Packet Classification Consuming Small Amount of Memory

Xuehong Sun, Sartaj K. Sahni and Yiqiang Q. Zhao

***Abstract*— In order to provide more value added services, the Internet needs to classify packets into flows for different treatment. This function becomes a bottleneck in the router. High performance packet classification algorithms are therefore highly demanded.**

This paper describes a new algorithm for packet classification using the concept of independent sets. The algorithm has very small memory requirements. The search speed is neither sensitive to the size of the rule table nor to the percentage of wildcards in the fields. It also scales well from two dimensional classifiers to high dimensional ones. In particular, the algorithm is inherently parallel. Hardware tailored to this algorithm can achieve very fast search speed. The update algorithm proposed is also very fast in general.

***Index Terms*— Packet Classification, Independent Set, Algorithm**

I. Introduction

INTERNET service provider (ISP) is going to provide more value added services to end users. Examples of these services are firewall packet filtering [1], policy routing [2], virtual private network (VPN) implementations [3], traffic billing [4] and quality of service (QoS) applications such as integrated services (IntServ) [5] and differentiated services (DiffServ) [6]. In order to provide these services, the router needs to classify the packets into flows according to different criteria. These criteria form rules which are based on L2/L3/L4 fields in the packet header. This function of router is called *packet classification*.

High speed internet relies on high speed packet classification functions. In the near future, 40 Gigabit per second (OC768) wire speed is expecting to be achieved. Given the smallest packet size of 40 bytes in the worst-case, the router needs to lookup packets at a speed of 125 million packets per second. That, together with other needs in processing a packet, amounts to less than $8ns$ per packet lookup. Nowadays, one access to on-chip memory takes 1-5ns for SRAM and about $10ns$ for DRAM; One access to off-chip memory takes 10-20ns for SRAM and 60-100ns for DRAM. This figure shows that it is highly demanding to develop high speed packet classification algorithms. It also shows that it is very difficult for serial algorithms to achieve ideal wire speed. Developing parallel algorithms and integrating parallel or pipeline mechanism into hardware seem a must for the future packet classification.

The speed of a packet classification algorithm is measured by the times of memory access. Other performance metrics for a packet classification algorithm include memory storage requirements, update speed and scalability to both the number of the rules and the number of fields included in the rules, among possible others.

We propose a new algorithm using the concept of independent sets. The new algorithm is theoretically sound. Experimental studies have shown that its performance is at least comparable to best available algorithms. Specifically, the algorithm could convert a higher dimensional classification problem into a lower dimensional one. The algorithm has very small memory requirements. The memory factor is expected to be smaller than two (the $memory\ factor$ is the ratio of the total amount of memory used to that needed to store the rules). This factor was best reported in existing algorithms as four [7]. The search speed of our algorithm is neither sensitive to the size of the rule table nor the percentage of wildcards in the fields. It scales well from two dimensional classifiers to high dimensional ones. The update algorithm proposed is also very fast in general. In particular, the algorithm is inherently parallel. It is easy to exploit the parallel mechanism in the hardware. One of the possible limitations of the new algorithm is that it depends on the characteristics in the rule table. Experiments show that the memory access times can be as low as 15 and as high as 100 for two-dimensional classification problems with a table size of 30000 (assuming that one dimensional range search uses four memory accesses).

The rest of the paper is organized as follows. In Section II, we highlight results from some existing algorithms. In Section III, the packet classification problem is defined and the related notation is developed. The concept of independent sets and the details of the new algorithm are described in Section IV. In Section V, results of experimental studies are presented. Concluding remarks are made in Section VI.

II. Previous Work

Surveys on packet classification algorithms were given in [8], [9]. Here, we highlight performance measurements for some of the algorithms.

The Recursive Flow Classification (RFC) [10] is very fast for a search. However, the memory requirement is so large and the preprocessing time is so slow that it is not suitable for large classifiers. Reference [11] proposed a Bit Vector (BV) search algorithm. For a d-dimensional classifier, the storage

This research is an initiative of Mathematics of Information Technology and Complex Systems, MITACS (www.mitacs.math.ca) and the National Capital Institute of Telecommunications, NCIT (www.ncit.ca) in Collaboration with Alcatel's Research and Innovation Centre in Ottawa, Canada (www.alcatel.com).

Xuehong Sun and Yiqiang Q. Zhao are with Carleton University. Sartaj K. Sahni is with University of Florida.

TABLE I
PERFORMANCE COMPARISON WITH OTHER SCHEMES.

Algorithm	Worst-case search time	Worst-case storage
This Paper	I	N
Tuple Space Search [13]	N	N
Grid-of-tries [15]	W^{d-1}	dWN
FIS-tree [7]	$(l+1)W$	$lN^{1+1/l}$
BV [11]	$dW+N/m$	dN^2
RFC [10]	d	N^d
HiCuts [14]	d	N^d
Cross-producting [15]	dW	N^d

requirement is $O(dN^2)$, where N is the number of rules in the classifier. Query time is d times that of the time needed for a range search, plus d times that of the time for a bit vector fetching. This is equal to N/w, where w is the size of cacheline. Reference [12] added new techniques to the BV algorithm and reported an order of magnitude improvement on performance over the standard BV algorithm, with a small price of increasing memory requirements. The tuple space search algorithm [13] has a small memory requirement ($O(N)$), however, the search speed depends on the number of tuples in the classifier and it supports only prefixes rather than arbitrary ranges. In addition, the use of hashing makes the time complexity of searches and updates nondeterministic [8]. The Fat Inverted Segment tree (FIS-tree) was proposed in [7]. The level of the FIS-tree can be adjusted to make a tradeoff between the search speed and memory requirements. Under the assumption that the cacheline is 32 bytes large and the entry size of a rule is 12 bytes, [7] reported that for a two dimensional classifier with more than 10^6 rules, the search needs less than 22 (17 respectively) memory accesses using three (two respectively) levels of the FIS-tree. The memory is at most 4.1 (7 respectively) times that of the rule table size. They did not report any experimental study on multifield classifiers. However, they pointed out that the memory requirement and memory accesses increase with a factor of l as the dimension d increases, where l is the number of levels in the FIS-tree. Other algorithms include HiCuts [14], Grid-of-tries [15], Cross-producting [15] and Set-pruning tries [16]. By refering to [8], we made a table (Table I) to compare performance of our paper with others. In the table, m is the memory width used in [11]. W is 32 for IPv4 and 128 for IPv6. I is number of independent sets (explained in this paper) which is comparable with W and much smaller than N. From this table, we can see our algorithm uses the smallest amount of memory. Though the time of RFC and HiCuts is smaller than ours, the amount of memory they consume is much large.

