SPACE DECOMPOSITION TECHNIQUES FOR FAST LAYER-4 SWITCHING

Milind M. Buddhikot,
Lucent Bell Labs
Subhash Suri,
Washington University in St. Louis
Marcel Waldvogel,
ETH, Zürich
milind@dnrc.bell-labs.com, suri@cs.wustl.edu, mwa@tik.ee.ethz.ch

ABSTRACT

Packet classification is the problem of matching each incoming packet at a router against a database of filters, which specify forwarding rules for the packets. The filters are a powerful and uniform way to implement new network services such as firewalls, Network Address Translation (NAT), Virtual Private Networks (VPN), and per-flow or class-based Quality of Service (QOS) guarantees [4]. While several schemes have been proposed recently that can perform packet classification at high speeds, none of them achieves fast worst-case time for adding or deleting filters from the database [3, 8, 9]. In this paper, we present a new scheme, based on space decomposition, whose search time is comparable to the best existing schemes, but which also offers fast worst-case filter update time. The three key ideas in this algorithm are as follows: (1) innovative data-structure based on quadtrees for a hierarchical representation of the recursively decomposed search space, (2) fractional cascading and precomputation to improve packet classification time, and (3) prefix partitioning to improve update time. Depending on the actual requirements of the system this algorithm is deployed in, a single parameter α can be used to tradeoff search time for update time. Also, this algorithm is amenable to fast software and hardware implementation.

1. INTRODUCTION

In recent years the Internet has transformed from a early day low speed network connecting predominantly educational institutions to a gargantuan fast growing commercial infrastructure. The diverse users of the Internet now range from ordinary home users downloading recipes to large corporations conducting sensitive transactions over the net. The expectations in terms of security, privacy, performance, and reliability of these diverse users are dramatically different. Realizing this, Internet Service Providers (ISP) are envisioning new differentiated network services that can meet demands of the full spectrum of clients. For example, a corporation that has multiple sites may want to connect its internal networks using the Internet but request strict bandwidth and delay guarantees and require that all its packets be encrypted as they flow through the Internet. To provide this new network service, commonly termed as *Virtual Private Networks* (VPN), the routers must be able to recognize packets originating from or destined to corporation sites and process them differently than other packets. However, IP routers that provide the best-effort Internet service of today differentiate packets based only on the IP destination address, the minimum requirement to get the packet closer and closer to its destination. To realize a service such as VPN requires the router to look at additional network layer information such as the source address, protocol type, and transport protocol fields such as source, destination ports [8, 9]. This new paradigm for packet forwarding based on network (ISO/OSI Layer 3) and transport (Layer 4) level information is termed as *Layer 4 Forwarding* or *Layer 4 Switching* and is central to realization of new differentiated network services such as firewalls, Network Address Translation, Virtual Private Networks, and per-flow or class-based Quality of Service (QOS) guarantees. A router supporting Layer 4 Switching maintains a table of rules for classifying packets, commonly called *filters*. Each rule also has an *action* item associated with it. Two important aspects of Layer 4 packet classification are: (1) *filter search* — classify/match every incoming packet to a lowest cost/highest priority filter and performs the associated action on the packet. (2) *filter update* — update filter table in the event of a filter addition or deletion. To be ready for the growing demands of users and ISPs, a Layer 4 router must be perform the filter matching operation at Gigabit per second rate. However, it is becoming increasingly evident that in addition to this, services such as firewalls and NAT require the router to support insertion and deletion of filters with sub-second latency. Unfortunately, recent packet classification algorithms reported in the literature only support fast filter search and require prohibitively large update time that grows at least linearly with the number of filters in the database or even require a complete rebuild of the lookup structure [8, 9, 3]. In addition, for a filter database with N entries, some of these algorithms [3, 9] require $O(N^2)$ space which is prohibitively high.

Contributions

In this paper, we describe a class of algorithms called (PACARS) — **PA**cket **C**lassification **A**lgorithms using **R**ecursive **S**pace-decompositions and present in detail a specific instance called Area-based Quad Tree (AQT). We focus primarily on 2-dimensional prefix based filters. However, our scheme can be extended to multi-dimensional filters using well known techniques in [8].

For N two-dimensional filters, our scheme requires $O(N)$ space, $O(\alpha W)$ worst-case search time, and $O(\alpha \sqrt[\alpha]{N})$ worst-case update (insert/delete) time. Using α as a tunable parameter, we can tradeoff lookup time for faster update time and thus, tune our algorithm to the requirements of dominant services. For example, with $\alpha = 2$, we get a search time of $O(2W)$ and update time of $O(2\sqrt{N})$, which are suitable for applications that require fast searches and reasonably fast updates. With $\alpha = 3$, the search time is increased to $3W$ but the update time is reduced to $O(3\sqrt[3]{N})$.

