2010 24th IEEE International Conference on Advanced Information Networking and Applications

Grid of Segment Trees for Packet Classification

Yeim-Kuan Chang, Yung-Chieh Lin, and Chen-Yu Lin
Department of Computer Science and Information Engineering,
National Cheng Kung University, Tainan, Taiwan
{ykchang, p7894110, p7696136}@mail.ncku.edu.tw

***Abstract*—*Packet classification problem has received much attention and continued to be an important topic in recent years. In packet classification problem, each incoming packet should be classified into flows according to a set of pre-defined rules. Grid-of-tries* (*GoT*) *is one of the traditional algorithmic schemes for solving 2-dimensional packet classification problem. The advantage of GoT is that it uses the switch pointers to avoid backtracking operation during the search process. However, the primary data structure of GoT is base on binary tries. The traversal of binary tries decreases the performance of GoT due to the heights of binary tries are usually high. In this paper, we propose a scheme called GST* (*Grid of Segment Trees*). *GST modifies the original GoT by replacing the binary tries with segment trees. The heights of segment trees are much shorter than those of binary tries. As a result, the proposed GST can inherit the advantages of GoT and segment trees to achieve better performance. Experiments conducted on three different kinds of rule tables show that our proposed scheme performs better than traditional schemes, such as hierarchical tries and grid-of-tries*.

Keywords-packet classification, segment tree, grid-of-tries

I. INTRODUCTION

Because of the rapid growth of the network, workloads of Internet routers are increased sharply. Nowadays, packet classification has received much attention and continued to be an important topic. Packet classification is an enabling function for network applications, such as quality of service (QoS), security, monitoring, and multimedia communications and it is often a bottleneck in high performance routers. In order to classify a packet into a particular flow, each incoming packet needs to be determined the output port it should be sent to and the action it should be taken. Unlike the IP lookup problem, packet classifiers in routers need to compare multiple header fields of each incoming packet with a set of rules to determine which action should be applied, for example, acceptance or denial. Grid-of-tries [7] is a traditional algorithm for solving 2-dimensional (2D) packet classification problem. The primary data structure of grid-of-tries is binary tries. Binary tries might have a maximum height $h = W$, where W is the length of address. As a result, the time complexity of operations on binary tries is $O(W)$. A segment tree is a data structure that stores a set R of n ranges; it allows querying which of stored range contain a given value efficiently. Based on the dynamic segment tree proposed in [3], time complexity of operations was reduce to $O(\log n)$.

In this paper, we proposed a packet classification scheme called Grid of Segment Tree (GST). GST is a hierarchical scheme based on segment tree by replacing the binary trie in grid-of-tries with segment tree. Furthermore, we employ the concept of switch pointers in grid-of-tries to speed up the search process. The rest of the paper is organized as follows. Section II formally describes the packet classification problem. In section III, we present an overview of previous works and segment trees. Section IV gives a detailed description of the proposed scheme. Section V presents the experimental results in terms of search speed, average tree nodes accesses, and memory requirement. Finally, our conclusions are stated in Section VI.

II. PROBLEM STATEMENT

In the general packet classification (PC) problem, query packets are classified according to a rule table, which define the patterns that are matched against to the query packet. Each rule R contains t components with a cost value and an attached action. Suppose C_i is the ith component of rule R, $R = \{ C_i \mid i = 1 \text{ to } t\}$, where $C_i = [L_i, U_i]$ is a range from L_i to U_i. For example, the rule of the layer-four switching contains five components: the source address, destination address, source port, destination port, and protocol number. Table I shows an example of a 5D rule table. A packet P is said to match R, if $\forall$ i, the ith header field of P satisfy the constraints of C_i. The goal of PC problem is to determine the least cost rule (sometimes called best matching rule) or multiple rules that matches the query packet. Consider a query through Table I with 5-tuple = (00101, 10111,

Table I. An example rule table consisting of eight 5D rules, prefix length $W = 5$.