III. PACKET CLASSIFICATION PROBLEM

In the packet classification application, packets are classified into flows according to policy or routing information. The policy is specified using fields in the header of a packet. Specifications of fields are called *rules*. So flows are specified by rules applied to incoming packets. Each rule consists of several fields, say d. A collection of rules is called a *classifier*. Each field is either an exact value or a prefix or a range. In fact, exact values and prefixes are special ranges. In this paper, we treat fields as arbitrary ranges. Each rule also has a priority index number. Usually, the rules in a classifier are sorted according to their priorities. This index number is necessary since a packet may match more than one rule. In this case, the rule with the highest priority index is chosen. Let C be the classifier; or $C = \{R_1, \ldots, R_N\}$, where R_i $(i = 1, \ldots, N)$ is a rule and N is the number of the rules in the classifier. For each rule R_i, let $R_i = (F_1^i, \ldots, F_d^i)$, where F_j^i $(j = 1, \ldots, d)$ is the jth field. Throughout the paper, a field or a range of integers is expressed as $F = [b, e]$, which means all integers greater than or equal to b and smaller than or equal to e. b and e are called the *begin point* and *end point* of the field F respectively. For example, if the field is an IPv4 destination address, then either point is an integer between 0 and $2^{32} - 1$.

When a packet is arriving, the values f_i $(i = 1, \ldots, d)$ from the relevant d fields are extracted and expressed as $P = (f_1, \ldots, f_d)$. We say that a rule $R = (F_1, \ldots, F_d)$ is matched by the packet, if $f_i \in F_i$ for all $i = 1, \ldots, d$. Among the all matched rules, the rule with the highest priority index defines the flow that the packet belongs to.

With the above definitions, a rule can be considered as a hyperrectangle in the d-demensional space. A classifier is a set of such hyperrectangles. Hyperrectangles in the classifier might be overlapped. A packet is then a point in the d-demensional space. Thus, packet classification is equivalent to finding all hyperrectangles which contain the query point. This resembles the point location problem in computational geometry [17]. The difference between the packet classification and the point location problem is that hyperrectangles in the point location problem are nonoverlapping, while hyperrectangles in the packet classification problem may overlap. Hence, the packet classification is more complex than the point location problem. However, structures and characteristics in classifiers could be exploited to develop high performance packet classification algorithms. Such packet classification algorithms may sidestep the performance upper bound achieved in the point location background. The algorithm to be developed in this paper serves as one such example.

IV. DEVELOPING THE ALGORITHM

A. Independent Sets

Our algorithm is based on the new concept of the independent sets of rules. We first give the formal definition of the independent sets and then explain the motivation behind the concept. **Definition**: Let $C = \{R_1, \ldots, R_N\}$, where R_i $(i = 1, \ldots, N)$ is a rule. For index k, $1 \leq k \leq d$, two rules $R_i = (F_1^i, \ldots, F_d^i)$ and $R_j = (F_1^j, \ldots, F_d^j)$ are called *independent* along dimension k, if $F_k^i \cap F_k^j = \phi$. For a set S of rules, if any two rules in it are independent along dimension k, we call S an *independent set* along dimension k, which is denoted as an I_k-set or simply an I-set if no confusion arises.

The number of the elements in a set A is referred to as the size of the set A denoted by $|A|$. For a classifier, consider its all possible independent sets along dimension k. An independent set with the largest size is defined as a *maximum independent*

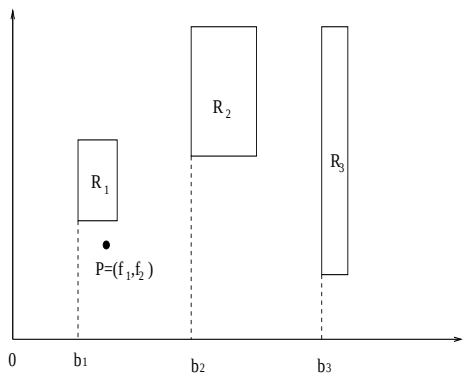

Fig. 1. An example of an independent set.

Consider dimension i. Let C be the classifier. Project all rules in C into dimension i. Let R be the resulting set of ranges in this dimension. R is sorted according to the right end points. Let M be the set of selected non-overlapping ranges of R. Let $S = \phi$.
1. If R is empty, then M is a maximum non-overlapping range set; terminate.
2. From R select the range r with the smallest right end point. If there are more than one such ranges, randomly select one. Add r to M. Remove r from R. Remove from R all ranges that overlap with r. Go to step 1.
The set of rules which project to M is a maximum independent set along dimension i.

Fig. 2. A greedy algorithm for finding a maximum independent set.

set along dimension k in C. A classifier C may have more than one maximum independent set. An independent set with the largest size among all independent sets along all dimensions is called a *global maximum independent set*.

The motivation of introducing I_k-sets is that rules in an I_k-set are easy to distinguish. For example, $S = \{R_1, R_2, R_3\}$ as in Fig. 1 is an I_1-set in the two-dimensional space. b_1, b_2 and b_3 are the *begin* points of field one in rules R_1, R_2 and R_3 respectively. b_1, b_2 and b_3 define the search intervals, say $[b_1, b_2)$, $[b_2, b_3)$ and $[b_3, \infty)$, in a one dimensional space. Apparently, each interval contains only one rule. Let each interval store all fields of the corresponding rule. In order to query a point, say, $P = (f_1, f_2)$, we only need to search for the interval that f_1 belongs to in one dimension. After the interval has been found, we compare the point with the rule stored in the interval. If the point is contained in the rule, there is a match. Otherwise, there is no rule matched by the point.

We can easily find two advantages here. One is that we only need the begin points of the field in rules to form a range search structure instead of using both begin points and end points. This reduces the search points to as at most one half as in a traditional range search algorithm, e.g. in [7]. The other advantage is that we only need to search in one dimension rather than in all dimensions. This means that we may only need to perform one range search for a multidimensional packet classification problem.

Based on the concept of independent sets, we can develop a new algorithm. The general procedure of implementing the algorithm is described as follows. Given a classifier $C = \{R_1, \ldots, R_N\}$, we try to separate from it a global maximum independent set. Then, from the set of the remaining rules (treated as a new classifier), we separate a global maximum independent set with respect to the new classifier, and continue the process until the set of the remaining rules is empty. More formally, assume that $I[1]$ is a global maximum independent set separated from C. Let $C_1 = C - I[1]$, we then find a global maximum independent set $I[2]$ with respect to C_1. Let $C_2 = C_1 - I[2]$, we continue the process until the resulting classifier is empty. Thus, C is divided into a collection of independent sets $\{I[1], I[2], \ldots, I[s]\}$. Next, we group the independent sets $\{I[1], I[2], \ldots, I[s]\}$ according to the dimension. Two independent sets belong to the same group if they are independent along the same dimension. Let $G_1, G_2, \ldots, G_d$ be the resulting groups, where d is the number of dimensions for rules in the classifier C. G_i $(i = 1, \ldots, d)$ is the group of independent sets which are independent along dimension i. Note that a group may be empty. We search in all nonempty groups of $G_1, G_2, \ldots, G_d$ respectively. The query result can be obtained by comparing all the matches to find the rule with the highest priority.

