Fast Packet Classification using Group Bit Vector

Tong Liu[1,2], Huawei Li[1], Xiaowei Li[1,2], Yinhe Han[1]
[1]Advanced Test Technology Lab., Key Lab. of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2]Graduate University of Chinese Academy of Sciences, Beijing, China
E-mail: {liutong, lihuawei, lxw, yinhes}@ict.ac.cn

Abstract—**Packet classification is important in fulfilling the requirements of new services such as policy-based routing in next generation networks. In this paper, we propose a novel bit vector based two-dimensional packet classification algorithm called Group Bit Vector. The key feature of the proposed algorithm is its ability to set the length of the bit vector to perform AND operation (the operation bit vector) regardless of the size of rule databases by effectively aggregating the rules and constructing particular groups of a limited number of members. Moreover, the proposed algorithm avoids the problem of false matches, which may be brought by rules aggregation. Thus, the length of the operation bit vector can be adjusted to be suitable to the width of memories to improve the performance. Experimental results demonstrate the flexibility and effectiveness of the proposed algorithm.**

Keywords-packet classification; bit vector

I. INTRODUCTION

Deployment of next generation networks services lies on the ability of Internet infrastructure to provide flow identification at physical link speeds. A packet classifier can compare header fields of every incoming packet against a database of rules to identify a flow. The resulting flow identifier is used to apply policy-based routing, security policies, application processing, and quality-of-service guarantees to packets belonging to the specified flow. A k-dimensional rule has k header fields, each of which is a variable length prefix, a range or an explicit value. The most common header fields are the IP source and destination address, the protocol type, the port numbers of source and destination applications, and protocol sets. A packet is said to match a particular rule if the packet header satisfies the rule for all fields. In order to decide the best matched rule, each rule is usually assigned a cost to define its priority among the matched rules. The least-cost matched rule with an associative action will be used to process the arriving packets.

In this paper, we present a new bit vector based two-dimensional packet classification algorithm named Group Bit Vector (GBV). Unlike other bit vector based algorithms such as BV [1] and Aggregate Bit Vector (ABV) [2], the GBV algorithm proposes a new scheme to set the length of the bit vector to perform AND operation regardless of the size of rule databases by aggregating the rules with the same field and constructing particular groups of a limited number of members. Moreover, the ability of setting the length of the bit vector makes it possible to adjust the length of the bit vector to the width of memories for meeting the requirements of fast packet classification.

The rest of the paper is organized as follows. The related work is introduced in Section 2. Section 3 describes the main idea of the proposed algorithm. Section 4 analyzes and compares the performance among BV, ABV and the proposed algorithm. Finally, Section 5 states our conclusion.

II. RELATED WORK

In the past few years, researchers have presented many algorithms for packet classification. There are many methods to categorize these algorithms. A simple and useful categorization is proposed in [3]. In [3], the related studies on packet classification algorithm can be categorized three classes: conversion into single-field search, dependent field search and independent field search. The algorithms of the first class include tuple space based algorithms and TCAM based algorithms. Tuple space based algorithms such as [4] use a hash table for each combination of prefix lengths (tuple). In order to get the best matched rule, tuple space based algorithms should search each tuple or some selected tuples, which makes their performance unstable and worse with the increase of tuples. TCAM based algorithms such as [5] can get excellent performance by using Ternary Content Addressable Memories (TCAMs). However, TCAMs have lower storage density and consume more power than other ordinary memories such as SRAM. So TCAM based solutions are still expensive for large size databases. The second ones such as Grid-of-Trees [6], FIS [7], HiCuts [8] and HyperCuts [9] commonly contain hierarchical tries, which are constructed with the search trees of each field. Therefore, the results of previously searched fields affect the way subsequent fields are searched. Thus, the hierarchical tries become more and more complex to get fast search speed. The third ones such as BV [1], ABV [2], RFC [10], P2C [3], DCFL [11] get the best matched rule by combining the whole results of each field search.

