sept. 1998.
- [16] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
- [17] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. *ACM Computer Communication Review*, 27(4):25–36, 1997. ACM SIGCOMM’97, Sept. 1997.
- [18] D.E. Willard. Log-logarithmic worst-case range queries are possible in space  $\Theta(N)$ . *Information Processing Letters*, 17(2):81–84, 1983.

implemented in hardware using off-the-shelf solutions for the IPL problem as remarked earlier.

<sup>7</sup>This is a ball park figure that disregards pipeline stalls.