Packet Classification Using Independent Sets

Xuehong Sun and Yiqiang Q. Zhao
School of Mathematics and Statistics
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, Canada K1S 5B6
E-mail: {xsun, zhao}@math.carleton.ca

Abstract

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.

1. Introduction

Internet service provider (ISP) is going to provide more value added services to end users. 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 (worst-case), the router needs to lookup packets at a speed of 125 million packets per second. That, together with other needs in processing, amounts to less than 8ns per packet lookup. Nowadays, one access to on-chip memory takes 1-5ns for SRAM and 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.

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. Thus, the lower bound that a d dimensional classification problem requires to perform at least d one-dimensional range searches is expected to break. 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 [2]. 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. 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 2, 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 3. In Section 4, results of experimental studies are presented. In Section 5, results from some existing algorithms are highlighted. Concluding remarks are made in Section 6.

2. 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.

Proceedings of the Eighth IEEE International Symposium on Computers and Communication (ISCC'03)
1530-1346/03 $17.00 © 2003 IEEE

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 [1]. The difference between the packet classification and the point location problem is that hyperrectangles in the point location problem are not overlapping, 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 break the performance upper bound achieved in the point location background. The algorithm to be developed in this paper serves as one of such examples.

3. Developing the Algorithm

3.1. 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 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 [2]. The other advantage is that we only need to search in one dimension rather than in all dimensions. This means

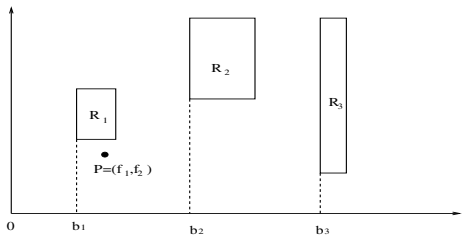

Figure 1. An example of an independent set.

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 search in G_i $(i = 1, \ldots, d)$.

3.2. 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. Finding a maximum independent set can be converted into the independent set problem in graph theory.

For each dimension k $(k = 1, \ldots, d)$, a graph $\mathcal{G}_k$ corresponding to the independence of rules along dimension k can be created as follows.

A vertex in the graph $\mathcal{G}_k$ corresponds to a rule in the classifier. There is an edge between two vertices if and only if the corresponding rules are not independent along dimension k. A set of vertices is called an *independent set* if no pair of vertices in the set defines an edge in $\mathcal{G}_k$. An independent set with the maximum size is called a *maximum independent set* in $\mathcal{G}_k$.

To find a maximum independent set along dimension k in the classifier is equivalent to finding a maximum independent set in the graph $\mathcal{G}_k$. The maximum independent set problem in graph theory is an NP-problem [3]. Exact algorithms are not scalable to graphs with large number of vertices. Numerous references where heuristic solutions are provided for solving this problem can be found in [4]. In our preliminary study, we choose a simple one as shown in Fig. 2 for finding a computed maximum independent set along one dimension. The intuition behind the algorithm is that the vertices of small degrees are less likely dependent of other vertices in the graph. Therefore, they have high priorities to be chosen for the testing of independence. Note that a computed maximum independent set may not be a true maximum independent set.

Let V[] be the vertex set of the graph $\mathcal{G}_k$. n is the number of vertices in the graph. V[] is sorted according to the vertex degrees in increasing order. Let I be the computed maximum independent set.

```
/*Begin Pseudocode*/
I={V[0]};
for i from 1 to n-1
  if V[i] is independent of all ver-
tices in I
  then I=I+{V[i]} endif
endfor
/*End Pseudocode*/
```

Figure 2. A heuristic algorithm for finding a maximum independent set in a graph.

3.3. 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 [5] 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

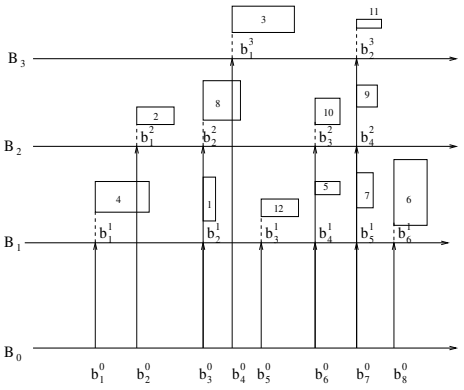

Figure 3. Merge sets of begin points into a master set.

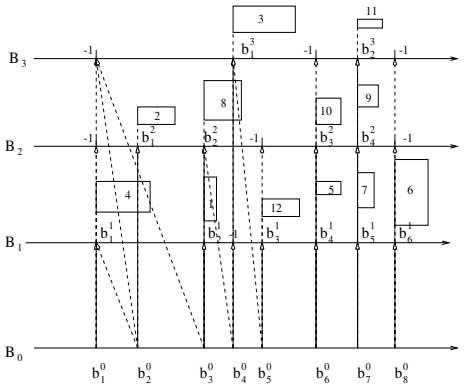

Figure 4. A data structure example.

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

Figure 5. Index array pointed by leaves.

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. 3 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. 3. The interval of $[b_1^3, b_2^3]$ of B_3 which 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 $b_{j+l_0}^0 \in B_0$ with $b_j^0 < b_{j+l_0}^0 < b_{j+l}^0$ such that $b_{j+l_0}^0$ is the smallest one that $e_i^k < b_{j+l_0}^0$ then add $b_{j+l_0}^0$ 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. As in Fig. 4, the points with -1 as indices are all virtual points.