With the development of multiprocessors and hardware technology, the algorithms based on independent field are promising. These algorithms can be performed in parallel on each field to get fast lookup speed. Therefore, they can be carried out in multiprocessors or implemented in hardware. Among them, bit vector based algorithms are one of the simplest algorithms to implement. The Lucent bit vector (BV) algorithm [1] is a famous bit vector based algorithm. It projects every rule onto each field and uses a bit vector (bv) to maintain

This paper was supported by the National Basic Research Program of China (973) under Grant No.2005CB321605.

the positional information in a rule database. The matched rule can be gotten after AND operation is performed on all matched bvs. Naturally, the length of the bv would increase in proportion to the number of rules in the rule database. The frequency of memory access to the bit vectors is the main performance factor of bit vector based algorithms. Thus, the length of the bit vector will affect the performance with the increase size of rule databases. Baboescu et al. [2] introduced a new bit vector called aggregate bit vector (abv) and proposed the aggregate bit vector (ABV) algorithm to enhance the BV algorithm. The ABV algorithm reduces the length of the bit vector by rules aggregation. The length of the abv is defined as $\lceil N/A \rceil$, where N is the number of rules and A is the aggregate size of the abv. Rule aggregation makes each bit of the abv maps several rules, thus, it might result in a new so-called false match phenomenon. In the worst case, extra memory accesses for abvs to solve false matches can cause the performance of the ABV algorithm even worse than that of the BV algorithm.

III. GROUP BIT VECTOR ALGORITHM

In this section, we describe the proposed algorithm named Group Bit Vector (GBV) for two-dimensional packet classification. We only discuss the rules whose header field is a variable length prefix because a range or an explicit value can be transformed into a variable length prefix. For the sake of clarity, the operation bit vector denotes the bit vector to perform AND operation in bit vector based algorithms. The algorithm is inspired by two observations from the BV and ABV algorithms.

1. In the BV algorithm, the length of the operation bit vector equals to the number of rules. It is too large for large size rule databases. In fact, the valid bits of the operation bit vector that may represent the matched rules are very sparse.
2. In the ABV algorithm, rules aggregation can reduce the length of the operation bit vector. However, it often brings false matches, which may degrade its performance.

Therefore, our algorithm adds two new ideas. One is to create rule sets. Each rule set includes and represents all the rules with the same feature in a rule database, which can decrease the size of the rule database. The other is to construct particular groups with the rule sets. Each group has a limited number of rule sets, which is useful to construct the operation bit vector of a fixed length. In addition, each group can be easily found if it is relative to the matched rule. Then, searching the matched rule becomes searching the group relative to the matched rule.

Consequently, the algorithm consists of three steps: 1) aggregate some rules with the same selected header field as a rule set, 2) gather some rule sets as a group called parameter group (PG), 3) construct group bit vectors (gbv) with the parameter groups.

In the first step, we select one of the two header fields as the selected field. All the rules are divided into different rule sets, each of which contains the rules with the same prefix in the selected field. The selected header field and the other header field are called the selected field and the other field of the rule set, respectively. Apparently, the selected field of a rule set contains a unique prefix (also called the common prefix) and the other field of a rule set may contain one or more prefixes. We use a simple two-dimensional rule (source prefix, destination prefix) database shown in Table I as an example and choose source prefix as the selected field. In Table II, eight rule sets, each of which has a unique prefix in its selected field and one or more prefixes in its other field, represent sixteen rules.

The other field of a rule set is said to be matched by the corresponding header field of a packet, if it has a prefix that can be matched by the packet's corresponding header field. Then the following properties of the rule set are satisfied according to the definition of the rule set.

Property 1: For two-dimensional rule databases, the selected field of a rule set has a unique prefix.

Property 2: For two-dimensional rule databases, if both the selected field and the other field of a rule set can be matched by a packet's corresponding header fields, the rule set surely includes the matched rule of the packet.

If a rule set includes the matched rule of a packet, we call that the packet matches the rule set. When a packet matches a rule set, we can use the other field of the matched rule set as the index to get the resulting rule. For example, the packet (1111, 0000) can match two rule sets, i.e. S3 and S5, in Table II. Thus, S3 and S5 can both include a matched rule. Using the other field of the matched rule set as the index, (00) for S3 and (00) for S5, we can get the matched rules, R15 and R5.

In the second step, we construct groups named parameter groups, each of which contains a limited number of rule sets. The members of a parameter group are selected from a tree constructed with the prefixes in the selected field.