Rule	Src. Addr.	Dest. Addr.	Src. Port	Dest. Port	Protocol	Cost	Action
R1	0*	10*	0:65535	80:80	TCP	3	Accept
R2	00*	11*	80:80	8080:8080	UDP	1	Accept
R3	011*	00*	0:65535	80:80	TCP	2	Accept
R4	10*	1*	0:65535	0:65535	*	4	Deny
R5	10*	00*	0:65535	0:65535	TCP	5	Accept
R6	0*	01*	17:17	17:17	UDP	8	Accept
R7	00*	10*	0:65535	0:65535	TCP	6	Deny
R8	0*	*	0:65535	0:65535	*	7	Deny

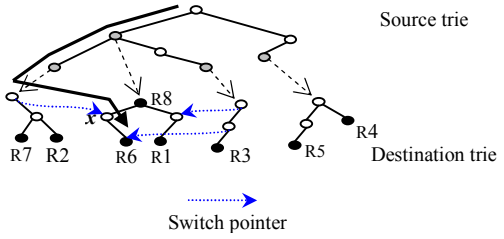

Figure 1. Grid-of-tries build according to Table I.

1550-445X/10 $26.00 © 2010 IEEE
DOI 10.1109/AINA.2010.38

Table II. Prefix table with five prefixes, $W = 6$.

Name	Prefix	Range	Minus-1 endpoint scheme	
			Start	finish
P1	0*	[0, 31]	-	31
P2	01000*	[16, 17]	15	17
P3	011*	[24, 31]	23	31
P4	100*	[32, 39]	31	39
P5	1101*	[52, 55]	51	55

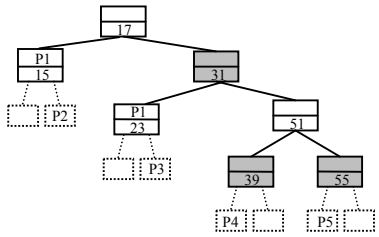

Figure 2. A possible DST built according to Table II.

80, 80, TCP). There are three rules match the query packet as follows: R1, R7 and R8. The classification result is R1 with least cost against to R7 and R8.

III. RELATED WORKS

In network routers, packet classifiers match the header of each incoming packet against to a set of predefined rules. Over the past few years a considerable number of studies have been made on 2D or 5D packet classification. In [7], authors proposed three trie-based algorithms for solving 2D packet classification. The Hierarchical tries is a simple extension of the 1-dimensional binary trie, and is constructed recursively. This algorithm stores each rule exactly once, and the storage complexity of hierarchical trie for N rules is $O(NdW)$. The major drawback of hierarchical trie is the need of backtrack while performing the search; hence, the time complexity of query a d-dimensional hierarchical trie is $O(W^d)$. A set pruning tries is similar to hierarchical tries with reduced query time obtained by replicating rules to eliminate backtracking. Although the query time complexity is reduced to $O(dW)$, however, the memory blowup problem cause the storage complexity increased to $O(N^dW)$. The grid-of-tries is designed to solve the shortness of hierarchical tries and set pruning tries. It reduces the storages space by allocating a rule to exactly one node, and achieves $O(W)$ query time complexity by using pre-computation and switch pointers. Three algorithms described above are design to handle 2D rules, such as source-destination address pairs. Although they can be extend to deal with other fields such as port ranges, however, it is an inefficient work due to the range needs to be converted into prefixes, and a W-bit range might be converted to $2W–2$ prefixes at most. A new multidimensional scheme based on the binary range and prefix searches with fast update is proposed in [4]. In [5], a hierarchical scheme called Fat Inverted Segment tree (FIS) was proposed. Two survey papers in [6] and [9] give a complete overview for a variety of software and hardware schemes. The dynamic segment tree (DST) proposed in [3] uses all of the distinct endpoints of ranges as the keys based on a new endpoint scheme. Although the segment trees are designed for ranges originally, we can treat a prefix as a limited range.

