A Non-Collision Hash Trie-Tree Based Fast IP Classification Algorithm

XU Ke (徐 恪), WU Jianping (吴建平), YU Zhongchao (喻中超) and XU Mingwei (徐明伟)

Department of Computer Science and Technology, Tsinghua University
Beijing 100084, P.R. China

E-mail: {xuke,yzc,xmw}@csnet1.cs.tsinghua.edu.cn; jianping@cernet.edu.cn

Received August 29, 2000; revised June 4, 2001.

Abstract With the development of network applications, routers must support such functions as firewalls, provision of QoS, traffic billing, etc. All these functions need the classification of IP packets, according to how different the packets are processed subsequently, which is determined. In this article, a novel IP classification algorithm is proposed based on the Grid of Tries algorithm. The new algorithm not only eliminates original limitations in the case of multiple fields but also shows better performance in regard to both time and space. It has better overall performance than many other algorithms.

Keywords IP classification, hash, Trie-tree

1 Introduction

Future IP network must provide more service types and better quality of service[5], including differentiated service[1], firewalls[2], policy-based routing[3], virtual private network, traffic billing[4], etc. All these functions need the classification of IP packets.

In this paper, we first provide the mathematical model of the IP classification problem. Then we present a novel IP classification algorithm applying to multiple fields based on the two-dimension IP classification. We also compare our new algorithm with others. The simulation result shows that our algorithm has the best overall performance.

2 Mathematical Model of IP Classification

2.1 Terminology Definitions[8]

An *address* D is a bit string of W bits in length.

A *prefix* P is a bit string of the length between 0 and W. We use *length* (P) to denote the number of bits in a prefix.

A *header* H has K fields, which are denoted by $H[1], H[2], \ldots, H[K]$ respectively. Each field is a string of binary bits.

A *filter* F also has K fields. Each field $F[i]$ in a filter can specify any of the three kinds of matches: exact match, prefix match, or range match.

It is called an *exact match* iff a single value is specified for the ith filter field (i.e., $F[i]$) and the header field $H[i]$ is equal to $F[i]$.

It is called a *prefix match* iff a prefix is specified for the ith filter field and the first *length*$(F[i])$ binary bits of the header field $H[i]$ are the same as those of $F[i]$.

It is called a *range match* iff a range of values $F[i] = val1 \ldots val2$ is specified for the ith filter field and the header field $H[i]$ falls into that range, i.e., $val1 \leq H[i] \leq val2$.

This work is supported by the National "863" High-Tech Programme of China (No.863-306-ZD-07-01) and the National Natural Science Foundation of China (No.90104002).

A filter F is said to be a *matching filter* for a header H iff each field $H[i]$ of H matches the corresponding field $F[i]$ of F. The type of match is specified by $F[i]$ and could be an exact match, a prefix match or a range match.

A set of N filters is called a *filter database*, which is denoted by FS.

Each filter F has a *cost* property denoted by $cost(F)$. For $\forall F_1, F_2 \in FS$, if $cost(F_1) = cost(F_2)$ then $F_1 = F_2$. We use the *cost* property to assure that there is at most one matching filter.

2.2 Best Matching Filter Problem and IP Classification

We define the following problem as the best matching filter problem:

Given a filter database $FS \neq \emptyset$ and a header H, find the best matching filter f_{best}, which meets the following conditions:

(1) $f_{best} \in FS$

(2) f_{best} matches H

(3) $\forall f \in FS$, $f \neq f_{best}$, if f matches H, then $cost(f_{best}) < cost(f)$.

IP classification is an instance of the best matching filter problem. In theory, seven fields can be used for the filter: the destination/source IP address (32 bits each), the destination/source transport port (16 bits each), the type of service (8 bits), the protocol type (8 bits) and the flag of transport layer (8 bits). The sum of bits of these fields is 120 (we assume that all the seven fields reside in the IP packet header for the sake of convenience, although some fields are in TCP header actually). Statistical results of some actual filter databases used by ISPs show that 17% of the filters specify only one field, 23% specify three fields, and 60% specify four fields[6].

3 Related Work

The packet classification based on patterns[7] is used in the operating system when dispatching data packets of the input queue to different process spaces. It is the first algorithm avoiding linear lookup. Its performance has direct proportion to the number of fields and is independent of the number of filters. But this algorithm has very strong limitations on filters, thus it is not suitable for IP routers.