TABLE I. TWO-DIMENSIONAL RULE DATABASE WITH 16 RULES

Rule	Source Prefix	Destination Prefix		Rule	Source Prefix	Destination Prefix
R1	000*	11*		R9	10*	110*
R2	0*	0000*		R10	011*	10*
R3	1*	1111*		R11	000*	0110*
R4	01*	010*		R12	011*	0*
R5	111*	00*		R13	000*	010*
R6	0*	110*		R14	011*	010*
R7	000*	1111*		R15	1*	00*
R8	10*	1110*		R16	*	00*

TABLE II. RULE SETS WHEN SOURCE PREFIX AS THE SELECTED FIELD

Rule Set	Selected Field	Other Field	Included Rules
S1	000*	11*, 1111*, 0110*, 010*	R1,R7,R11,R13
S2	0*	0000*, 110*	R2,R6
S3	1*	1111*, 00*	R3,R15
S4	01*	010*	R4
S5	111*	00*	R5
S6	10*	1110*, 110*	R8,R9
S7	011*	10*, 0*, 010*	R10,R12,R14
S8	*	00*	R16

The procedure is as follows:

1) Use the data in the selected field of all the rule sets to construct a binary tree. Each node of the tree has two kinds of pointer: one named downward pointer pointing to its child nodes if it has any child nodes, the other named upward pointer pointing to its parent node if it has a parent node. Each node of the tree represents a prefix with its position in the tree and contains one rule set at most according to Property 1[1].
2) Number the leaf nodes.
3) Set a parameter named group parameter that is the multiple of the maximum length of a prefix. For example, the group parameter is the multiple of 32 for IPv4 or 128 for IPv6.
4) Traverse a branch of the tree using upward pointers from a leaf node to the root node. The traversing paths are said to be consecutive if the no. of their starting nodes, i.e. leaf nodes, are consecutive.
5) Group all the different rule sets of the nodes that are in consecutive traversing paths and make sure the number of the rule sets in the group is close to the group parameter.
6) Repeat 5) until all the rule sets in the leaf nodes are grouped.

The detailed algorithm for parameter group division is shown in **Appendix** 1.

For simplicity, we assume the maximum length of prefix is four and assign four to the group parameter. Moreover, the leaf nodes are numbered from left to right. Fig. 1 shows the binary tree constructed with the data in Table II.

For the example in Fig. 1, three rule sets are in the traversing path starting from node 1 and four rule sets in the traversing path starting from node 2. Since the total number of the different rule sets of the nodes in the consecutive traverse paths starting from node 1 and node 2 is 5, larger than 4. They cannot be grouped as one parameter group. Similarly, the different rule sets in the traversing paths starting from node 2 and node 3 cannot be grouped. Therefore, the rule sets in the traversing paths starting node 1 and node 2 constitute respective parameter groups. The different rule sets in the traversing paths starting from node 3 and node 4 can be grouped as one parameter group because the total number of them is not larger than 4. Fig. 1 shows the traversing paths and Table III presents the result of the parameter groups division.

After the second step, several parameter groups are created. Then the last step is to construct the group bit vector (gbv) with the parameter group. Each parameter group has a corresponding gbv, and each bit position of a gbv maps to a corresponding rule set in one parameter group. Because the number of rule sets in a parameter group is not larger than the group parameter, the length of a gbv can be set equal to the group parameter. If the number of the rule sets in the parameter group is less than the group parameter, the remainder bits of the gbv can be defined as invalid bits. For example, the PG1 in Table III corresponds to a 4-bit long gbv. The first three bits correspond to S1, S2 and S8, respectively. The fourth bit is defined as an invalid bit. Then similar to other bit vector based algorithms, the gbvs are added into the search tree in each field. For the search tree of the other field, the gbv of a parameter group is set according to the prefixes of the other field included in the rule sets of the parameter group, as shown in Table II. Fig. 2 and Fig. 3 show their search trees constructed with the data in Table II and Table III, recoding the gbvs of PG1, PG2, and PG3 in each node.