A. Grid-of-tries

The key ideas of this algorithm are use *pre-computation* and *switch pointers* to speed up search in a later source trie based on the search in an earlier source trie. Figure 1 shows the grid-of-tries build according to two address fields in Table I. The switch pointers are shown using dotted lines between destination tries. This distinguishes the switch pointers from the dimensional pointers using dashed lines that connect the source trie nodes to corresponding destination trie.

To understand the role of switch pointers, consider matching a packet with source address 001 and destination address 010 in Figure 1. The search in the source trie gives $S = 00$ as longest match. So we start our search in the associated destination trie. However, the search immediately fails, since the first bit of the destination address is 0. In hierarchical trie [7] without the help of switch pointers, we would backtrack along the source trie and restart the search in the destination trie of all the ancestors of S. In grid-of-tries, however, we use switch pointer to directly jump to the node x in destination trie containing R1, R6 and R8. Therefore, we can find a matching rule R6. This in turn improves the search complexity from $O(W^2)$ to $O(W)$. The bold line in Figure 1 shows the traversing path of this query example. By using the switch pointers, we could find a matching rule ultimately. However, the matching rule might not be the least cost rule due to we possibly miss some rules with lower cost which also match the query packet. For instance, the cost of R8 was smaller than R6 and also matches the above query; hence, the result of query is incorrect. Grid-of-tries solve this problem by using pre-computation.

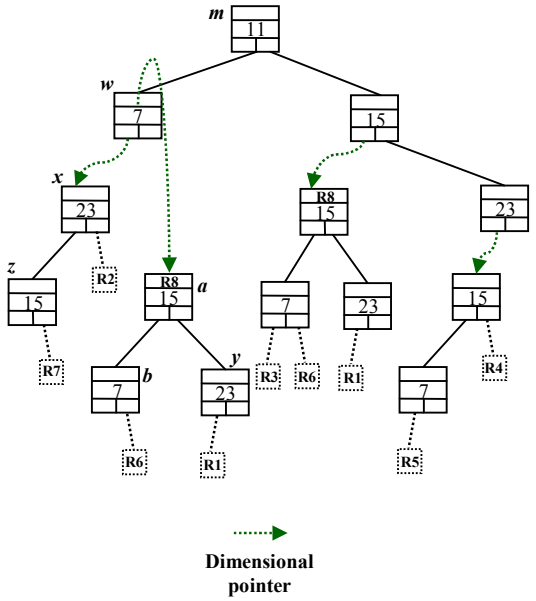

Figure 3. A possible 2-dimensional DST built according to Table I.

```
Algorithm GST_Construct_Sw_Pointers([L, U], root_T2)
Begin
1     x = root_T2;
2     while (1){
3       if (U > key(x)){
4         if (L < key(x))
5           SWITCH_POINTER points to x; break;
6         else{
7           if (right(x) ≠ null) x = right(x);
8           else{
9             if (L □ key(x))
10              SWITCH_POINTER points to x; break;
11            else
12              SWITCH_POINTER points to x; break;
13          }
14      }
15      else if (U < key(x)){
16        if (left(x) ≠ null) x = left(x);
17        else SWITCH_POINTER points to x; break;
18      }
19      else if (U = key(x)){
20        if (left(x) = null)
21          SWITCH_POINTER points to x; break;
22        else{ y = left(x);
23          while (1){
24            if (L □ key(y))
25              SWITCH_POINTER points to y; break;
26            else
27              if (right(y) ≠ null) y = right(y);
28              else SWITCH_POINTER points to y; break;
29          } // END of 2nd while loop
30    } // END of 1st while loop
End
```

Figure 4. The switch pointer construction algorithm.