The crossproducting algorithm[8] is based on caches. For bigger classifiers, the authors propose a caching technique (on-demand crossproducting) with a non-deterministic classification time.

The modular algorithm[9] is an IP classification algorithm based on statistics. It may optimize the lookup data structure according to the distribution of filter matching ratio and IP traffic. Without enough effective statistic parameters, this algorithm cannot be practically used for IP routers now.

The RFC (Recursive Flow Classification) algorithm[6] is a simple multi-stage classification algorithm, which maps the S-bit header to the T-bit *ClassID* ($T \ll S$) step by step. It is the fastest algorithm ever known, but it needs a lot of pre-computation (usually more than ten seconds) and it may suffer from space explosion.

A solution called Grid of Tries is proposed in [8]. In this scheme, the Trie-tree data structure is extended to two dimensions. This is a good solution if the filters are restricted to only two fields, but it is difficult to extend it to apply to more fields.

A hardware-only algorithm can employ a ternary CAM (content-addressable memory). Ternary CAMs store words with three-valued digits: '0', '1' or '*' (wildcard). The rules are stored in the CAM array in the order of decreasing priority. Given a packet-header to classify, the CAM performs a comparison against all of its entries in parallel, and a priority encoder selects the first matching rule. While simple and flexible, CAMs are currently suitable only for small tables; they are too expensive, and consume too much power for large classifiers. Furthermore, some operators are not directly supported, and so the memory array may be used very inefficiently.

In this paper, we propose a novel lookup algorithm called non-collision hash Trie-tree algorithm, which is based on the Grid of Tries algorithm. The average time consumed and the space requirement of this algorithm are less than those of Grid of Tries, and it gets rid of the limitation of filters in Grid of Tries. It is the most attractive candidate algorithm if implemented by means of software.

4 Non-Collision Hash Trie-Tree Algorithm

4.1 Basic Algorithm

We mentioned that seven fields in IP packets might be the candidate fields of the filter. But actual filter databases usually use only five fields: destination/source IP addresses, destination/source ports and protocol type. The width of the protocol type field is 8 bits. To program conveniently, we extend the protocol type field to 16 bits.

The values of the destination port, the source port and the protocol type range from 0 to 65535, but in actual filters they only use a very small part of the whole range. Currently, the value of the protocol field is limited to TCP, UDP, ICMP, IGMP and (E)IGRP. In most Client/Server software architectures, ports can be roughly divided into two classes[10]. One is the reserved port which numbers in 1–1023, the other is the ephemeral port which numbers larger than 1023. Ephemeral ports are usually used in client software and are usually assigned by the operating system kernel. They are nothing but to identify an endpoint of a connection. It is almost impossible that a filter would specify a specific port larger than 1023. Usually, filters specify a range such as gt1023 meaning all ports larger than 1023 (but less than 65535). The most widely used reserved ports are 20, 21 (FTP) and 80 (www), other ports are used less frequently.

Our analysis above shows that the number of combinations of destination/source ports and protocol values in actual filters is very small. Based on this observation, we construct a two-stage look up table that can be used to lookup without collision at all. We take Table 1 as an example to illustrate our idea.

Table 1. An Example Database of Filters

ClassID	Dest-IP	Src-IP	Dest-port	Src-port	Prot
0	10.1.*.*	10.2.*.*	*	*	*
1	10.3.*.*	10.4.*.*	80	*	17
2	10.5.*.*	10.6.*.*	80	*	17
3	10.5.*.*	10.6.*.*	[20, 21]	*	6
4	10.7.*.*	10.7.*.*	*	gt 1023	6
5	*	*	*	*	*

Table 1 is a filter database that contains six filters. We assume *ClassID* is the same as the filter cost in this database. Take the destination port as an example. We assign each port in 0–65535 a bitmap. This bitmap denotes the filters the port matches (the length of bitmap is equal to the number of filters). For example, the bitmaps of ports 21 and 22 are both 100111, which means they match filters 0, 3, 4 and 5. According to these bitmaps, we classify all possible destination ports into different equivalent classes. The ports with the same bitmap belong to the same class. The bitmap of equivalent class A is denoted by $bmp(A)$. The set of all such destination port equivalent classes is denoted by D_Set. And the total number of destination port equivalent classes is denoted by D. For example, the D_Set of Table 1 is {{80}, [20,21], {0–65535 except 20, 21, 80}}. In the same manner, we construct the set of source port equivalent classes S_Set and that of protocol equivalent classes P_Set, whose element numbers are S and P respectively.