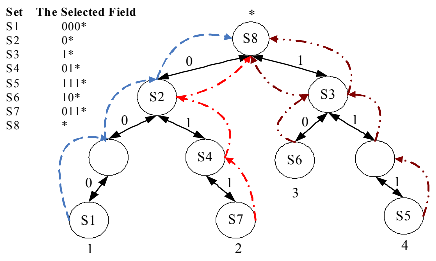
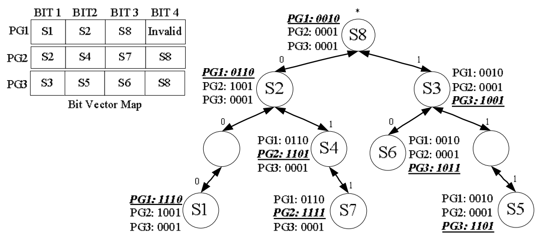


Figure 1. The selected field tree (The number below the leaf node is the no. of the leaf node. Two arrows in the solid lines represent the upward and downward pointers. The dashed lines with arrows represent the traversing paths. The rule sets of the node in the traversing paths with the same pattern belong to the same parameter groups.)

TABLE III. PARAMETER GROUPS

Parameter Group	Included Rule Sets
PG1	S1, S2, S8
PG2	S2, S4, S7, S8
PG3	S3, S5, S6, S8

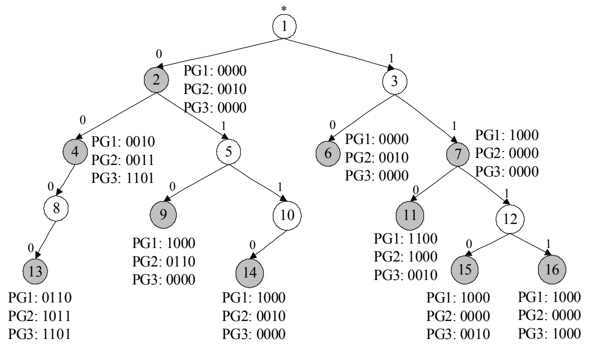


	BIT 1	BIT2	BIT 3	BIT 4
PG1	S1	S2	S8	Invalid
PG2	S2	S4	S7	S8
PG3	S3	S5	S6	S8

Bit Vector Map

*
PG1: 0010 PG2: 0001 PG3: 0001 S8
PG1: 0110 PG2: 1001 PG3: 0001 S2
S3 PG1: 0010 PG2: 0001 ***PG3: 1001***
PG1: 0110 ***PG2: 1101*** PG3: 0001 S4
S6 PG1: 0010 PG2: 0001 ***PG3: 1011***
PG1: 1110 PG2: 1001 PG3: 0001 S1
PG1: 0110 ***PG2: 1111*** PG3: 0001 S7
PG1: 0010 PG2: 0001 ***PG3: 1101*** S5
0
1

Figure 2. The selected field search tree (The gbv in italic and underline is the primary gbv)



*
1
2 PG1: 0000 PG2: 0010 PG3: 0000
3
4 PG1: 0010 PG2: 0011 PG3: 1101
5
6 PG1: 0000 PG2: 0010 PG3: 0000
7 PG1: 1000 PG2: 0000 PG3: 0000
8
9 PG1: 1000 PG2: 0110 PG3: 0000
10
11 PG1: 1100 PG2: 1000 PG3: 0010
12
13 PG1: 0110 PG2: 1011 PG3: 1101
14 PG1: 1000 PG2: 0010 PG3: 0000
15 PG1: 1000 PG2: 0000 PG3: 0010
16 PG1: 1000 PG2: 0000 PG3: 1000
0
1

Figure 3. The other field search tree (The number in the circle is the no. of the node.)

[1] In fact, the node stores a rule set with the common prefix of the rule set.

From the construction of parameter groups, a parameter group contains all rule sets stored in the nodes that are in a traversing path of the selected field tree from a leaf node up to the root node. Thus if a parameter group has a rule set whose selected field is A, the parameter group includes all the rule sets whose selected field are the prefixes of A. In addition, a rule set may be included in multiple parameter groups. Each parameter group has a corresponding gbv for each node in the selected field search tree. Thus, redundant information maybe exists to record the rule set stored in the node in all gbvs of the node. Consequently, the gbvs of the node in the selected field search tree have Property 3.

Property 3: For two-dimensional rule databases, to record each rule set at least once in the gbvs of the selected field search tree, only one gbv is necessary for each node storing a rule set. This gbv should contain the bit corresponding to the rule set stored in the node.