B. *Dynamic Segment Tree*

In [3], authors propose the dynamic segment tree (DST) to solve the IP lookup problem by treating the prefixes as ranges. The skeleton of DST is a height balanced binary search tree that is built from the distinct endpoints of ranges based on a novel *minus-1 endpoint scheme*, which generates fewer endpoints than the traditional endpoint scheme. The elementary intervals (EIs) [1,3], that are constructed from endpoints of the range set R, correspond to the leaf nodes of the DST. As the statement in [3], the interval covered by an internal node v is the union of EIs corresponding to the leaf nodes in the subtree rooted at v. Each node v is associated with a subset of R (called canonical set). The DST can efficiently access and update the ranges stored in the canonical set of a DST node. The set of matching ranges for the given address d can be obtained by traversing the DST from the root to the leaf node that corresponds to the EI containing d. Figure 2 shows a possible DST built from the prefixes in Table II. The query time complexity of DST is $O(\log N)$, where N represents the number of arbitrary ranges. Traditionally, the segment tree is constructed by pre-computing the elementary intervals and then using a bottom-up approach to build the data structure, this makes the segment tree becomes a static data structure, hence, the segment tree does not fit to dynamic routing tables. However, the DST proposed in [3] can dynamically insert/delete the ranges into/from the segment tree.

IV. PROPOSED SCHEME

By using the DST [3] as the basic data structure, the skeleton of the proposed grid of segment trees (GST) is a 2-dimensional DST and the DST were implemented as height balanced binary search tree such as red-black tree [2]. The details of how to build a DST are carefully described in [3]. This section will be focus on the construction between dimensions in GST. Figure 3 shows a possible 2-dimensional DST built according to two address fields in Table I.

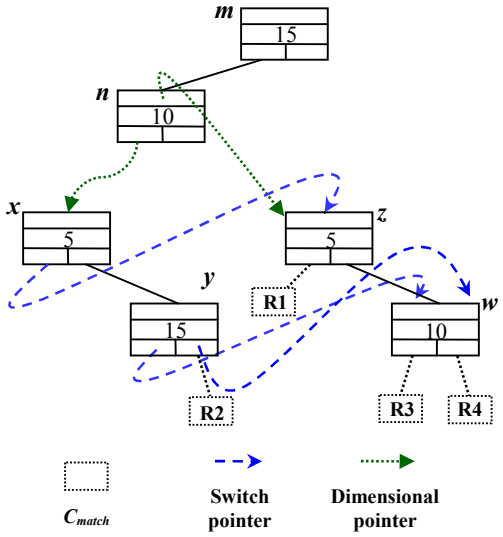

Figure 5. An example to illustrate the switch pointers.

A. *Node structure*

As the node of red-black tree, each node in GST contains the following fields: *key* (endpoint of range), *color* (red or black), *left*, *right* and *parent* (pointers to the left and right children and parent). In addition to these original fields, each node also contains six new fields: *left_switch*, *right_switch* (left and right switch pointer), P_{left}, P_{right} and P_{center} (pointers to DST in the next dimension, called dimensional pointers). The node in segment tree can determine two intervals, which are the left interval ($\text{Int}_{left}()$) and right interval ($\text{Int}_{right}()$), by its *key*. Moreover, the interval covered by the node itself is the union of left and right intervals, called center interval (i.e., $\text{Int}_{center}() = \text{Int}_{left}() \cup \text{Int}_{right}()$). Each interval of a node is correspond to a dimensional pointer and also associated with a canonical set C (or called Cset). In our scheme, we will store the rule R into the Cset in the last dimension of GST.

B. *Insertion in GST*

To insert a rule $R = \{\text{SA} = [s_1, f_1], \text{DA} = [s_2, f_2]\}$ into GST requires the following steps:

1. If s_1 is not zero, insert $s_1 - 1$ as a new key in 1st-dimensional DST; if f_1 is not $2^W - 1$, insert f_1 as a new key in 1st-dimensional DST.