In the following, we will discuss how to find a maximum independent set and how to create a data structure for searching G_i $(i = 1, \ldots, d)$.

B. Finding a Global Maximum Independent Set

For a classifier, there are two steps for finding a global maximum independent set. First, a maximum independent set along each dimension is found and then by comparing the size of these maximum independent sets, a global maximum independent set is identified.

A greedy algorithm in Fig. 2 can be used to find a maximum independent set along one dimension.

Essentially, through projecting rules into a dimension, the algorithm in Fig. 2 turns the finding a maximum independent set problem into the finding a maximum non-overlapping ranges problem.

We can prove the set of rules found is really a maximum independent set. In order to do so, we only need to prove M is a maximum non-overlapping range set.

We first prove a Lemma.

Lemma: Let R_1, R_2 be two arbitrary range sets. Let the sizes of a maximum non-overlapping range set of R_1 and R_2 be n_1 and n_2 respectively. Let $R = R_1 \cup R_2$. Then the sizes of a maximum non-overlapping range set of R is not greater than $n_1 + n_2$.

Proof: We prove this by contradiction. Assume there is a maximum non-overlapping range set M of R, the size of M is greater than $n_1 + n_2$. Let M_1 be all the ranges of M which are from R_1. Then $M - M_1$ is from R_2. Since M_1 is a non-overlapping range set, its size is less than or equal to n_1. Since $M - M_1$ is also a non-overlapping range set, its size is less than or equal to n_2. Thus the size of M is less than or equal to $n_1 + n_2$ which is a contradiction. $\triangle$

Input: Classifier C.
Output: Groups of independent sets $G_1, G_2, \ldots, G_d$. I[] is an array to store the I-sets.

```
/*Begin Pseudocode*/
count=0;
while(C is not empty) {
  I[count]=Find a maximum independent sets
  in C;
  count++;
  C=C-I[count];
}
/*End Pseudocode*/
```

Divide I[] into groups $G_1, G_2, \ldots, G_d$ according to dimension along which they are independent.

Fig. 3. Algorithm for dividing a classifier C into the set of independent sets.

We next prove the M in Fig. 2 is a maximum non-overlapping range set.

Theorem: Let R and M be defined as in Fig. 2. Then M is a maximum non-overlapping range set.

Proof: We prove this by induction. If $|R| = 2$, then obviously M is a maximum non-overlapping range set.

Let $|R| = n$. By induction, we assume for any $|R| < n$, the M obtained using algorithm in Fig. 2 is a maximum non-overlapping range set.

Let r be the first range with the smallest right end point. Let R_1 be the set of ranges that overlap with r. Let $R_2 = R - R_1 - \{r\}$. Let M_2 be the range set obtained using algorithm in Fig. 2. Then by assumption of the induction, M_2 is a maximum non-overlapping range set of R_2. Next we need to prove $M_2 + \{r\}$ is a maximum non-overlapping range set of R.

Since r has the smallest right end point p_r, the right end point of a range in R_1 is greater than or equal to p_r. On the other hand, since any range in R_1 overlaps with r, the left end point of the range must be less than or equal to p_r. Thus every range in R_1 with p_r. Therefore, the size of a maximum non-overlapping range set of $R_1 + \{r\}$ is one. According to Lemma, the size of a maximum non-overlapping range set of $R = R_1 \cup R_2 \cup \{r\}$ is less than or equal to $|M_2| + 1$. Since $M = M_2 + \{r\}$ is a non-overlapping range set and $|M| = |M_2| + 1$, M is a maximum non-overlapping range set. $\triangle$

In Fig. 2, the complexity of sorting the right end point is $O(Nlog(N))$ where N is the number of rules. However, only once needed for each dimension. The subsequent independent set finding does not need the sorting any more. The complexity of the algorithm itself is $O(N)$, since each range needs to be compared once.

Fig. 3 is an algorithm for dividing the classifier C into a set of independent sets.

C. Basic Data Structure for a Group of Independent Sets

In this section, we develop a data structure to facilitate the search in a group of independent sets obtained from the last section. We were inspired by the fractional cascading technique [18] in developing this data structure.

Let $G = \{I[0], I[1], \ldots, I[s^G]\}$ be a group of independent sets along dimension D obtained in the last section, where $I[k]$ $(k = 1, \ldots, s^G)$ is an independent set and s^G is the number of independent sets in G. Let $I[k] = \{R_1^k, \ldots, R_{n_k}^k\}$, where R_i^k $(i = 1, \ldots, n_k)$ is a rule and n_k is the number of rules in $I[k]$. For each $I[k]$ $(k = 1, \ldots, s^G)$, we extract the begin point b_i^k of each rule R_i^k $(i = 1, \ldots, n_k)$ along dimension D which gives a set of points $B_k = \{b_1^k, \ldots, b_{n_k}^k\}$ for each $I[k]$. All points in B_k are different, since rules in $I[k]$ are independent. We assume that all points in B_k are sorted in increasing order. Next, we merge the points in all s^G sets $B_1, \ldots, B_{s^G}$ into a master set B_0. Apparently, the number of points in B_0 satisfies $|B_0| \leq \sum_{i=1}^{s^G} |B_i|$, since two different sets B_i and B_j may contain a same point. Let $N^G = |B_0|$ and $B_0 = \{b_1^0, \ldots, b_{N^G}^0\}$. Refer to Fig. 4 for an example. The number in the rectangle (rule) is the priority of the corresponding rule. The smaller the number is, the higher the priority is. For convenience of explanation, we assume that the priority is also the index of the corresponding rule.

Next, for each k $(k = 1, \ldots, s^G)$, we add "*virtual points*" to the set B_k. We first explain why we need the virtual points by the example in Fig. 4. The interval of $[b_1^3, b_2^3]$ of B_3 corresponds to the interval of $[b_4^0, b_7^0]$ of B_0. There are four points b_4^0, b_5^0, b_6^0 and b_7^0 in the interval of $[b_4^0, b_7^0]$. Note that any rules between b_4^0 and b_5^0 and between b_5^0 and b_6^0 are dependent of rule 3; any rules between b_6^0 and b_7^0 are independent of rule 3. In order to distinguish these two situations, we add the point b_6^0 to B_3 as a virtual point. Next, we give the definition.