This gbv is called the primary gbv of the node if it satisfies Property 3. A node is called a matched node of a packet if the prefix represented by the node is matched by the packet corresponding header field. The gbvs with the same parameter groups in nodes of different search trees are called isomorphic gbvs. It is only reasonable to perform AND operation on isomorphic gbvs, i.e. gbvs with the same parameter group. Thus, we can get Theorem 1.

Theorem 1: If the selected field search tree and the other field search tree respectively have the matched node A and the matched node B of a packet, the primary gbv of node A and its isomorphic gbv of node B are enough to perform AND operation to get the rule set of the matched rule.

Property 3 and Theorem 1 assure that only a small part of gbvs is required to perform AND operation.

Finally, we show how to look up the matched rule. Firstly, when a packet arrives, a longest prefix match is performed in each field search tree to get the matched node. Secondly, the primary gbv and its isomorphic gbv are read from the matched nodes and performed AND operation to identify the rule sets of the matched rules. Thirdly, the least-cost matched rule is chosen from the rule sets of the matched rules according to Property 2 and the matched rules' cost. For example, a packet (0111, 1000) arrives to search the matched rule in Table I. We use 0111 and 1000 as search keys to lookup the matched nodes in the selected field and other field search trees simultaneously. In selected field search tree shown in Fig. 2, the search result is (PG2:1111) in the node S7. In the other field search tree shown in Fig. 3, the match node is node 6. Node 6 has three different gbvs, but we only NEED read the isomorphic gbv PG2 (0010) of the node. After the AND operation of the two bit vectors, S7 includes the matched rule according to the definition of PG2's gbv and R10 is the matched rule.

IV. PERFORMANCE EVALUATION

The major performance metrics include the lookup speed and the required storage. Firstly, we analyze the worst-case lookup speed and required storage of our algorithm when the lookup procedures on each field are performed in parallel. Like other bit vector based algorithm, the worst-case lookup speed is determined by both the worst-case speed of longest prefix match in each field and the times of memory access to the operation bit vectors. That is $O(W+\lceil R/M \rceil)$, where W, R and M are the maximum length of prefix, the length of the operation bit vector and the width of memories, respectively. Unlike BV and ABV, the length of the operation bit vector is equal to the group parameter independent of the size of rule databases. Thus, the worst-case lookup speed is $O(W+\lceil P/M \rceil)$, where P is the group parameter. The worst-case required storage of GBV is affected by both the maximum number of parameter groups and rule sets. In the worst case, the parameter group has only one rule set and the rule set includes only one rule. The number of parameter groups and the number of rule sets are both equal to the number of rules. Therefore, the worst-case required storage is $O(N^2 * P)$, where N is the number of the rules.

Secondly, we evaluate the performance of the GBV algorithm and compare it with the BV algorithm, ABV algorithm in two-dimensional packet classification. For large size rule databases, the lookup speed of bit vector based algorithms is mainly affected by the memory accesses, which is determined by both the length of the operation bit vector and the times to solve false matches. The required storage is affected by the total length of bit vector in each node and the number of the nodes containing bit vectors, which can be simply calculated with the product of the total length of bit vectors and the number of rules. Therefore, we compare the total length of bit vector (TLBV) in each node, the length of the operation bit vector (LBV), and the times of false matches (TFM) to occur between BV, ABV with different aggregate parameter and our algorithm with different group parameter. We use Classbench [12] to synthesize four large size two-dimensional rule databases and measure the mean value of TLBV, LBV and TFM. The experimental results are presented in Table IV.

TABLE IV. EXPERIMENTAL RESULTS FOR SYNTHESIZED DATABASES

	Rule size = 50,000		Rule size = 100,000		Rule size = 150,000		Rule size = 200,000	
Algorithm	TLBV	LBV /TFM	TLBV	LBV /TFM	TLBM	LBV /TFM	TLBV	LBV /TFM
BV	50,000	50,000 /0	100,000	100,000 /0	150,000	150,000 /0	200,000	200,000 /0
ABV(aggregate parameter =32)	1,563	1,563 /110	3,125	3,125 /201	4,688	4,688 /265	6,250	6,250 /432
ABV(aggregate parameter =64)	782	782 /97	1,563	1,563 /102	2,344	2,344 /189	3,125	3,125 /201
ABV(aggregate parameter =128)	391	391 /49	782	782 /96	1,172	1,172 /121	1,563	1,563 /145
GBV(group parameter=32)	10,144	32 /0	23,454	32 /0	28,800	32 /0	42,752	32 /0
GBV(group parameter=64)	9,536	64 /0	21,888	64 /0	27,904	64 /0	41,600	64 /0
GBV(group parameter=128)	8,832	128 /0	20,736	128 /0	26,624	128 /0	40,704	128 /0