2. RELATED WORK

The problem of Layer 4 packet classification has received significant attention in the recent past. Existing commercial implementations of firewalls that use layer-3/4 filters often use linear search and hence do not scale to large databases. Caching based approaches are not scalable, since each cache miss requires a linear search of the database, which can be a big bottleneck.

Among other recently proposed schemes, Stiliadis et al. [9] present two algorithms: their first scheme is hardware oriented and requires wide data buses. It can handle general K-dimensional filters, but requires $O(N^2)$ space and expensive hardware. Their second algorithm is a 2D scheme that is more appropriate for software implementation but it does not handle general filters. Also, the worst case update time of both the schemes is $O(N)$.

In [8], Srinivasan et al. present a fast 2D scheme, called *Grid-of-Tries*, with $O(N)$ space requirement and attractive worst case search time of $O(W)$. By maintaining four such grid-of-tries, they can handle 5-dimensional filters. The worst-case update time of this scheme is also $O(N)$ and requires complicated *lazy update* schemes to improve average case performance.

Decasper et al. [3] present a packet classification scheme based on finite state machines. This scheme, though fast for lookups, requires $O(N^2)$ memory and is thus completely impractical for the number of filters that are expected in the future.

A more recent scheme called Tuple Space Search [6] proposed by Srinivasan-Suri-Varghese can handle arbitrary general filters, and has fast update time, but its worst-case bounds on both the search and update time are very poor.

In recent years, a new form of Content-Addressable Memories (CAM) called ternary CAMs have been proposed for use in packet classification and routing. However, they suffer from high cost, large power dissipation, and $O(N)$ worst case update time.

3. PACKET CLASSIFICATION, SPATIAL DECOMPOSITION, AND QUADTREES

This section presents the basics of the packet classification problem, and how to approach it using recursive search space decomposition, and the quadtree data structure.

Overview of Packet Classification Problem

Before we discuss our algorithms, we briefly review the multidimensional packet classification problem [8]. We assume that the router maintains a filter database or table that consists of N filters $F_1, F_2, \ldots, F_N$, each with K fields corresponding to the packet headers which it should match. In case of IPv4 packets, fields such as IP source address (SA, 32 bits), IP destination address (DA, 32 bits), protocol identification number (PID, 8 bits), Type-of-Service (TOS, 8 bits), and transport protocol level source/destination port (SP, DP, 16 bits each) have been considered as relevant fields. Each of the header fields is assigned one of the four match types: *exact match*, *wildcard match*, *prefix match*, and *range match*. For an exact match, the field in the header must completely match the specified filter field. Wildcard matches allow the database to contain either a fully specified field or a match-all (wildcard) symbol. In a prefix match, the packet's field must match the first *prefix length* bits of the filter's field, where the prefix length is also specified in the filter. In a range match, the value of field in the packet header must fall in the range specified in the filter. Each filter has associated *action* that is taken when the packet matches it. Consider an example of a 5-tuple firewall filter (SA, DA, PID, SP, DP) = $(1110*, 101*, \text{TCP}, [1110\ldots6000], [2000\cdots4000])$ with associated action *Allow the packet*. A packet $(1110\ldots, 10111\cdots$, $\text{TCP}, 2000, 3000)$ matches this filter and will be allowed to pass through the router but $(1110\ldots, 10111\ldots$, $\text{TCP}, 1009, 3000)$ does not match the filter and hence will be dropped, unless it matches another filter.

Our algorithm allows for matching against a database of prefix pairs and range pairs, respectively. It can be augmented in a straightforward way to also match against a small number of fields with wildcard matches and a limited number of ranges. Due to space limitations, in the remaining discussion, we exclusively focus on 2D prefix based filters. We will first describe the basic ideas in our algorithm, namely the geometric interpretation of filters and hierarchical quadtree based representation of decomposed space.

Space Decomposition and Quadtrees

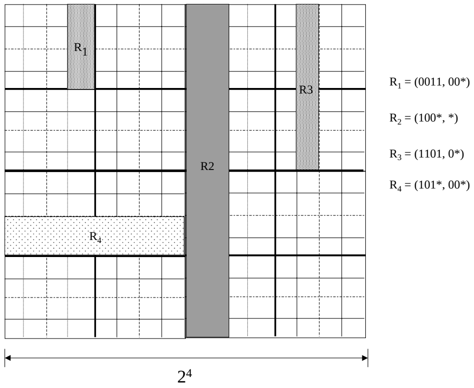


Figure 1 Geometric interpretation of filters: An example

The geometric interpretation of 2D filters forms the foundation of our scheme. If W is maximum prefix length, a prefix filter can be viewed as a rectangle in the $2^W \times 2^W$ search space. For example, a filter $F = (S*, D*)$, where S is a i bit prefix and D is a j bit prefix, can be represented by a $2^{W-i} \times 2^{W-j}$ rectangle. Figure 1 illustrates this using an example of four filters R_1, R_2, R_3, R_4 with a maximum prefix length $W = 4$. Here the filter $R_2 = (100*, *)$, represents a rectangle of size $2^1 \times 2^4$ in the search space of size $2^4 \times 2^4$. An incoming packet with a fully specified source and destination address, defines a point in the space. In the rest of the paper, we will use the terms filter and rectangle as well as point and packet interchangeably. Note that in a geometric representation of a general filter database, rectangles (filters) can potentially overlap and the point (packet) can thus belong to multiple rectangles (filters).