Let b_m^0 be the largest element in B_0. For each $b_i^k \in B_k$, let e_i^k be the end point corresponding to b_i^k. Let $b_{i+1}^k \in B_k$ be the successor of b_i^k. If the successor does not exist, let $b_{i+1}^k = b_m^0$. Assume $b_j^0, b_{j+l}^0 \in B_0$ such that $b_j^0 = b_i^k$ and $b_{j+l}^0 = b_{i+1}^k$. If there is an point $b_{j+l_o}^0 \in B_0$ with $b_j^0 < b_{j+l_o}^0 < b_{j+l}^0$ such that $b_{j+l_o}^0$ is the smallest one that $e_i^k < b_{j+l_o}^0$ then add $b_{j+l_o}^0$ to B_k as a virtual point. For convenience, if the smallest $b_1^0 \notin B_k$, then add b_1^0 to B_k as a virtual point. Each virtual point is assigned -1 as its index. Refer to Fig. 5 for the illustration. Also, in Fig. 6, the points with -1 as indices are all virtual points.

Our algorithm consists of two parts. One is an array that

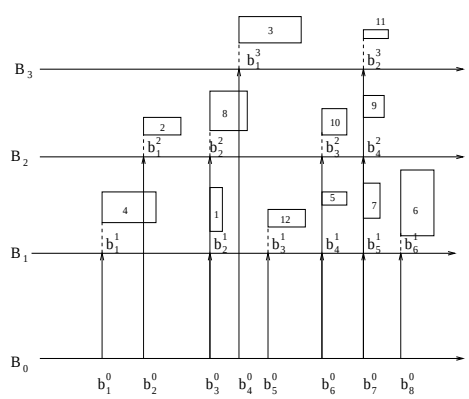

Fig. 4. Merge sets of begin points into a master set.

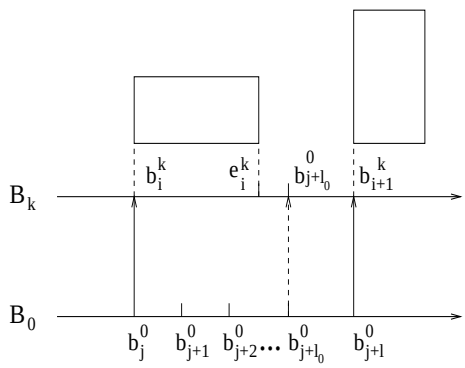

Fig. 5. Add a virtual point to B_k.

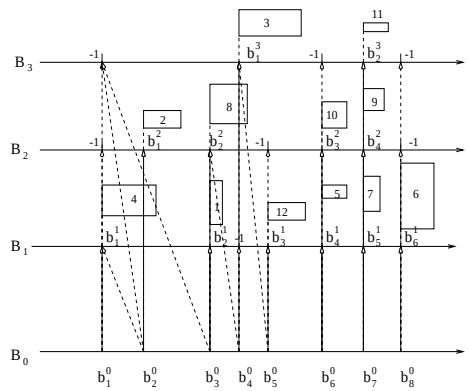

Fig. 6. A data structure example.

$b_1^0 \rightarrow$	4	-1	-1
$b_2^0 \rightarrow$	4	2	-1
$b_3^0 \rightarrow$	1	8	-1
$b_4^0 \rightarrow$	-1	8	3
$b_5^0 \rightarrow$	12	-1	3
$b_6^0 \rightarrow$	5	10	-1
$b_7^0 \rightarrow$	7	9	11
$b_8^0 \rightarrow$	6	-1	-1

Fig. 7. Index array pointed by leaves.

stores the classifier called the *classifier array*; each entry of the classifier array stores a rule which includes the fields, priority and port number. The other part is a one-dimensional range search structure based on B_0. Algorithms for one-dimensional range search are plenty in the literature. They have advantages and disadvantages. Some have small number of access memory times, while others are easy to update. The multiway search [19], [20] and van Emde Boas trees [21] are among the best algorithms. The multiway search algorithm is easy to implement, but it exploits large cacheline. Van Emde Boas trees do not need a large cacheline but are complicated to implement. We have developed a new one-dimensional range search algorithm [22]. By using a data compression technique, this algorithm consumes very small amount of memory and therefore intends to be implemented in a on-chip memory to achieve very high speed. In this paper, we use this algorithm as the one-dimensional range search algorithm to perform the experimental study.

We use B_0 to create a one-dimensional range search tree where each element in B_0 corresponds to a leaf. Each leaf points to an entry of an array which stores the indices of the rules as follows. For $b_i^0 \in B_0$ $(i = 1, \ldots, N^G)$, we find in each B_j $(j = 1, \ldots, s^G)$ the largest $b_{k_j}^j$ such that $b_{k_j}^j \leq b_i^0$. Each such a $b_{k_j}^j$ corresponds to an index of a rule (or -1 for a virtual point). Essentially, the number stored in the array indices in the data structure is the number of the rule overlapping b_i^0 in B_j. There are s^G indices. The indices are stored in an array pointed by the leaves.

Fig. 6 and Fig. 7 illustrate an example with such a data structure. In Fig. 6, -1 is the index (and priority) of a virtual point. -1 has the lowest priority. Fig. 7 shows the indices array pointed by the leaves.

D. Search

In Section C, we demonstrated how to create the data structure for one group of I-set. There are at most d such data structures needed to be created. The search and update can be performed in parallel or in serial in the d groups of I-sets.

For the search of a packet in a group, the value in the relevant field of the packet is used as the key for the search in the multiway range search tree. Find the leaf with the value that is the largest one among all which are smaller than or equal to the key. Fetch the indices pointed by the leaf. Use the indices (ignore the index -1) one by one to get the rule fields by indexing into the classifier array. Compare them with the relevant fields of the packet. If there is a match, choose the matching rule with the highest priority in this group. Continue this process until all groups have been searched. Compare all the matching rules found from the groups and choose the one with the highest priority. This rule defines the flow to which the packet belongs.

E. Update

For the dynamic packet classification, we need update the algorithm to accomodate the changes in the packet classification table. There are two kinds of updates. One is to create the table data structure from scratch whenever there is a change. Sometimes, this is called preprocessing. Another one is to modify the table data structure whenever there is a change. This is called incremental update. Usually incremental update is faster than recreating table data structure from scratch. However, incremental update may make the table data structure non optimal. We discuss both of them as follows.

1) Incremental update: For an update, we distinguish between adding a rule and deleting a rule. There are two steps for adding a rule. The first step is to find a group containing a set that the new rule is independent of. The second step is to make modification of the relevant data structure of the group

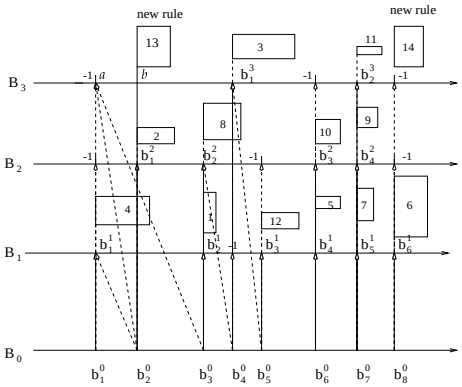

Fig. 8. Add new rules.