If $a \in D\_Set$, $b \in S\_Set$, $c \in P\_Set$ then the 3-tuple (a, b, c) is called a cross-combination.

Now we further divide the set of all the cross-combinations into different equivalent classes called DSP_Set. We do it as follows.

Consider two cross-combinations (a, b, c) and (d, e, f). If $bmp(a)\ \&\ bmp(b)\ \&\ bmp(c) = bmp(d)\ \&\ bmp(e)\ \&\ bmp(f)$ then (a, b, c) and (d, e, f) belong to the same class, otherwise they belong to different classes ("&" denotes logic "and" operation bit by bit).

Each element of the DSP_Set (a cross-combination equivalent class, notice that it is a set itself) has a corresponding set of destination and source IP prefix pairs. These IP prefix pairs are those of the filters whose 3-tuple destination port, source port and protocol number belongs to the element. The cross-combinations that belong to the same element of *DSP_Set* share the same pointer to a destination and source IP prefix set. The non-collision hash Trie-tree algorithm first finds out the pointer to the destination and source prefix pair set by looking up a non-collision hash table according to the destination port, the source port and the protocol number. Then we perform a two-dimension Trie-tree lookup in the destination and source IP prefix pairs to obtain the final *ClassID*.

4.2 Non-Collision Hash Lookup

When classifying a packet with its header *H*(*dport, sport, proto*) (representing the destination port, the source port and the protocol respectively, we do not consider IP address fields in this step), we first look up three tables using *dport*, *sport* and *proto* as the indices respectively. We use a function of these lookup result *g*(*fd*(*dport*), *fs*(*sport*), *fp*(*proto*)) as the index to perform another lookup. The result is *h*(*g*(*fd*(*dport*), *fs*(*sport*), *fp*(*proto*))), which is the pointer to a set of destination and source IP prefix pairs. We number the D equivalent classes' IDs of *D_Set* as $0, 1, 2, \ldots, D-1$ and define *fd*(*dport*) as the equivalent *ClassID* of *dport*. And it is the same with *fs* and *fp*.

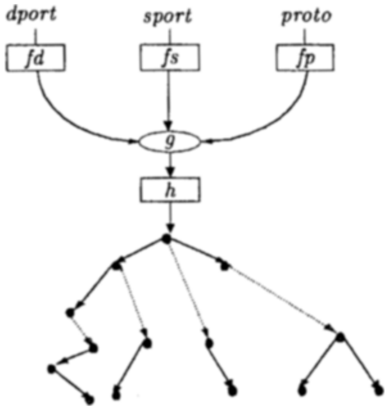

Fig.1. Non-collision hash Trie-tree algorithm.

Fig.1 shows the lookup process stated above. The rectangle represents a lookup table and g is a hash function. The bottom of Fig.1 is the src-dest Trie tree that is used to look up with the source IP address and the destination IP address. We choose $g(d,s,p) = PSd + Ps + p$. According to our definitions of *fd, fs* and *fp*, we have $0 \le d \le D-1$, $0 \le s \le S-1$, $0 \le p \le P-1$. Now we prove g will not cause any collision.

Theorem. *Define* $g(d,s,p) = PSd + Ps + p$ $(d,s,p,D,S,P \in Z)$, $0 \le d \le D-1$, $0 \le s \le S-1$, $0 \le p \le P-1$, *if* $g(d_1,s_1,p_1) = g(d_2,s_2,p_2)$, *then* $d_1 = d_2, s_1 = s_2, p_1 = p_2$.

Proof. Because $PSd_1 + Ps_1 + p_1 = PSd_2 + Ps_2 + p_2$, $p_2 - p_1 = PSd_1 + Ps_1 - PSd_2 - Ps_2 = P[S(d_1 - d_2) + s_1 - s_2]$. Taking the absolute values of both sides, we have $|p_2 - p_1| = P|S(d_1 - d_2) + s_1 - s_2|$. Because $p_1, p_2 \in [0, P-1]$, $|p_2 - p_1| < P$. Since $|S(d_1 - d_2) + s_1 - s_2|$ is an integer, we conclude that $p_2 - p_1 = 0$, i.e., $p_2 = p_1$. For the same reason, we have $s_2 = s_1$, $d_1 = d_2$. □

Now we have proved that g is a non-collision function. The construction of these tables and the two-dimension Trie-tree is completed by reading the filter database during the pre-computation stage. Thus, we can find the pointer to the set of destination-source IP prefix pairs with four memory accesses. The next step is to look up through the two-dimension Trie-tree using destination and source IP addresses in the packet header. Algorithms 1 and 2 are the setup algorithms of these tables.