In the fields of image processing, computer graphics, and remote sensing such 2-dimensional point and region data is commonly represented using quadtrees. A quadtree is a representation of a recursive partitioning of an address space where regions are split until there is a constant amount of information to be stored in them. Several variants of the basic quadtree that differ in the type of data they store and the semantics of tree construction and search have been reported in literature [5]. A basic quadtree is a *4-way branching tree* that represents a recursive binary decomposition of space wherein at each level we divide a square subspace into four equal size squares – the north-east (NE), north-west (NW), south-east (SE), and south-west (SW) quadrants. Each node v in the tree corresponds to a square in the decomposition and its four children correspond to the four sub-squares obtained by dividing the square of v. Figure 2 illustrates this decomposition scheme.

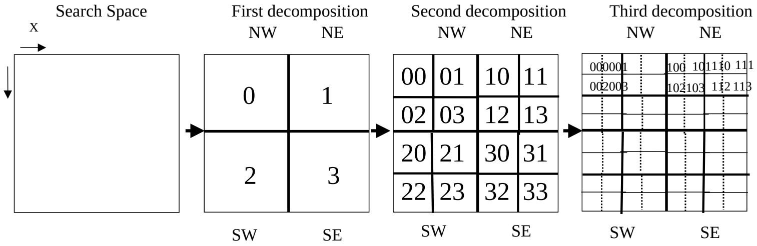


Figure 2 Recursive spatial decomposition

In our basic packet classification scheme, the decomposition is induced by a set of filters. We continue to divide a square recursively using binary decomposition until all packets mapped to that square are classified by the same filter and no more decomposition is required. Since the corresponding quadtree represents search space, every fixed point in the space has fixed location in this tree. Therefore, given a point's coordinate, this data structure will help us answer questions such as does it belong to a specific area in the space. Specifically, starting at the root of the tree, we can use successive bits of the X and Y co-ordinates of the point to make branching decisions at the nodes along the search path terminating at a leaf node and use the information therein to answer the query. The runtime and number of memory accesses for this search are proportional to the height h of the quadtree.

However, this naive scheme has the memory explosion problem — N filters in the worst case can lead to N^2 space. In the next section we present a scheme that requires $O(N)$ space.

4. AREA-BASED QUAD TREES (AQT)

In this section, we will first describe the key insight that leads to the concept of a *crossing filter set* (CFS) and then describe the quadtree data structure called Area-based Quad Tree (AQT) based on this insight.

Crossing Filter Set (CFS)