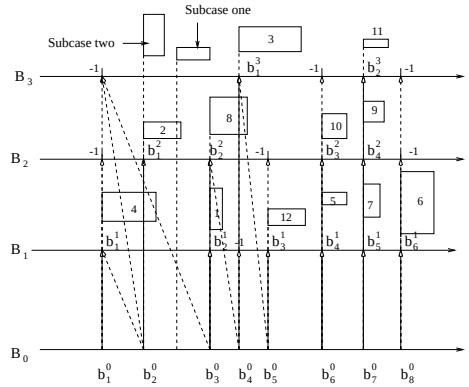

Fig. 9. The case 1.1 and case 1.2 for adding new rules.

found. We first illustrate these steps by the following example and then describe them in detail.

If we add a new rule with priority 13 to the rule sets in Fig. 6 (refer to Fig. 8). We first find the group B_3 which contains the set of rules that are independent of the new rule. As a result of adding the new rule, we need to modify the data structure. Specifically, we need to modify the entries pointed by b_2^0 and b_3^0. This is because a was the largest point in B_3 that was smaller than or equal to b_2^0 and b_3^0. Now, it becomes b. Therefore, the entry $(4, 2, -1)$ pointed by b_2^0 is modified to $(4, 2, 13)$ and the entry $(1, 8, -1)$ pointed by b_3^0 is modified to $(1, 8, 13)$.

In order to find a group containing a set that the new rule is independent of, we need to search the groups one by one. How to choose a group first can be decided by the real application. In searching a group, we extract the **end** point of the relevant field of the rule as the key. In the group, find the largest point that is smaller than or equal to the key in the corresponding multiway range search tree. Assume the leaf we found is b_k^0. Let E_k be the entry of indices pointed by the leaf b_k^0. For example, we add new rule 14 as in Fig. 8. We find $b_k^0 = b_8^0$ and the entry pointed by b_8^0 is $(6, -1, -1)$. The index 6 in the entry corresponds to rule 6. We compare the new rule 14 with rule 6. If they are not independent, we conclude that the new rule 14 is not independent of the independent set B_1. If they are independent, we conclude that the new rule 14 is also independent of the other rules in the independent set B_1 and therefore is independent of the independent set B_1. The second index in the entry is -1 which corresponds to a virtual point. Since virtual point does not correspond to a real rule, we need to find a real rule in -1's shoe. For this -1, we find rule 9 which is the prodecessor point of this -1. We compare the new rule 14 with rule 9. If they are not independent, we conclude that the new rule 14 is not independent of the independent set B_2. If they are independent, we conclude that the new rule 14 is also independent of the other rules in the independent set B_2 and therefore is independent of the independent set B_2. For the second -1, we find rule 11. The same arguement follows. According to the above example, we conclude that if the entry of indices E_k contains -1, we need to replace it with the index of predecessor rule. The following explains how to replace -1 in E_k with the index of a rule.

If there is any -1 in E_k, fetch E_{k-1}, which is pointed by the leaf b_{k-1}^0, the predecessor of b_k^0. Replace any -1 index in E_k with the index in E_{k-1} which is in the same column as the -1 index lies in. This results in a new entry. For example, in Fig. 8, the entry pointed by b_8^0 is $(6, -1, -1)$. There are two -1 indices in it. The entry pointed by b_7^0 is $(7, 9, 11)$. We replace the two -1s with 9 and 11 respectively resulting in a new entry $(6, 9, 11)$. If the new entry still contains -1s, then we fetch the entry E_{k-2} pointed by b_{k-2}^0. Replace the -1 indices with the same column indices in E_{k-2}. We continue this procedure until there is no -1 index in the resulting entry or we exhaust all the predecessor leaves in which case the -1s are kept as the indices. Then we check the indices in the resulting entry one by one and fetch the rule fields by indexing into the classifier array. Thus we can check whether the rule is independent of the new rule or not. As soon as we find a rule that is independent of the new rule, we conclude that the independent set that contains the rule is independent of the new rule. Note that, if an index is -1, it indicates that the corresponding independent set is independent of the new rule. It is possible that we cannot find an independent set in any group that is independent of the new rule. So there are two cases for the result. Case one: We find a rule that is independent of the new rule (hence the independent set the new rule is independent of). In this case, assume the index of the found rule is i_f. Case two: There is no rule in any independent set that is independent of the new rule. We separately describe the modification of the multiway range search tree and the index array for these two cases.

In case one, we have two subcases. In subcase one, the begin point of the relevant field of the new rule is not in the master set. In subcase two, the begin point of the relevant field of the new rule is already in the master set (refer to Fig. 9 for these two cases). In subcase one, we need to add a new leaf corresponding to this point to the multiway range search tree (we omit the procedure here) and we also need to insert the corresponding entry of indices in the indices array formed as follows.

Assume the index of the new rule is i_n. Get a copy of the indices pointed by b_k^0. In the copy the index i_f is changed to i_n. This forms the entry of indices of the new leaf. It is inserted in the array after the entry pointed by b_k^0. (In Fig. 9,

$b_k^0 = b_2^0$ and $i_f = -1$.) We still need to modify the index i_f in other entries. Starting from the entry of indices of b_{k+1}^0, we check at the same position as where i_f is in the entry of indices if the entry contains i_f. If it contains, change i_f into i_n and check the next leaf; otherwise, finish the modification.

There is a tradeoff here. We could use a link rather than an array to store the indices. This avoid moving all entries after b_k^0 when inserting the new entry. However, a link structure needs more memory than an array. Moreover, we still need to modify some entries after b_k^0 which, in worst-case, is as same expensive as moving all entries after b_k^0.

In subcase two, we do not need to insert a new leaf. We only need to modify the indices starting from b_k^0. The modification procedure is the same as in subcase one.

In case two, we need to add a new I-set which only contains the new rule to a chosen group. (The group is chosen randomly or with some criteria. If deleting a rule is considered, we may keep track of groups if they are empty. Empty group is the right place to put the new rule.) Thus the length of the indices in the chosen group will become one index length longer. Due to this reason, we can appoint a group as the "chosen group". In this group, the entries of indices are deliberately made large as a backup for new I-sets (the size of the backup needs to be decided by real world applications). The backup indices in the entries are all set to -1. The remainder modification is the same as in case one.

For deleting a rule, we first find the group that contains the rule and then check if the deletion of the rule results the deletion of the corresponding leaf in the multiway range search tree. In order to do that, we just need to compare the begin point of the rule with the begin points of the rules in the same entry. If there is a match, we do not need to remove a leaf; otherwise, the corresponding leaf is removed. The rest of the work is a modification of entries of indices which is the reverse procedure for adding a rule. We will not detail it here.