As shown in Table IV, the BV algorithm is unpractical with the increase size of rule databases because its operation bit vector is too long. The ABV algorithm has the smaller value of LBV compared with GBV, but the value of TFM is much larger than that of the GBV algorithm. The ABV algorithm may have an unstable lookup speed because solving the problem of false matches needs extra memory accesses. The GBV algorithm has a fixed value of LBV independent of the size of rule databases. Thus, the GBV algorithm can get stable lookup speed.

The BV algorithm has the largest TLBV value and the ABV algorithm has the smallest one. When the problem of false matches is taken into consideration, extra storage is needed to store the information for distinguishing the true matched bits. The TLBV value of the GBV algorithm is between that of the BV algorithm and the ABV algorithm. In addition, it becomes smaller with the increase of the group parameter because more rule sets in consecutive traversing paths can be grouped.

Since the modification of the group parameter of the GBV algorithm can effectively change the length of the operation bit vector without false matches, the GBV algorithm is flexible and effective to improve the lookup speed compared with the BV algorithm and the ABV algorithms.

V. Conclusion

In this work, we proposed the GBV algorithm, which constructs the novel group bit vectors with the parameter group. The group bit vector of the algorithm makes the lookup speed independent of the size of rule database without false matches. Moreover, the group parameter of the algorithm can be adjusted so that the length of the gbv can be adapted to the width of memories. The GBV algorithm appears to be equally simple to be implemented in hardware or software.

Further work includes more simulations and the implementation in a network processor prototype. The aim is to study how the division of parameter groups depends on the selection of the field and the group parameter. We expect the required storage of the algorithm can be reduced further and the algorithm can be applied to multi-field packet classification.

Acknowledgments

We would like to thank Feifei Zhang, Shaohua Lei, Chen Li, Lei Zhang and Xiaoxin Fan in our lab for their helps in the implementation of network processor prototype. We also thank the anonymous reviewers for their thoughtful suggestions.

References

[1] T. V Lakshman and D. Stiliadis, "High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," *Proc. of ACM SIGCOMM*, pp. 203-214, October 1998.

[2] F. Baboescu and G. Varghese, "Scalable Packet Classification," *Proc. of ACM SIGCOMM*, pp. 199-210, August 2001.

[3] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 560-571, May 2003.

[4] V. Srinivasan, S. Suri and G. Varghese, "Packet classification using tuple space search," *Proc. of ACM SIGCOMM*, pp. 135-146, August 1999.

[5] E. Spitznagel, D. Taylor and J. Turner, "Packet classification using extended TCAMs," *Proc. of IEEE International Conference on Network Protocols(ICNP)*, pp.120-131, November 2003.

[6] V. Srinivasan, S. Suri, G. Varghese and M. Waldvogel, "Fast and Scalable Layer Four Switching," *Proc. of ACM SIGCOMM*, pp. 191-202, October 1998.

[7] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. of INFOCOM*, pp. 1193-1202, March 2000.

[8] P.Gupta and N.McKeown, "Packet Classification using Hierarchical Intelligent Cuttings," in Hot Interconnects VII, Aug. 1999.