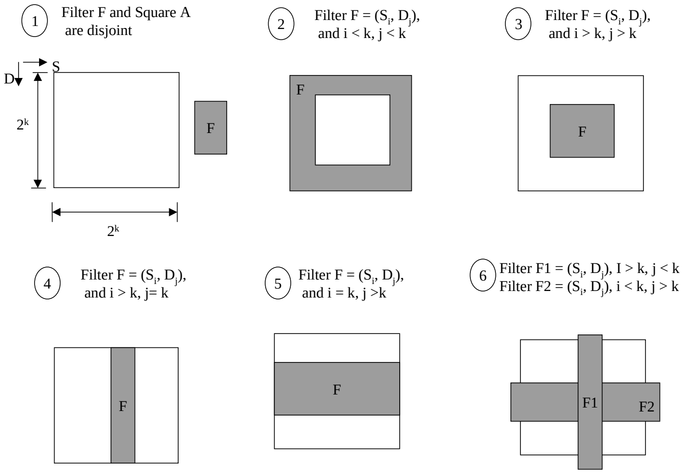


Figure 3 Filter overlaps

Our key insight is as follows (Refer Figure 3): consider a square A of size $2^k \times 2^k$ in the quadtree subdivision, and a filter $F = (S, D)$, where S is i bits long, and D is j bits long. There are several ways in which A and F can be inter-related: (a) If F and A are disjoint, then F is irrelevant to square A (case 1). (b) If the rectangle F completely contains A, then we don't need to pass F down to nodes that correspond to smaller squares. Of all the filters whose rectangles completely contain A, it suffices to just keep track of the lowest cost filter. This happens if both i and j are smaller than k (case 2). (c) If F lies entirely inside A, then of course, we continue to subdivide A and pass F down. This happens if both i and j are greater than k (case 3). (d) Last, the most interesting case is when F falls in none of the cases considered so far. In this case, F intersects square A, but neither contains the other completely. Because our filters are prefix filters, and since each square in the quadtree decomposition has size $2^l \times 2^l$ for some l, it follows easily that the rectangle F can intersect A in only one way — crossing A completely in one dimension. Figure 3 (4, 5, 6) shows various cases of overlaps to illustrate this point. Clearly, if $i \geq k$ and $j \leq k$, then F crosses A in the D dimension (case 4, 6 (F1)). Cases 5 and 6(F2) are complimentary cases in S dimension. In all these cases, we say that F_i *crosses* A. We call the set of all filters that cross a given region A as its *Crossing Filter Set* (CFS). We use this basic idea to construct a quadtree representation of the search space, called Packet Classification using Recursive Space-decomposition, (PACARS), as follows (Figure 4): Given a filter set FD, start with the root of the tree that corresponds to the entire search space.

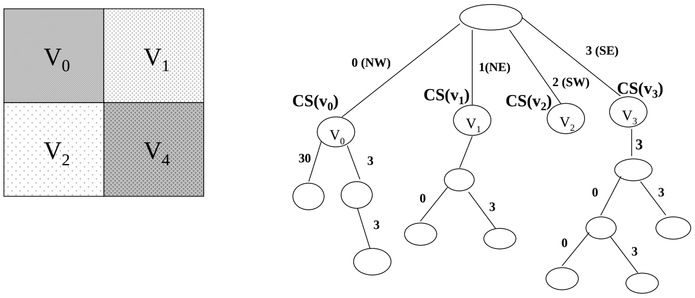


Figure 4 Basic PACARS Data Structure

Compute the CFS of the root and store it in a Crossing Filter Set Data Structure (CFSDS). Remove these filters from FD and recursively divide the search space into four children v_1, v_2, v_3, v_4. Compute the set $F(v_i)$ of filters that are completely contained in the space associated with v_i and then repeat the process of computing CFS at v_i. Do this recursively at each v_i and its children until the node (region) under consideration has only one or zero filters left. The method of space decomposition decides the height h of the quadtree and type of the PACARS algorithm. Now, our basic algorithm in the form of pseudo-code is as follows:

Algorithm 4.1 *Constructing the* PACARS *quadtree*

```
 1  r = InitQuadTreeRoot()
 2  A(r) = Square area ( $2^W \times 2^W$) associated with r
 3  F(r) = FD;  //The set of input filters
 4  C(r) = Set of filters in F(r) that cross A(r)
 5  BuildCFSDS($C(r), r$));  //Build a crossing filter data structure on C(r)
 6  $F(r) = F(r) - C(r)$;  //Remove C(r) filters from current set
 7  $v = r$;
 8  Divide A(v) into four children sub-squares: $A(v_1), A(v_2), A(v_3), A(v_4)$;
 9  $F(v_i)$ = Filters in F(v) that lie entirely in $A(v_i)$
10  for $i = 1$ to $4$ do
11      BuildCFSDS($C(v_i), v_i$));
12      $F(v_i) = F(v_i) - C(v_i)$;
13      if $(F(v_i) == \phi)$
14        then continue ;  // F($v_i$) is empty, do nothing
15            else //F($v_i$) has $>$ 1 filter, recursively compute decomposition for $v_i$
16                  Recurse from 8
17        fi
18      od
```

The main novel idea here is the use of crossing filter sets. This idea ensures that the memory requirement is $O(N)$, because each filter F is stored exactly once, at the highest node for which F is a crossing filter. Now, we will describe the crossing filter data structure (CFSDS), and how the query algorithm works.

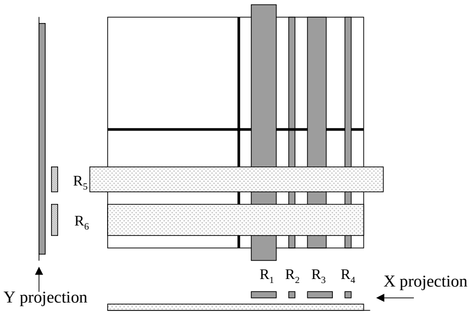


Figure 5 Crossing filter lookup

Consider what a CFS set C(v) must look like (Figure 5). The filters of C(v) can be divided into two groups, CX(v) and CY(v). The former is the set of filters that cross the square A(v) perpendicular to the X axis; and the set Y(v) is the set of filters that cross A(v) perpendicular to the Y axis. In our example, R_5, R_6 belong to CY(v), whereas R_1, R_2, R_3, R_4 belong to CX(v). We can exploit this special structure of the CFS to efficiently find the filter match at each node. Observe that for each CFS, we can project the component filters along X and Y axis, and since, the filters are specified using prefixes, these projections are also prefixes. This therefore reduces the problem of filter match to problem of finding the best matching prefix (BMP) along X and Y axis and selecting the one corresponding to the high priority (lowest cost) filter. The problem of finding best-matching-prefix has been widely researched. We look at three possible ways to solve this problem at each CFS.