2) Recreate the data structure: After many times of incremental update, the data structure may become non optimal. For example, in Fig. 10, B_1 and B_2 are maximum independent sets. After the deletion of four rules (in Fig. 11), B_1 and B_2 are not maximum independent sets any more in this case. We would have combined B_1 and B_2 to form a maximum independent set. This situation will make the data structure work inefficiently. In order to avoid this situation, we may recreate the data structure. How often we need to recreate the data structure can be decided by real application.

F. Complexities of the Algorithm

Globally, we need an array to store the classifier. The size of the array is linear to the number of rules in the classifier. In addition, a multiway range search tree will be constructed corresponding to each of at most d groups of independent sets. Assume that the group G_i consists of s^{G_i} independent sets and contains N^{G_i} rules. Then the memory storage requirement for the multiway range search tree corresponding to group G_i is linear to the number of N^{G_i}. The number of indices stored in the leaves is at most $N^{G_i} * s^{G_i}$, since there are at most N^{G_i} leaves and each leaf stores s^{G_i} indices. To add all these

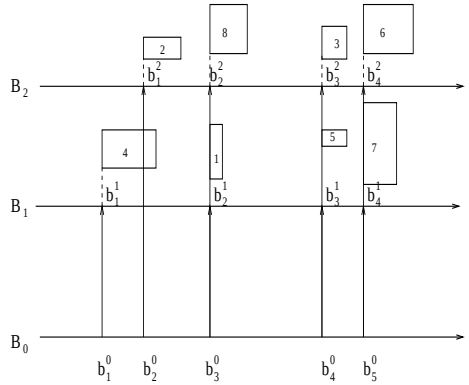

Fig. 10. Original maximum independent sets.

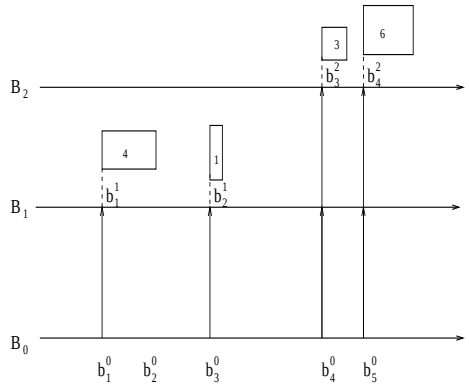

Fig. 11. After the deletion of four rules.

together, the memory storage requirement for the algorithm is upper bounded by $O(sN)$, where N is the number of rules and s is the number of total independent sets. We will see later that this bound is very loose.

The memory access times consist of the times needed to search in the multiway range search trees and to fetch the indices plus s accesses to the rules.

The update includes the search part and modification part. The complexity of the search part is similar to the search of a matching rule for a packet. The modification part needs $O(N)$ in the worst-case, where N is the number of rules. However, in reality, the worst-case is very rare. On average, the update is very fast.

The performance of our algorithm relies on s, the number of independent sets that the classifier is partitioned into. The smaller the number is, the less memory access times and memory storage is required. Fortunately, s is not expected high. Our experimental studies and studies in the literature support this conclusion. Reference [12] observed that every packet matches at most 4 rules. Similar small numbers have been seen in [23]. The prefix containment is quite rare in the backbone table and is limited to at most 6 [12], [13]. These characteristics of classifiers guarantee that s is small. The quantitative study is provided in Section IV.

G. Variations

So far, a basic version of the new algorithm has been described. In fact, many variations of this algorithm, which tailor to particular application or hardware architecture, could be developed.

One variation is that we can increase the average search speed by sorting the indices in each entry of indices. Thus, the high priority rules are searched before the low priority rules; whenever we have a match, we will stop the search procedure. However, this change will affect the update. This will make it difficult to find the predecessor or successor of a rule in the I-set. This can be solved by using extra memory (not larger than the index arrays) to keep track of the predecessor or successor of a rule.

Another variation is that, in the index arrays, we replace the index with the corresponding rule itself. This removes the need of index access and increases the search and update speed. However, this will increase the memory storage.

In the preliminary study, we use the linear search for each index in the index arrays. In fact, we can use other techniques such as a multiway search, binary search or hash search along other dimension(s) to speed up the search.

We will detail the variations in our future work.

H. A Property of the Set of Independent Sets

Our algorithm is scalable to the high dimension classification problem. The following property gives the exact meaning of this scalability.

We first give the following definition.

Definition: Let $C = \{R_1, \ldots, R_N\}$ be a d-dimensional classifier, where $R_i = [F_1^i, \ldots, F_d^i]$ $(i = 1, 2, \ldots, N)$ is a rule consisting of d fields F_j^i $(j = 1, 2, \ldots, d)$. We add one more field to each rule in C such that $R_i^e = [F_1^i, \ldots, F_d^i, F_{d+1}^i]$ and $C^e = \{R_1^e, \ldots, R_N^e\}$. The new $(d+1)$-dimensional classifier C^e is called the $(d+1)$-*dimensional extension* of C. The rule R_i^e is called the $(d+1)$-*dimensional extension* of R_i.

We have the following property for a group of I-sets.

Property: Assume that C is a d dimensional classifier. Let $\{I[0], I[1], \ldots, I[s]\}$ be a group of I-sets of C. Let C^e be the $(d+1)$-dimensional extension of C. Let $\{I^e[0], I^e[1], \ldots, I^e[s]\}$ be the corresponding $(d+1)$-dimensional extension of $\{I[0], I[1], \ldots, I[s]\}$. Then $\{I^e[0], I^e[1], \ldots, I^e[s]\}$ is a group of I-sets of C^e.

This property shows that instead of analyzing the set of I-sets with multifields, we can analyze the corresponding set of I-sets with two fields. The performance of searching a multiple dimensional classifier is not worse than that of searching the corresponding two dimensional classifier.

Specifically, we can use this property in the following way. In a multidimensional classifier, some fields may have small number of distinct entries and therefore do not contribute much to the I-sets. In that case, we can avoid separating I-sets along these dimensions. Thus, the high dimension classification problem is converted to low dimension classification problem. This shows why our algorithm could sidestep the lower bound that a d dimensional classification must at least perform d times one dimensional range searches. This property also shows that our algorithm scales very well with respect to the dimension.

In the following section, we only do experiments for two dimensional classification for an illustrative purpose.

V. Experimental Study

For a classifier, we first define the repeating factor of a field. For a specified field, let n_1 be the total number of entries of the field in a classifier. Let n_2 be the number of distinct entries of the field. The *Repeating factor* of the field is defined as n_1/n_2. In the IP destination address field, the repeating factor measures the average times that an IP address is used in the rules. The performance of our algorithm is directly related to the repeating factor of each field. The data used in [7] show that repeating factors are around 50 for large classifiers (with about 1M entries). The data used in [13] show that repeating factors are around 5 for small classifiers (with about 200 entries).