Algorithm 1. Construction of *fd*(*fs*, *fp*) Table

```
*table_fx table_fx_setup()
{
        /*allocate memory and initialization*/
        $p$ = new_table_fx();
        for ($n = 0; n < 65536, n++$)
        {
                Get bmp($n$);
                eq = search_in_equivalence_class_x_set(bmp($n$));
                if (eq == NULL) /* new bmp */
                {
                        eq = new_equivalence_class_x(bmp($n$));
                        add eq into x_set;
```

Table 2. An Example of Source-Destination IP Pairs

ClassID	Dest-IP	Src-IP
0	0*	10*
1	0*	01*
2	0*	1*
3	00*	1*
4	00*	11*
5	10*	1*
6	*	00*

```
            }
            p → table[n].ID=eq → ID;
        }
        return p;
}
```

Algorithm 2. Construction of h Table

```
*table_h table_h_setup()
{
        indx = 0;
        /*allocate memory and initialization*/
        p = new_table_h(D, S, P);
        for eqd in d_set, eqs in s_set, eqp in p_set
        {
            bmp = eqd → bmp&eqs → bmp&eqp → bmp;
            eq = search_equivalence_class_dsp(bmp);
            if (eq is null) /*new bmp*/
            {
                eq = new_equivalence_class_dsp(bmp);
                add eq into dsp_set;
            }
            p → table[indx++].ID=eq → ID;
        }
}
```

4.3 Lookup in Destination-Source IP Prefix Pairs

In this section, we introduce a simplified Grid-of-Tries lookup algorithm. Extending the Trie-tree data structure from one-dimension to two-dimension, we have the two-dimension Trie-tree. We take the filter database in Table 2 as an example to show this process (assume that the width of IP address in the table is 2).

We first build up a Trie-tree (denoted by Dest-Trie tree) according to the destination IP prefixes. For each node in Dest-Trie tree, if there exists the corresponding destination IP prefix, it points to a source IP prefix Trie-tree (denoted by Src-Trie), otherwise the pointer is null. A Dest-Trie node not only contains the corresponding source IP prefixes but also those of its ancestors in Dest-Trie. In that case, time complexity of lookup in the two-dimension Trie tree is $O(W)$, but since each Dest-Trie node stores both the pair source IP prefixes of its own and those of its ancestors, the space compexity turns out to be $\Theta(N^2)$.

We can get rid of the redundant copies. Every Dest-Trie node only contains the corresponding source IP prefixes in the database. But in this case, in order to find out the final *ClassID* with the least cost, we need to search not only the Src-Trie but also that of its ancestors. Thus the time complexity rises up to $O(W^2)$, although we need less space.

The solution is to introduce a switch pointer. In the process of pre-computation, we direct the null pointer of the Src-Trie node to an Src-Trie node of one of its Dest-Trie ancestors' so that we can proceed further when we go along the longest matching path. In addition, we must make sure that the longer a destination-source prefix pair is, the lower its cost is. Take filters 2, 3 and 4 of Table 2 as an example, filter 2 is shorter than filter 3 in destination-source pair length, and filter 3 is shorter than filter 4. But the fact is that filter 2's cost is lower than filter 3 and filter 3's cost is lower than filter 4's. So they do not accord with our principle of a longer pair with a lower cost. However, we observe that if we remove filters 3 and 4 from the table, our lookup result does not change. That is because a header matching filter 3 and filter 4 will surely match filter 2, and filter 2 has a lower cost. In other words, filter 3 and filter 4 are redundant. There are two ways to deal with the problem. The first one is to guarantee that there is no redundancy at all when building up our filter database; the other is to change both filters 3's and 4's *ClassID* to 2 in the pre-computation stage to guarantee the correctness of the algorithm.

The ultimate two-dimension Trie tree is shown as Fig.2, where the number beside the letter "f" denotes the corresponding *ClassID* (it is also the sequence number of the filter and its cost). Given this figure, we look up the matching filter with the lowest cost for a coming header as follows.

First perform the longest destination IP prefix matching process ending at some node in Dest-Trie. Then go along the 0 or 1 pointer (or if null, a switch pointer) of the corresponding Src-Trie to perform the longest source IP prefix matching according to the header's destination and source IPs. We go as farther as we can, and the *ClassID* of the filter with the lowest cost is the final result we want.