Store the prefixes in a binary trie: This approach reported in [7] can find a BMP in $O(W)$ time with very small constant, where W is the maximum prefix length.

Binary search based on prefix length: In this approach reported in [10], a modified binary search is performed among prefixes sorted using the length of the prefixes. This scheme finds a BMP in $O(\log W)$ time.

Binary search on prefix endpoints: Note that each prefix $X*$ covers a range of numbers $[(X0 \cdots 0) \dots (X1 \cdots 1)]$. Therefore, we can store m prefixes as $2m$ numbers or keys. With each key we store, two prefix ids

— *equal* and *less-than*, which are used to decide the matching prefix if the point/packet under search is equal to or less-than than the key under consideration. This formulation reduces the BMP problem to finding the successor element which is the smallest entry greater than the search value. If the key found exactly matches the key under consideration, the prefix ID stored in the *equal* field defines the most-specific or the best matching prefix. On the other hand, if the successor key is greater than the key under consideration, the *less-than* field defines the matching prefix ID. We can use simple binary search to obtain the matching or successor key and thus, solve the BMP problem in $O(\log N)$.

Now that we have all the parts, we summarize the search procedure: Given an incoming packet $P = (S, D)$, we form a location code L_p by interleaving S and D bit strings. The search begins at the root of the quadtree. We initialize a variable – $match$ to remember the least-cost filter along the search path. Starting at the most significant bit (MSB), we use the successive 2-bit values of L_p to make the branching decisions at the nodes that the search visits. At each node, the we search the CFS structure for the best matching (least-cost or highest priority) filter $bestf$ at that node. If the filter matches 'better' than the filter recorded in $match$, we replace value of $match$ with $bestf$ and continue. If we exhaust the bits or reach a quadtree leaf node, indicating end of the search path, the search is complete and $match$ represents the filter match. This suggests that in this naïve formulation we can solve filter matching problem in $O(h \log W)$ or $O(h \log N)$.

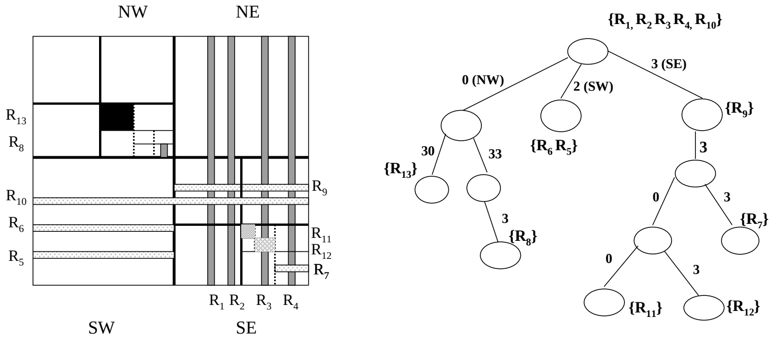


Figure 6 Area-based quadtree (AQT)

In the following, we will combine these ideas, namely binary space decomposition and crossing filter sets (CFS), with a simple way to form CFS sets and formulate our complete scheme called Area-based Quad Tree (AQT). Specifically, the root node of an AQT has an area of $2^B \times 2^B$ associated with it, where as each of its 4 children has squares of area $2^{B-1} \times 2^{B-1}$ associated with them. In general, a node at level i has a square with an area of $2^{B-i} \times 2^{B-i}$ associated with it (the root node is at level 0). We store a rectangle R at a quadtree node, if the square associated with the node is the *smallest* square that fully contains R. We can see that every filter passed down to the node at level i has at least one prefix of length i bits. This observation leads to the following rule for placing filters in quadtree nodes: A rectangle R represented as $(X_{w_1}*, Y_{w_2}*)$, where w_1 is the length of the X prefix and w_2 is the length of the Y prefix, should be placed at a node at level $i = min(w_1, w_2)$. The square $SQ = (X_i*, Y_i*)$ associated with this node, where X_i (Y_i) is a prefix of X_{w_1} (Y_{w_2}), represents the *smallest* square that fully contains the rectangle R. If W is the maximum prefix length, we can see that the worst-case height of the area-based quadtree is W. Since, every filter is stored at only one node, the space complexity of this quadtree is $O(N)$.

Figure 6 illustrates an example of an AQT with 13 rectangles constructed using this rule. In this figure, rectangles in the form of vertical strips R_1, R_2, R_3, R_4, and horizontal strips R_{10} are fully contained in the square of size $2^B \times 2^B$ and are therefore listed at the root of the quadtree. Similarly, southwest quadrant of size $2^{B-1} \times 2^{B-1}$ contains rectangles R_5, R_6 and hence, the two are listed at the node reached by bit string $10(2)$.