There is no data of classifiers available publicly due in part to the reason that ISPs and enterprises, for privacy and security reasons, protect access to their rule databases. In the literature, either proprietary data or artificial data is used for experimental study. There is no benchmark for evaluating packet classification algorithm in the industrial either. For a preliminary study as in the paper, we construct classifiers with characteristics we have seen from field data. To do this, we download IP routing tables from [24] as bases and conduct several groups of experiments for two dimensional classifiers. The results are presented as follows.

A. Classifiers without Wildcards

We first study classifiers without wildcards. In the first group of experiment, we study the effect of the repeating factor on the performance of our algorithm. In the second group we show that the size of the classifier is not sensitive to the performance of our algorithm when the repeating factor is fixed.

We use the routing table at Mae-west taken on March 15, 2002 from [24] as a base for constructing a classifier C. Each rule in the classifier contains two fields: The IP destination address and the source address.

In the first group of experiment, we generate four classifiers of the same size of 30000 with different repeating factors. In order to generate a classifier, the expected repeating factor, say f, is given first. Then, from the Mae-west routing table, $30000/f$ addresses are sampled. Next, from the $30000/f$ addresses, 30000 addresses are randomly selected as the IP destination addresses. For each destination address, the corresponding source address is randomly selected from the $30000/f$ addresses to form a rule. If the rule just formed is a duplicate rule of a previous generated one, we reselect a source address until the rule formed is unique.

Table II shows the resulting classifiers. The first column is the expected repeating factor (f). The second column is the number of distinct destination addresses (des), the third column is the repeating factor (rf) of destination address field and so on. The last column is the number of independent sets

of the classifiers. We can see that the number of independent sets increases as the repeating factor increases. This can be explained intuitively as follows. When an address is repeated in the same field of two rules, these two rules are not independent along the dimension corresponding to the field. If an address is repeated in the same field of r rules, then the number of Independent set along the corresponding dimension is obviously at least r. Hence, the larger r is, the larger the number of Independent set tends to be. Since the repeating factor measures the average times that an IP address is repeated in the rules, it has the same property that the larger it is, the larger the number of Independent set tends to be.

In the second group of experiment, six classifiers of different sizes with the same expected repeating factor of 30 are constructed. The base prefixes are from the AADS routing table taken on March 15, 2002 from [24]. The results are shown in Table III. It shows that the number of independent sets of the classifiers is not sensitive to the size of classifiers provided that the repeating factor is unchanged. This shows that our algorithm is not sensitive to the size of classifiers with a similar repeating factor. By observing data used in the literature, we found that it is rare that the repeating factor exceeds 100 even for large size classifiers. Together with our experimental study, we believe that our algorithm scales well to the size of classifiers. We also envision that our algorithm scales well to IPv6, since points space in IPv6 is much larger than that in IPv4 and therefore the repeating factors should be much lower.

B. Classifiers with Wildcards

We construct three classifiers with different percentages of wildcards in them. The base prefixes are from the AADS routing table taken on March 15, 2002 from [24]. The size of base prefixes is 30000. Given a percentage p, $p/2$ percent rules have wildcards as their destination fields and $p/2$ percent rules have wildcards as their source fields. Altogether, p percent rules have wildcards. The number of rules is 30000 for all three classifiers. Table IV shows that the number of independent sets of the classifiers is not sensitive to percentage of wildcards in the classifiers. We note that the repeating factor is small and hence the number of independent sets is small. In fact, in a two-dimensional classifier, it is not possible to generate high percent wildcards in one field with a high repeating factor in the other field. We can prove the following property.

Property: Assume that C is a two-dimensional classifier without duplicated rules. If there are $p\%$ rules that have wildcards in one field, then the repeating factor in the other field is less than $100/p$.

Proof: If two rules are wildcarded in one field, then the entries of these rules in the other field are distinct. Since there are $p\%$ rules that have wildcards in one field, the entries of these $p\%$ rules in the other field are distinct. Therefore, the number of distinct entries in the other field is greater than $p\%$ of the total rules. Hence the repeating factor in the other field is less than $100/p$. $\triangle$

For example, if 50% wildcards were generated in the destination field, then the repeating factor in the source field can not be higher than two.

TABLE II

EXPERIMENTS WITH DIFFERENT REPEATING FACTORS.

f	des	rf	src	rf	I-set
2	15422	1.95	14321	2.09	9
10	2807	10.69	2960	10.14	23
20	1422	21.10	1610	18.63	33
60	489	61.35	498	60.24	74

TABLE III

EXPERIMENTS WITH DIFFERENT TABLE SIZES.

rules	des	rf	src	rf	I-set
2000	65	30.77	62	32.26	34
10000	323	30.96	360	27.78	40
20000	677	29.54	710	28.17	43
100000	3499	28.58	3287	30.42	45
200000	7155	27.95	6567	30.46	48
1000000	32778	30.51	33698	29.68	61

TABLE IV

EXPERIMENTS WITH DIFFERENT PERCENT OF WILDCARDS.

% wildcards	des rf	src rf	I-set
10	1.46	1.46	6
30	1.59	1.58	7
50	1.72	1.72	6

C. One Scenario of Implementation

We choose the third classifier in Table II for the discussion of implementation. There are 33 I-sets for this classifier among which 7 are I_1-sets and 26 I_2-sets. Corresponding to these two groups of I-sets, two one-dimensional range search trees [22] are created. The range search tree corresponding to field one has at most 2844 leaves, since the number of distinct destination prefixes is 1422 (each prefix has two end points, different prefixes may share end point). Each leaf points to 7 rule indices. Since the number of rules is 30000, each index uses 16 bits (in fact 15 bits are enough). So the memory for the index arrays is less than 19908 bytes. Using the same argument, we find the memory for the index arrays in the other range search tree is less than 83720 bytes. For the array storing the rules, we assume that each rule uses 128 bits. Among the 128 bits, 32 bits are for begin point of the destination address field and 5 bits for the length of the destination prefix. Another 37 bits are for the source address field. 15 bits are used for specifying priority, 32 bits for the port number and 7 bits are empty. So 30000 rules need 480000 bytes of memory. Together, 583628 bytes memory are needed excluding for the range search trees. The range search trees need about 15 K bytes. Experimental study shows that totally about 600 K bytes are needed. Comparing the original rule table, our algorithm increases the memory requirement in less than a factor of one. This comes out with no surprise. On one hand, the memory requirement is large if the number of I-sets is large; on the other hand, if the number of I-sets is large, the repeating

TABLE V
EXPERIMENTAL COMPARISON WITH OTHER SCHEMES.