Our algorithm consists of two parts. One is an array that 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 [6, 7] and van Emde Boas trees [8] 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. Here we choose the multiway search algorithm for the preliminary study. We use B_0 to create a multiway 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). There are s^G indices. The indices are stored in an array pointed by the leaves.

Fig. 4 and Fig. 5 illustrate an example with such a data structure. In Fig. 4, -1 is the index (and priority) of a virtual point. -1 has the lowest priority. Fig. 5 shows the indices array pointed by the leaves.

3.4. Search

In Section 3.3., 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.

3.5. Update

For the dynamic packet classification, we need update the algorithm to accommodate 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. Our algorithm is amenable to incremental update. Details of the update will be presented in an expanded version of this paper.

3.6. 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 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 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 [9] observed that every packet matches at most 4 rules. Similar small numbers have been seen in [10]. The prefix containment is quite rare in the backbone table and is limited to at most 6 [9, 11]. These characteristics of classifiers guarantee that s is small. The quantitative study is provided in Section 4.

3.7. 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.

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.

4. 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 [2] show that repeating factors are around 50 for large classifiers (with about 1M entries). The data used in [11] show that repeating factors are around 5 for small classifiers (with about 200 entries).

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 [12] as bases and conduct several groups of experiments for two dimensional classifiers. The results are presented as follows.

4.1. 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 [12] 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 1 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.

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 [12]. The results are shown in Table 2. 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.

Table 1. 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 2. 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

4.2. 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 [12]. 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 3 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 en-

Table 3. 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

tries 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.

4.3. One Scenario of Implementation

We choose the third classifier in Table 1 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 multiway range search trees are created. The multiway range search tree corresponding to field one has at most 1422 leaves, since the number of distinct destination addresses is 1422. 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 multiway 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 multiway range search trees. 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 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 memory access times for a search depends on the cachline. We assume that 128 bits cacheline 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. Excluding for the multiway range search, we need 38 memory accesses.

Update is fast. Since there are only 1610 distinct source addresses, the maximum number of entries of the index arrays to be modified is 1610 rather than 30000.

5. Previous Work

It is difficult to make an exact comparison among all other algorithms, not only because different algorithms represent different tradeoffs, but also the databases used for experiments are different and the performance measurement is rather coarse. However, we can highlight some performance measurements for some of the algorithms. More detailed discussions on packet classification algorithms were given in [13, 14].

The Recursive Flow Classification (RFC) [15] 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 [16] proposed a Bit Vector (BV) search algorithm. For a d-dimensional classifier, the storage requirement is $O(dN^2)$, where N is the number of rules in the classifier. Query time is d times of the time needed for a range search plus d times of the time for a bit vector fetching which is equal to N/w, where w is the size of cacheline. Reference [9] 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 requirement. The tuple space search algorithm [11] needs 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 [13]. The Fat Inverted Segment tree (FIS-tree) was proposed in [2]. 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, [2] 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 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.

6. 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.

7. Acknowledgments

The authors would like to thank Gerard Damm and Milan Zoranovic from Alcatel for their valuable comments and suggestions. We also thank the anonymous referees for their valuable comments for revising the paper.

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).

References

[1] F. Preparata and M. I. Shamos, "*Computational Geometry: an Introduction,*" Springer-Verlag, 1985.

[2] A. Feldman and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proceedings of Infocom*, v3, March 2000, pp. 1193-2002.

[3] D. S. Johnson and M. A. Trick (eds.), "Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge," v26, *DIMACS*, American Mathematical Society, 1996.

[4] I. Bomze, M. Budinich, P. Pardalos and M. Pelillo, "The Maximum Clique Problem," In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 4, Kluwer Academic Publishers, Boston, MA, 1999.

[5] B. Chazelle and L. J. Guibas, "Fractional Cascading I: A Data Structuring Technique," *Algorithmica* , v1, n2, 1986, pp. 133-162.

[6] 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.

[7] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *Proc. IEEE INFOCOM 98* , Apr. 1998, pp. 1248-56.

[8] D. E. Willard, "Log-logarithmic Worst-case Range Queries Are Possible in Space $\Theta(N)$," *Information Processing Letters*, 17(2), 1983, pp. 81-84.

[9] Florin Baboescu and George Varghese, "Scalable Packet Classification," *ACM SIGCOMM* , 2001, pp. 199-210.

[10] Pankaj Gupta and Nick McKeown, "Packet Classification Using Hierarchical Intelligent Cuttings," *IEEE Micro*, v20, n1, January/ February 2000, pp. 34-41.

[11] V. Srinivasan, S. Suri and G. Varghese, "Packet Classification Using Tuple Space Search," *Proceedings of ACM Sigcomm*, September 1999, pp. 135-146.

[12] http://www.merit.edu/ipma/routing_table/.

[13] Pankaj Gupta and Nick McKeown, "Algorithms for Packet Classification," *IEEE Network Special Issue*, March/April 2001, v15, n2, pp 24-32.

[14] X. Sun, "IP Address Lookups and Packet Classification: A Tutorial and Review", Technical Report, Carleton University, 2002.

[15] Pankaj Gupta and Nick McKeown, "Packet Classification on Multiple Fields," *Proc. Sigcomm, Computer Communication Review*, v29, n4, September 1999, pp. 147-160.

[16] T. V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multidimensional Range Matching," *Proceedings of ACM Sigcomm*, September 1998, pp. 191-202.