Optimizing the Average Case Search Time

Several optimizations are possible to improve the average case performance of the naïve search procedure. First optimization that relies on pre-computation is based on following two simple observations: (1) Note that if a filter with small area is *fully contained* in another larger filter, the node at which the smaller filter is stored will always be at a lower level than the the node at which the larger filter is stored. (2) Also, if two filters have partial *intersection* overlap, they are stored either at the same node or different nodes. We use these observations to *pre-compute* a variable *MaxPriID* at each node, which records the ID of the highest priority filter among all filters found in a subtree rooted at the node. When the search visits a node, before searching its filter list, we first check if the priority of the filter currently matched by the partial search is greater than *MaxPriID*. If it is, then we conclude that no higher priority filters exist in the subtree rooted at the node that can match the packet under consideration. So we abort the search and report $match$ as the best filter match, else we continue the search along the path to a leaf node. Clearly, if this comparison fails at each node, the search ends up visiting every node along the path to the leaf node. Therefore, this optimization does not improve the worst case performance of our basic search.

However, it is possible that quadtree nodes will be unevenly populated with filter prefixes i.e. some of the nodes in the quadtree will be empty and constitute only branching points in the tree, whereas others will contain a large number of prefixes. In fact study of real routing tables has revealed that prefix lengths are not uniformly distributed but have peaks at lengths 8, 16, and 24 which correspond to prefix lengths of the original Class A, B, and C networks [10]. This suggests that we can use $k = 8$ or more bits instead of just two bits to make branching decision at each node. This can reduce the worst case complexity dramatically to $O((2W/k) \log N)$ at the cost of increasing space requirement by 2^k. However, it is still not comparable to the search time of the state-of-the-art search algorithms such as [8, 9].

However, the AQT can take advantage of a well known technique called *Fractional Cascading* [2] to reduce the $O(W \log N)$ worst case search complexity to $O(W + \log N)$, comparable to other algorithms. Also, since $\log N \leq W$, the worst case complexity is bounded by $O(2W)$. By combining k-bit trie and fractional cascading, we can reduce the worst case complexity further to $O(W/k + W)$. The details of application of fractional cascading to AQT are not presented here due to space limitations and can be found in [1].

5. EFFICIENT FILTER INSERTION AND DELETIONS

In this section, we discuss insertion, deletions or changes to a filter database represented using AQT quadtree. We will first present an overview of changes to AQT datastructure that are necessary to effect a filter insertion or deletion and then present our schemes to reduce the overheads in implementing these changes.

Overview of Implications of Insert/Delete

The insertion of a new filter $(X*, Y*)$ to a filter database represented by a AQT requires following set of operations:

1. **Find a quadtree node to which the filter belongs:** We first use the filter placement rule to find in $O(W)$ time the smallest square that will fully contain this rectangle. This in turn defines the node in the quadtree to which this the new filter belongs to. If the node does not exist, a new quadtree node is initialized and inserted.

2. **Insert the prefixes in the projection lists:** The endpoints of the X and Y prefixes of this filter are inserted to the list of prefix keys at the node using ordinary binary search procedure.

3. **Update *equal* and *less-than* fields of keys:** The insertion of a new prefix into the X and Y prefix lists at the node can alter the *equal* and *less-than* fields of each key in the prefix endpoint list at the node. So these fields must be modified consistent to the new prefix overlaps. This problem is same as inserting a new prefix to a prefix database [10].

4. **Update the fractional cascading structure:** The addition of new keys to the prefix endpoint lists alters the augmented lists at the node and possibly changes the keys that need to be passed to the parents.

Clearly, Step 1 in the procedure above can be accomplished in $O(W)$ time whereas Step 2 takes at the most $\log n$ time if the number of keys in the augmented list is n (n is bounded by N and $\log N$ is bounded by W). In the third step, in the worst case we may have to modify every existing key record and thus, may require $O(N)$ time. Note that in the last step, passing new keys to parent lists can alter the *Successor-in -original-list* information for potentially all keys in the list and thus, in the worst case can take $O(N)$ time. Therefore, the worst case complexity of inserts in the naïve implementation is $O(N)$.

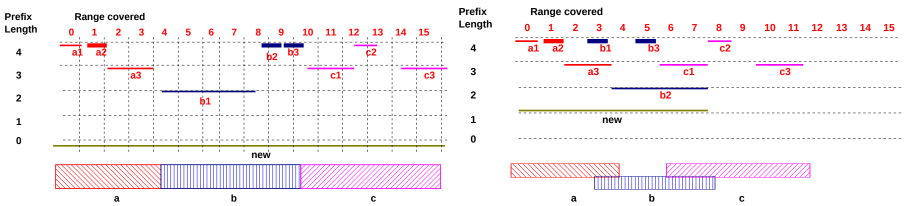
When a filter is deleted, we follow the complement of the 4-step process described above. In the following we will discuss how we can reduce complexity of steps 3 and 4 above using the *prefix partitioning* framework.