Algorithm	# of rules	# of fields	rule length (bits)	memory size (KBytes)	memory efficiency
This Paper	30,000	5	128	600	1.3
Tuple Space Search [13]	278	2	64	NA	NA
Grid-of-tries [15]	20,000	2	64	2,000	12
FIS-tree [7]	1,000,000	2	64	100,000	12
BV [11]	512	5	128	640	80
RFC [10]	15,000	4	112	4,000	20
HiCuts [14]	1,700	4	112	1,000	40
Cross-producting [15]	50	2	64	1,525	4,000

factors should be large hence the number of distinct values in a field is small, thus the number of leaves in the multiway range search tree is small. Therefore, small memory requirement is needed to store the indices array.

The number of memory accesses for the range search tree is 4 each. The number of memory accesses for the indices depends on the memory width. We assume that 128 bits memory width is used. Then, 5 memory accesses are needed for fetching the $7 + 26$ indices and 33 memory accesses are needed for fetching the rules including port number. Totally 46 memory accesses are needed. This is the worst case plain implementation. In real application, we may easily exploit parallel and/or pipelined implementation. The range search tree only needs one memory access using pipelined implementation. Since the number of the indices associated with each leave is rather small, it is possible to rely upon a small, special-purpose hardware unit (e.g., a TCAM unit) to perform the comparison in parallel. Since the structure of the range search tree and the indices array are separated, their operations can also be pipelined. Thus one memory access is needed for one search.

In Table V, we give the experimental results for different schemes. As mentioned before, there is no benchmark for evaluating packet classification algorithm in a unified manner. The figures in the table only gives rough idea about the performance of each scheme. The memory efficiency column is obtained by dividing the memory size by the number of rules and the rule length. It shows that our algorithm uses significant small amount of memory comparing to other algorithm. This table is consistent with Table I.

Using dynamic range search algorithm [25], we can extend the packet classification algorithm to support fast update. In general, there is a trade-off between the update speed and the memory requirement. Since the detail is related to the one dimensional range search, it is out of the scope of this paper.

VI. CONCLUSIONS

We developed a novel packet classification algorithm based on independent sets. We proposed a basic data structure and an update algorithm for the data structure. We also conducted an experimental study on our algorithm.

As mentioned earlier, packet classification algorithms are measured by times of memory access, memory storage requirements, update speed and scalability, etc. Existing algorithms could perform well with respect to one or two of these measurements. Our algorithm performs well in all aspects of the criteria. It seems that our algorithm is the first to achieve such a success: Small memory requirements, fast search and update speed and scalability to large classifier, to multidimensional classifier. The algorithm is feasible for parallel implementation. With these merits, our algorithm could be a candidate among the possible bests for the future high speed packet classification task.

ACKNOWLEDGMENTS

The authors would like to thank Gerard Damm and Milan Zoranovic of Alcatel for their valuable comments and suggestions.

REFERENCES

[1] S. Bellovin and W. Cheswick, “Network Firewalls,” *IEEE Communications Magazine*, v32, n9, 1994, pp. 50-57.
[2] Y. Jyh-haw, C. Randy and N. W. Richard, “Interdomain Access Control with Policy Routing,” *Proceedings of the IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Oct. 1997, pp. 46-52.
[3] B. Gleeson, A. Lin, J. Heinanen, G. Armitage and A. Malis, “A Framework for IP Based Virtual Private Networks,” *RFC 2764*, February 2000.
[4] Richard Edell, Nick McKeown, and Pravin Varaiya, “Billing Users and Pricing for TCP,” *IEEE Journal on Selected Areas in Communications*, v13, n7, September 1995, pp. 1162-1175.
[5] R. Braden, D. Clark and S. Shenker, “Integrated Services in the Internet Architecture: an Overview,” *RFC 1633* , June 1994.
[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, “An Architecture for Differentiated Services,” *RFC 2475*, December 1998.
[7] A. Feldman and S. Muthukrishnan, “Tradeoffs for Packet Classification,” *Proceedings of Infocom*, v3, March 2000, pp. 1193-2002.
[8] Pankaj Gupta and Nick McKeown, “Algorithms for Packet Classification,” *IEEE Network Special Issue*, March/April 2001, v15, n2, pp 24-32.
[9] X. Sun, “IP Address Lookups and Packet Classification: A Tutorial and Review,” Technical Report, Carleton University, 2002.
[10] Pankaj Gupta and Nick McKeown, “Packet Classification on Multiple Fields,” *Proc. Sigcomm, Computer Communication Review*, v29, n4, September 1999, pp. 147-160.
[11] T. V. Lakshman and D. Stiliadis, “High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching,” *Proceedings of ACM Sigcomm*, September 1998, pp. 191-202.
[12] Florin Baboescu and George Varghese, “Scalable Packet Classification,” *ACM SIGCOMM* , 2001, pp. 199-210.
[13] V. Srinivasan, S. Suri and G. Varghese, “Packet Classification Using Tuple Space Search,” *Proceedings of ACM Sigcomm*, September 1999, pp. 135-146.
[14] Pankaj Gupta and Nick McKeown, “Packet Classification using Hierarchical Intelligent Cuttings, ” *IEEE Micro*, vol. 20, no. 1, January/February 2000, pp 34-41,.
[15] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, “Fast and Scalable Layer four Switching,” *Proceedings of ACM Sigcomm*, September 1998, pages 203-14.
[16] P. Tsuchiya,“A search algorithm for table entries with non-contiguous wildcarding,” *unpublished report*, Bellcore.

[17] F. Preparata and M. I. Shamos, "*Computational Geometry: an Introduction*," Springer-Verlag, 1985.
[18] B. Chazelle and L. J. Guibas, "Fractional Cascading I: A Data Structuring Technique," *Algorithmica* , v1, n2, 1986, pp. 133-162.
[19] S. Suri, G. Varghese and P.R. Warkhede, "Multiway range trees: scalable IP lookup with fast updates ," *Proc. IEEE GLOBECOM '01* , v3 2001, pp. 1610 -1614.
[20] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *Proc. IEEE INFOCOM 98* , Apr. 1998, pp. 1248-56.
[21] D. E. Willard, "Log-logarithmic Worst-case Range Queries Are Possible in Space $\Theta(N)$," *Information Processing Letters*, 17(2), 1983, pp. 81-84.
[22] X. Sun and Yiqiang Q. Zhao, "An On-chip IP Address Lookup Algorithm," Technical Report, Carleton University, 2003.
[23] Pankaj Gupta and Nick McKeown, "Packet Classification Using Hierarchical Intelligent Cuttings," *IEEE Micro*, v20, n1, January/ February 2000, pp. 34-41.
[24] http://www.merit.edu/ipma/routing_table/.
[25] X. Sun, Sartaj K. Sahni and Yiqiang Q. Zhao, "Fast Update Algorithm for IP Forwarding Table Using Independent Sets,"*Lecture Notes in Computer Science, HSNMC04*, June 30-July 2, 2004 Toulouse, France.