2. Find proper dimensional pointers $P_{left}(v)$, $P_{right}(v)$ or $P_{center}(v)$ to insert DA into 2nd-dimensional DST for some node v, where $\text{Int}_{left}(v) \cup \text{Int}_{right}(v) \cup \text{Int}_{center}(v) = [s_1, f_1]$.

3. Same as step 1 with s_2 and f_2 to build the 2nd-dimensional DST.

```
Algorithm GST_Query_First_Dimension(root, s, d)
Begin
1       z = root;
2       while (1){
3         if (P_center(z) ≠ null)
4           Next = P_center(z);
5         if (s □ key(z)){
6           if (left(z) = null){
7             if (P_left(z) ≠ null)
8               Next = P_left(z);
9             break;
10          }
11          else
12            z = left(z);
13        }
14        else{
15          if (right(z) = null){
16            if (P_right(z) ≠ null)
17              Next = P_right(z);
18            break;
19          }
20          else
21            z = right(z);
22        }
23      } // END of while loop;
24      GST_Query_Second_Dimension(Next, d);
End
```

Figure 6. The GST query algorithm for 1st-dimension.

4. Store R into proper canonical sets in the 2nd-dimensional DST.

5. Construct the switch pointers if necessary according to the *GST_Construct_Sw_Pointers* algorithm (Figure 4).

1) *Inserting a field in GST*

To insert one of the rule fields into GST is executed in step 1 or 3. The detail of insert an endpoint into DST was described in [3]. We insert endpoints follow the minus-1 endpoints scheme [3]. If the new endpoint already exists in the DST which we are going to insert, no new node is created.

2) *Finding the dimensional pointers to next dimension*

After inserting the endpoints of SA, we need to find proper intervals that contain the range of SA (i.e., from s_1 to f_1) in step 2, because each interval is correspond to a dimensional pointer which point to another DST in next dimension. We can follow these dimensional pointers to insert DA in step 3.

3) *Storing the rule into GST*

After two endpoints of DA are inserted in step 3, the rule R can then be stored in the canonical sets of some proper nodes in the 2nd-dimensional DST.

C. Constructing the switch pointer

The purpose of switch pointers is to avoid backtracking during search. There are two possible switch pointers in the node, each of them corresponds to an interval, and that is, the *left_switch* and *right_switch* corresponds to $\text{Int}_{left}()$ and $\text{Int}_{right}()$, respectively. Consider matching a packet with (source address, destination address) = (5, 10) in Figure 5. Assume that the switch pointers do not exist. The search in the DST rooted at node x is fail and we need to backtrack to another DST rooted at z. The query path without switch pointers is *m-n-x-y-n-z-w*. However, while the switch pointers have been constructed, query path becomes *m-n-x-y-w* and R3 will be found as the match rule without backtracking. The switch pointer was connected from one DST to another, say from T_1 to T_2. Assume we would like to construct the switch pointer of a range [L, U] in T_1 (i.e., the corresponding interval), for example, the range of *left_switch*(y) in Figure 5 is [6, 15]. The aim is to find an interval of a node, g, in T_2, where $\text{Int}_{center}(g)$ cover the range [L, U] and both $\text{Int}_{center}(right(g))$ and $\text{Int}_{center}(left(g))$ do not cover [L, U]. Figure 4 shows the algorithm of constructing the switch pointers. The purpose of switch pointers is to reduce the number of traverd nodes during the search process as many as possible.

```
// C_match stores rules that matches (s, d). C_match is initially empty.
Algorithm GST_Query_Second_Dimension(x, d)
Begin
1       while (x ≠ null){
2         C_match = C_match ⋃ C_center(x);
3         if (d ≦ key(x)){
4           if (left(x) ≠ null)
5             x = left(x);
6           else{
7             C_match = C_match ⋃ C_left(x);
8             if (left_switch(x) ≠ null)
9               x = left_switch(x);
10            else
11              break;
12          }
13        }
14        else{
15          if (right(x) ≠ null)
16            x = right(x);
17          else{
18            C_match = C_match ⋃ C_right(x);
19            if (right_switch(x) ≠ null)
20              x = right_switch(x);
21            else
22              break;
23          }
24        }
25      } // END of while loop
26      if (C_match ≠ Ø)
27        Report all rules stored in C_match.
28      else
29        No rules match the query packet.
End
```

Figure 7. The GST query algorithm for 2nd-dimension.