Prefix Partitions

The scheme introduced below, *Recursive Prefix Partitioning*, reduces the cost of prefix updates significantly at a modest cost being paid in search time. Additionally, it offers a tunable tradeoff between the penalty incurred for updates and searches, which makes it very convenient for a wide range of applications.

Basic Partitioning

The idea of prefix partitioning is to group N prefixes in a shallow tree of height α instead of a general binary tree of height $log(N)$. To understand the concept and implications of partitioning, we start with $\alpha = 1$ ie a single layer of partitions. We will use a simple example illustrated in Figure 7 (a): Assume an address space of 4 bits with addresses ranging from 0 to 15. This space also contains nine prefixes, labeled $a1$ to $c3$. For the fractional cascading to work, each left endpoint of a range contains the information what is covered by prefixes in higher layers. This is referred to as the less-than pointer and is the data that requires update whenever the closest covering prefix is changed.



(a) Simple partitioning example

(b) Partitions with overlaps

Figure 7 Prefix partitioning explained

Assume the prefix designated new is inserted. Traditional approaches would require the insert procedure to walk through all the prefixes and correct their less-than pointer, taking up to N steps. The *Prefix Partitioning* scheme groups these prefixes together. Assume we had grouped prefixes $a1$ to $a3$ in group a, prefixes $b1$ to $b3$ in b, and $c1$ to $c3$ in c. Note that the prefixes in the group are disjoint and hence, we can store a single overlapping prefix or less-than pointer information for all of them instead of each of them. Thus, in this example, we would remember only three such entries — one per group or partition. This improves the update time from updating each entry to just updating the information common to the group. In our example above (Figure 7 (a)), when adding the new prefix, we see that it entirely covers the partitions a, b and c. Thus, our basic scheme works well as long as the partition boundaries can be chosen so that no prefix overlaps them and the new prefix covers entire groups.

Consider one more example in Figure 7 (b), where partition A contains prefixes a_1, a_2, a_3, partition B contains prefixes b_1, b_2, b_3 and partition C contains prefixes c_1, c_2, c_3. Clearly, the partition boundaries now overlap. Although in this example it is possible to find partitioning without overlaps, in a general case prefixes that cover a large part of the address space would severely limit the ability to find enough partitions. In other words, in a general case, the boundaries between the splits are no longer well-defined; there are overlaps. The key insight that solves this problem is as follow: Instead of introducing a special case for these overlaps, we observe that only the less-than field of the key inserted for the left prefix endpoint contains information about the enclosing region. This starting address of the range covered by the prefix is thus the only relevant part. Therefore, it is not necessary to keep information about the covered range and the information about the starting point is sufficient. Since we only deal with individual addresses now, there is no need to treat overlaps and partitions can split the database at any arbitrary point. For ease of explanation, we nevertheless define a range for the partition, defined by the minimum and maximum starting address of the covered prefixes.

Continuing our example above (Figure 7 (b)), when adding the new prefix, we see that it entirely covers the partitions a, b and partially covers c. For all the fully covered partitions, we update the partitions' Best Match. Only for the partially covered partitions, we need to process their individual elements. The changes for the less-than pointers are outlined in bold in the Table 1. The real value of the less-than pointer is the entry's value, if it is set, or the partition's value otherwise. If neither the entry nor the entry's containing partition contain any information, as is the case for $c3$, the packet does not match a prefix (filter) at this level.

Table 1 Updating Less-Than Pointers

Entry /Group	Old<	New<
a1	—	—
a2	a1	a1
a3	a2	a2
a	—	**new**

Entry /Group	Old<	New<
b1	a3	a3
b2	b1	b1
b3	b2	b2
b	—	**new**

Entry /Group	Old<	New<
c1	b2	b2
c2	c1	c1
c3	—	—
c	—	—

Generalizing to p partitions of e entries each, we can see that any prefix will cover at most p partitions, requiring at most p updates.Thanks to the starting-address rule, all partitions are now disjoint. Therefore at most two partitions can be partially covered, one at the start of the new prefix, one at the end. In a simple-minded implementation, at most e entries need to be updated in each of the split partitions. If more than $e/2$ entries require updating, instead of updating the majority of entries in this partition, it is also possible to relabel the container and update the minority to store the container's original value. This reduces the update to at most $e/2$ per partially covered prefix, resulting in a worst-case total of $p + 2e/2 = p + e$ updates.

As $p * e$ was chosen to be N, minimizing $p + e$ results in $p = e = \sqrt{N}$. Thus, the optimal splitting solution is to split the database into$\sqrt{N}$ sets of $\sqrt{N}$ entries each. This reduces update time from $O(N)$ to $O(\sqrt{N})$ at the expense of at most a single additional memory access during search. This memory access is needed only if the entry does not store its own less-than value and we need to revert to checking the container's value.