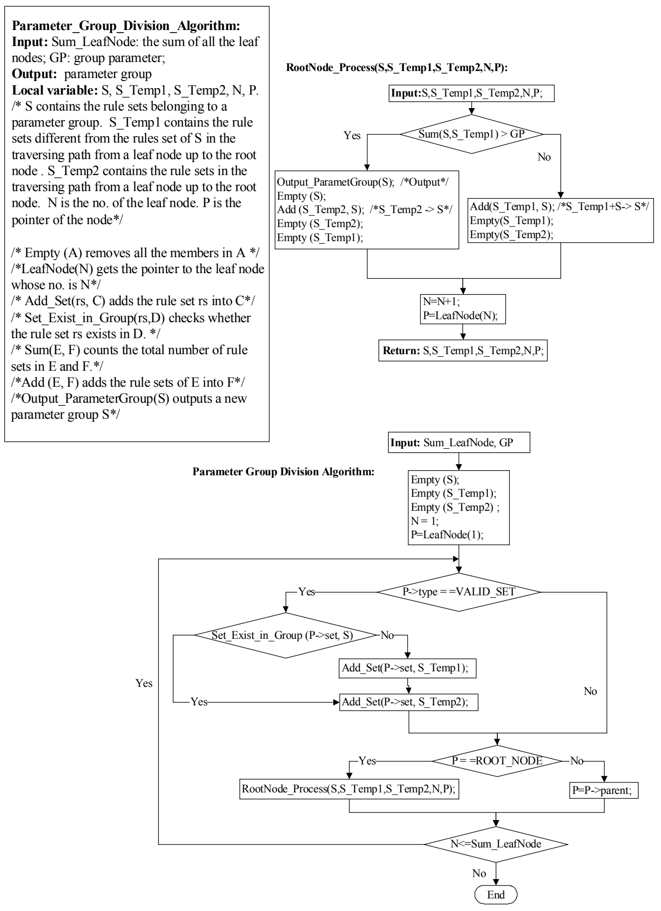
[9] S. Singh, F. Baboescu, G. Varghese and J. Wang, "Packet Classification Using Multidimensional Cutting," *Proc. of ACM SIGCOMM*, pp. 213-224, August 2003,

[10] P. Gupta and N. McKeown, "Packet Classification on Multiple Field," *Proc. of ACM SIGCOMM*, pp. 147-160, August 1999.

[11] D. Taylor and J. Turner, "Scalable Packet Classification using Distributed Crossproducting of Field Labels," *Proc. of INFOCOM*, pp. 269-280, March 2005.

[12] D. E. Taylor, J. S. Turner. The source code of Packet Classification Bench, http://www.arl.wustl.edu/~det3/ClassBench/index.htm

Appendix 1



Parameter_Group_Division_Algorithm:
Input: Sum_LeafNode: the sum of all the leaf nodes; GP: group parameter;
Output: parameter group
Local variable: S, S_Temp1, S_Temp2, N, P.
/* S contains the rule sets belonging to a parameter group. S_Temp1 contains the rule sets different from the rules set of S in the traversing path from a leaf node up to the root node . S_Temp2 contains the rule sets in the traversing path from a leaf node up to the root node. N is the no. of the leaf node. P is the pointer of the node*/

/* Empty (A) removes all the members in A */
/*LeafNode(N) gets the pointer to the leaf node whose no. is N*/
/* Add_Set(rs, C) adds the rule set rs into C*/
/* Set_Exist_in_Group(rs,D) checks whether the rule set rs exists in D. */
/* Sum(E, F) counts the total number of rule sets in E and F.*/
/*Add (E, F) adds the rule sets of E into F*/
/*Output_ParameterGroup(S) outputs a new parameter group S*/

RootNode_Process(S,S_Temp1,S_Temp2,N,P):
Input:S,S_Temp1,S_Temp2,N,P;
Sum(S,S_Temp1) > GP
Yes
No
Output_ParametGroup(S); /*Output*/
Empty (S);
Add (S_Temp2, S); /*S_Temp2 -> S*/
Empty (S_Temp2);
Empty (S_Temp1);
Add(S_Temp1, S); /*S_Temp1+S-> S*/
Empty(S_Temp1);
Empty(S_Temp2);
N=N+1;
P=LeafNode(N);
Return: S,S_Temp1,S_Temp2,N,P;
Input: Sum_LeafNode, GP
Parameter Group Division Algorithm:
Empty (S);
Empty (S_Temp1);
Empty (S_Temp2) ;
N = 1;
P=LeafNode(1);
P->type = =VALID_SET
Yes
Set_Exist_in_Group (P->set, S)
No
Add_Set(P->set, S_Temp1);
Yes
Add_Set(P->set, S_Temp2);
Yes
No
P = =ROOT_NODE
Yes
No
RootNode_Process(S,S_Temp1,S_Temp2,N,P);
P=P->parent;
N<=Sum_LeafNode
No
End

Figure 4. Parameter group division algorithm