D. Querying the GST

The goal of packet classification is to determine the input packet belongs to which rules in the rule table. In a GST, a query packet p with two addresses (s, d) may be found by traversing a path from the root in the 1st-dimension toward a leaf in the 2nd-dimension. Figure 6 and 7 shows the GST search algorithms in 1st and 2nd dimension separately. The *while loop* in Figure 6 does the DST traversal according to query key s, the traverse of 1st-dimension will finally leads to a dimensional pointer which points to another DST in 2nd-dimension. The search in the 2nd-dimension was more complicated, for some node v, while *left*(v) = null, the search still need to examine whether the *left_switch*(v) is null or not. Consider matching a packet with (s, d) = (5, 20) in Figure 3. We traverse the DST in 1st dimension with s = 5 and leads to a dimensional pointer $P_{left}(w)$ which covered the interval [0, 7]. Then we start the search for d = 20 in the DST pointed by $P_{left}(w)$, and then following the *right_switch*(z) = y to search another DST rooted at node a. The query path of this packet is *m-w-x-z-y*. The

Table III. Performance of ACL.

# of traversed nodes in average			
Size	HT	GoT	GST
1k	33	14	6
2k	28	13	6
3k	26	14	7
5k	34	15	8
8k	60	20	10
10k	60	19	11

Table IV. Performance of FW.

# of traversed nodes in average			
Size	HT	GoT	GST
1k	46	16	9
2k	63	17	11
3k	95	20	11
5k	124	29	12
8k	50	24	13
10k	50	18	13

Table V. Performance of IPC.

# of traversed nodes in average			
Size	HT	GoT	GST
1k	32	15	8
2k	33	17	8
3k	35	17	8
5k	44	19	8
8k	51	21	9
10k	49	19	9

matching rule of the query packet is R1, R7 and R8. Note that R1 and R7 are founded during the traversal of GST and R8 is founded via the precomputation technique that is similar to [10].

V. EXPERIMENTAL RESULTS

A. Environment and test data

We programmed our GST in C code, and all experiments were run on a 1.86GHz Core-2 PC with 1GB main memory. The compilation environment is gcc-3.4.2 with optimization level O2. Our experiments were conducted by using three different kinds of IPv4 rule tables. The rule tables of various sizes are generated by using ClassBench [8] with parameters "acl1_seed", "fw2_seed", "ipc1_seed", wher ACL, FW, and IPC stand for Access Control List, Firewall, and IP Chain, respectively. All rules are 5-tuples that consist of 32-bit source/destination IP addresses (represented as prefixes), 16-bit source/destination port numbers (represented as ranges), and 8-bit transport layer protocol (represented as discrete numbers).

B. Performance comparison

In this section, we present the experimental results of the proposed GST and other existing schemes. First, we compare the proposed GST with hierarchical tries (HT) [7] and grid-of-tries (GoT) [7] in terms of number of traversed nodes during the search process. For each of these three schemes, we first build a hierarchical structure that consists of two dimensional tries (1'st dimensional trie and 2'nd dimensional trie), where 1'st and 2'nd dimensional tries are built according to the source address prefix field and destination prefix field, respectively.

Table VI. Performance of ACL in terms of classification speed and memory requirement.

Search Time (clock cycles)			Memory Requirement (MB)	
Size	GoT	GST	GoT	GST
1k	1,815	509	0.140	0.071
2k	1,666	491	0.207	0.145
3k	2,062	665	0.349	0.212
5k	2,142	688	0.584	0.320
8k	2,132	757	1.817	0.851
10k	2,126	646	3.038	1.584

Table VII. Performance of FW in terms of classification speed and memory requirement.