Extensions of this basic to multiple layers of partitioning and the update behavior are described in more detail in [10, 1].

6. PERFORMANCE ESTIMATION

Table 2 shows the worst-case update and search times we expect to see for our algorithm when running on a typical processor used in workstations or PCs. Our calculations assume that the processor accesses are from 10ns SRAMs which are cheap and widely used in PCs. Besides that, we assume worst case conditions: No data cache hits in the processor improve the performance, and the data structure is laid out in the worst possible case with almost all the entries in a single quadtree node at the bottom of the tree. All these worst-case assumptions are very unlikely to hold. We therefore expect real-world average performance to be about an order of magnitude better. Still, our numbers compare well with even the search time results of the other known schemes. Please note that the worst-case search time is independent of the actual database size.

Table 2 Worst-case search and update times for PACARS

Update ($\alpha\sqrt[\alpha]{N}$)	$\alpha = 2$	$\alpha =$ 3x	$\alpha = 4$
$N = 10,000$	$3\ldots4\mu s$	$1.2\ldots1.8\mu s$	$.4\ldots.6\mu s$
$N = 100,000$	$7\ldots9\mu s$	$1.6\ldots2.6\mu s$	$.8\ldots1.3\mu s$
$N = 1,000,000$	$20\ldots30\mu s$	$4\ldots6\mu s$	$1.5\ldots2.4\mu s$
Search (αW)	$.64\mu s$	$1.28\mu s$	$1.92\mu s$

7. CONCLUSIONS

A number of results on multi-dimensional packet classification have appeared in recent years. Some of them have been geared for hardware implementation, some for software, all of them delivering fast classification, but none of them has been designed with efficient updates in mind.

In this paper we presented space and time efficient algorithm for fast-packet filtering that use *space decomposition* to efficiently represent the search space. For N two-dimensional filters specified using prefixes of up to W bits in length, our *Area-based Quadtrees* (AQT) data structure requires $O(N)$ space, $O(\alpha W)$

search time, and $O(\alpha\sqrt[\alpha]{N})$ update complexity. Both the average and worst-case search times and memory consumption are comparable or better than other schemes known in the literature. Our algorithm clearly outperforms them when it comes to updating the database by inserting or deleting entries. Note that using well-known approaches such as *lazy deletes*, and *multibit tries*, performance of our basic schemes can be improved even further.

We have also devised an alternate scheme, called Median-based Quad Tree (MQT), that supports arbitrary filters and efficient search and update operations. One of the applications of these algorithms we are focusing on is a dynamically adapting firewall, which is currently being developed and requires sub-second update latency.

REFERENCES

[1] Buddhikot, M., Suri, S., and Waldvogel, S., "Space Decomposition Techniques for Fast Layer-4 Switching," *Bell Labs Technical Memorandum*, BL011345-990726-06TM, Lucent Bell Labs, Holmdel, NJ, 1999.

[2] Chazelle, B., and Guibas, J., L., "Fractional Cascading," *Digital Systems Research Center* Technical Report, Palo Alto, June 1986.

[3] Decasper, D., Dittia, Z., Parulkar, G., and Plattner, B., "Router Plugins: A Software Architecture for Next Generation Routers," *Proceedings of ACM SIGCOMM 98*, Vancouver, Canada, pp. 229-240, Sept. 1998.

[4] Kumar, V., P., Lakshman, T., V., and Stiliadis, D., "Beyond Best-Effort: Gigabit Routers for Tomorrow's Internet," *IEEE Communications Magazine*, pp. 152-164, May 1998.

[5] Samet, H., "Design and Analysis of Spatial Data Structures," Addison Wesley, ISBN 0-201-50255-0, 1990.

[6] Srinivasan, V., Suri, S., and Varghese, G., "Tuple Search for Fast Layer-4 Packet Classification," To appear *ACM Sigcomm'99*, Boston, Sept. 99.

[7] Srinivasan, V., and Varghese, G., "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Transactions on Computer Systems (TOCS)*, pp. 1-40, Feb. 1999.

[8] Srinivasan, V., Varghese, G., Suri, S., and Waldvogel, M., "Fast and Scalable Layer Four Switching," *Proceedings of SIGCOMM '98*, Vancouver, Canada, pp. 191-202, Sept. 1998.

[9] Stiliadis, D., and Lakshman, T., V., "Multidimensional Range Matching for Fast Packet Classification," *Proceedings of SIGCOMM '98*, Vancouver, Canada, pp. 203-214, Sept. 1998.

[10] Waldvogel, M., "Fast Prefix Matching: Algorithms, Analysis, and Applications," *Ph.D. Dissertation*, Dept. of Electrical Engg., ETH, Zürich, July 1999.