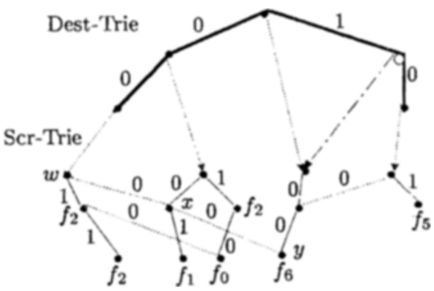

Fig.2. Improved data structure of the 2-dimension Trie-tree.

Algorithm 3 is the construction algorithm of the Src-Dest Trie tree.

Algorithm 3. Construction of Src-Dest Trie Tree

```
void insert_ip_pair(dst_trie_node *root, ip dip, ip sip)
{
          /*insert destination IP address into dst-trie node*/
          dp = dst_trie_insert(root,,dip);
          /*insert source IP address into src-trie node */
          sp = src_trie_insert (&dp → psrc, src_ip);
          insert information of the rule into sp node;
}
/*set up dest_ip nodes*/
dest_trie_node* dest_trie_insert(dest_trie_node *root, ip dest_ip)
{
          if (!dest_ip) return NULL;
          if (strlen (dest_ip)>NMAXIPWIDTH) return NULL;
          if (!(*root))
          {
                    *root = new dst_trie;
                    initialize root;
          }
          set up dest_trie nodes according to string dest_ip;
          continue until we meet 0 or * in dest_ip;
          return the pointer to the last dest_trie node;
}
src_trie_node* src_trie_insert(src_trie_node *root, ip src_ip)
{
          if (!src_ip) return NULL;
          if (strlen (src_ip)>NMAXIPWIDTH) return NULL;
          if (!(*root))
          {
                    *root = new src_trie;
                    initialize root;
          }
          set up src_trie nodes according to string src_ip,
          continue until we meet 0 or * in src_ip;
          return the pointer to the last src_trie node;
}
```

Algorithm 4 is the pre-compute procedure of the src-dest Trie tree.

Algorithm 4. Pre-Computation of Src-Dest Trie Tree

```
void pre_compute(dst_trie_node *root)
{
        if (root == null) return;
        p = pointer to the lowest src_trie of root's ancestors;
        if (root → src_trie == null)
                root → src_trie = p;
        else
        {
                if (p is not null and has less cost)
                        *root → src_trie = *p;
                search root → src_trie, be sure the cost of src_ip is less than its prefix;
        }
        /*search left sub-tree*/
        pre_compute (root → child[0]);
        /*search right sub-tree*/
        pre_compute (root → child[1]);
}
```

The whole working procedure of the non-collision hash Trie-tree algorithm is divided into two steps.

The first step is the pre-computation of four hash tables (*fd, fs, fp* and *h*) and src-dest Trie-tree according to the filter database. The construction algorithms of these tables are shown in Algorithm 1 through Algorithm 4.

After establishing these tables and the Trie tree, the second step is the lookup procedure.

5 Lookup Performance

In the worst case, it takes four serial lookups to obtain the pointer to the two-dimension Trie-tree, i.e., the lookups in tables *fd, fs, fp* and *h*. Lookup through the two-dimension Trie-tree needs to visit $2W$ nodes in the worst case. So the total number of memory accesses is $2W + 4$ in the worst case. What is more, the consumed time is irrelevant to the number of filters. In contrast, even with a hash function without collision, Grid-of-Tries needs $4(1 + 2W)$ memory accesses.

It is a little more complicated as for the space complexity of the non-collision hash algorithm. The numbers of entries of the *fs, fd* and *fp* tables are all 65536 and the number of entries of table h is $D \times S \times P$. Theoretically speaking, the number of table h's entries could be up to $65536 \times 65536 \times 65536$. However, as analyzed above, D, S and P are rather small in normal cases, so we expect that the number of table h's entries is quite small. As for the two-dimension Trie-tree, since a filter needs $2W$ Trie nodes at most and there are N filters altogether, the space needed is about $2NW$. Thus we could estimate that the total space is $Table\_Size{+}2NW$ in the worst case, while Grid-of-Tries for multi-fields also needs about $Hash\_Size{+}2NW$, where $Hash\_Size$ denotes the space for the hash table. In order to gain higher time efficiency, the hash table usually consumes a lot of memory. In our test, the non-collision hash Trie-tree algorithm also shows better performance in space.