Search Time (clock cycles)			Memory Requirement (MB)	
Size	GoT	GST	GoT	GST
1k	1,043	533	0.268	0.150
2k	1,209	875	0.686	0.345
3k	1,304	1,015	1.000	0.513
5k	1,446	1,285	1.611	0.885
8k	1,461	1,336	2.530	1.601
10k	1,633	1,403	3.079	1.906

Table VIII. Performance of IPC in terms of classification speed and memory requirement.

Search Time (clock cycles)			Memory Requirement (MB)	
Size	GoT	GST	GoT	GST
1k	1,976	704	0.294	0.141
2k	1,915	865	0.508	0.242
3k	2,010	957	0.658	0.359
5k	2,178	1,095	1.030	0.704
8k	2,062	1,453	1.672	1.284
10k	1,974	1,293	2.169	1.776

For the remaining three fields of port ranges and protocol numbers, we store them linearly in the 2'nd dimensional tries like [10]. Table III (Table VI), Table IV (Table VII) and Table V (Table VIII) shows the performance on ACL, FW, and IPC tables, respectively. As we can see, the average traversed nodes of hierarchical tries are quite huge because the backtracking was needed. GoT can reduce the average traversed nodes to less than 30 with the help of switch pointers. Our proposed GST decreases the tree height to $O(\log N)$ by implementing the segment tree as a height balanced search tree, where N is the number of rule in table. The average traversed nodes of our scheme are much smaller than GoT and HT. Moreover, since the number of nodes in GoT is much more that that in GST, GST consumes less memory than GoT.

VI. CONCLUSION

We have presented a dynamic segment tree (DST)-based hierarchical structure called GST (Grid of Segment tree) which can solve the multidimensional packet classification problem efficiently. GST replaces the binary tries structure in Grid-of-tries [7] with dynamic segment trees [3] to improve the shortness of Grid-of-tries. Hence, our proposed scheme combines the advantages of dynamic segment trees [3] and Grid-of-tries [7]. The experiments using the rule sets generated

from ClassBench [8] showed that the GST acheive a better performance than Hierarchical trie [7] and Grid-of-tries [7].

REFERENCES

[1] M.D. Berg, M.V. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.

[2] T. Cormen, C. Lieserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd edition MIT Press, 2001.

[3] Yeim-Kuan Chang and Yung-Chieh Lin, "Dynamic Segment Trees for Ranges and Prefixes", *IEEE Transactions on Computers*, vol. 56, no. 6, pages. 769-784, June 2007.

[4] Yeim-Kuan Chang, "Efficient Multidimensional Packet Classification with Fast Updates", *IEEE Transactions on Computers*, vol. 58, no. 4, pages. 463-479, April 2009.

[5] A. Feldman and S. Muthukrishnan, "Tradeoffs for Packet Classification", *Proceeding of IEEE INFOCOM*, vol. 3, pages. 1193-1202, March 2000.

[6] P. Gupta and N. McKeown, "Algorithms for Packet Classification", *IEEE Network Special Issue*, vol. 15, no. 2, pages. 24-32. March/April 2001.

[7] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching", *ACM SIGCOMM Computer Communication Review*, vol. 28, pages.191-202, October 1998.

[8] David E. Taylor and Jonathan S. Turner, "ClassBench: A Packet Classification Benchmark", *IEEE/ACM Transactions on Networking*, vol.15, no. 3, pages. 499-511, June 2007.

[9] David E. Taylor, "Survey and Taxonomy of Packet Classification Techniques", ACM Computing Surveys, vol. 37, no. 3, pages. 238-275, September 2005.

[10] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an alternative to CAMs?", *Proceeeding of IEEE INFOCOM*, vol. 1, pages. 53-63, March 2003.