It is difficult to analyze the average performance of both time and space. Even worse, little is done in sampling for both filters and IP flow in the real Internet. Because of that, we design a virtual environment to perform a testing. Our concern focuses on the relative performance between the non-collision hash Trie-tree and Grid-of-Tries, so the virtual environment will be sufficient. We make reasonable assumptions about IP flows and filters and generate IP packet flow and filters from a random number generator. For the sake of comparability, we add the same limitations needed by the Grid-of-Tries algorithm. We observe that even our assumptions favor the Grid-of-Tries, the non-collision hash algorithm still shows better performance in both time and space. So we expect the difference will be more obvious in practice. Testing results are shown in Fig.3 and Fig.4.

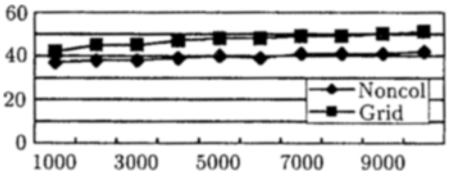

Fig.3. Comparison of time performance between nonconllision Trie-tree (denoted by Noncol) and Grid-of-Tries (denoted by Grid), where x-axis plots the number of filters and y-axis plots the total seconds consumed while processing 10^7 packets.

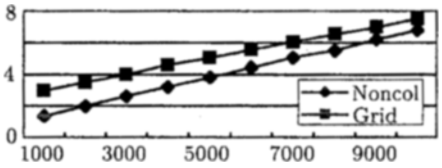

Fig.4. Comparison of space performance between non-collision Trie-tree (denoted by Noncol) and Grid-of-Tries (denoted by Grid), where x-axis plots the number of filters and y-axis plots the maximum memory (MB) consumed.

6 Conclusions and Future Work

In this article, a novel IP classification algorithm is presented based on the Grid-of-Tries algorithm. The new algorithm not only eliminates original limitations in the case of multiple fields but also shows better performance in regard to both time and space. It has better overall performance than many other algorithms.

We have implemented the algorithm in the "High-Performance Security Router", a key project of the High Technology Research and Development Program of China. The "High-Performance Security Router" implements packet filter and IPSec. It may support four Gigabit Ethernet interfaces at most, so traditional classification algorithms cannot keep up with the speed of interface. Our new algorithm meets the performance requirement of IP packet classification in the "High Performance Security Router".

The algorithm we presented can be improved further. In the process of lookup through the two-dimension Trie-tree, our algorithm will go one step according to a bit of the header. If it can look up several bits at a time, the depth of the Trie-tree will reduce greatly and the performance will improve. Future work is to explore the distribution of IP prefixes[11], by which we hope that we can select the depth of Trie-tree and decide which bits to look at when going down the Trie-tree.

References

[1] Weiss W. QoS with differentiated services. *Technical Journal,* 1998, 3(4): 48–62.

[2] Bellovin S, Cheswick W. Network firewalls. *IEEE Communications Magazine,* 1994, 32(9): 50–57.

[3] Jyh-haw Y, Randy C, Richard N W. Interdomain access control with policy routing. In *Proceedings of the IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems,* Oct., 1997, pp.46–52.

[4] Richard Edell, Nick McKeown, Pravin Varaiya. Billing users and pricing for TCP. *IEEE Journal on Selected Areas in Communications,* Sept., 1995, 13(7): 1162–1175.

[5] Xu Ke, Xiong Yong-qiang, Wu Jian-ping. Analysis of broadband IP router architecture. *Journal of Software,* 2000, 11(2): 179–186.

[6] Gupta P, McKeown N. Packet classification on multiple fields. *ACM Computer Communication Review,* 1999, 29(4): 146–160.

[7] Bailey M L, Gopal B, Pagels M A, Peterson L L. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the 1st Symposium on Operating System Design and Implementation.* Usenix Association, 1994, pp.95–104.

[8] Srinivasn V, Varghese G, Suri S *et al.* Fast scalable level four switching. *ACM Computer Communication Review,* 1998, 28(4): 191–205.

[9] Woo T Y C. A modular approach to packet classification: Algorithms and results. In *Proceedings of IEEE Infocom'2000,* 2000.

[10] Stevens W R. UNIX Networking Programming (2nd Edition). Prentice Hall, Inc. 1998, Vol.1.

[11] Merit Inc. IPMA Statistics. http://nic.merit.edu/ipma.