Building Balanced Search Tree based on Layered Decision Tree for Packet Classification

Yeim-Kuan Chang
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701, Taiwan
ykchang@mail.ncku.edu.tw

Chao-Yen Chien
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701, Taiwan
p76994026@ mail.ncku.edu.tw

***Abstract*—Packet classification is an important building block of the Internet routers for many network applications, such as Quality of Service (QoS), security, monitoring, analysis, and network intrusion detection (NIDS). In this paper, we propose a scheme called Layer based Search Tree (LST) to solve multi-field packet classification problem. LST improves the traditional decision tree based schemes (e.g. HyperCuts and EffiCuts) by reconstructing the leaf nodes of the decision tree as an approximately balanced search tree. Since all the address subspace covered by each node of LST is disjoint, the buckets of the leaf and internal nodes in LST must not be empty. Thus, only the rules in one bucket can match the header values of the incoming packet. Searches on LST are completed immediately after the packet matches a rule in some internal node. In addition, we design the hardware search engine with pipeline and parallel architecture for the LST in Xilinx Virtex-5 FPGA environment. Because the memory usage of LST is very efficient, our search engine can support the ACL, FW, and IPC tables of 50k rules. LST search engine with dual ported memory can sustain the throughput of over 120 Gbps for the packets of minimum size (40 bytes).**

Keywords- packet classification; Pipelined Architecture; FPGA; decision tree;

I. INTRODUCTION

The packet classification problem is to determine the desired action (e.g., deny or permit) that should be taken by the incoming packets according to the highest priority rule selected among a set of predefined rules. Typically, the rules are identified by a 5-field packet header that includes the source and destination IP address, the source and destination port, and the protocol number. Each rule is also associated with a priority value to distinguish the importance among multiple matched rules. Packet classification is an enabling function provided by routers for many network applications, such as Quality of Service (QoS), security, monitoring, analysis, and network intrusion detection (NIDS). In order to keep pace with the increase of the link rates and the growing size of classifiers, how to search the larger classifiers efficiently is an important topic in recent years.

There are numerous solutions for packet classification. Among them, decision tree based algorithms, like HyperCuts [23] or HiCuts [9], are well-known approaches. The memory needed in the decision tree based schemes is used to store the internal nodes, leaf nodes, and the rules in the buckets associated with the nodes. And the lookup speed depends on the height of decision tree. HyperCuts builds the decision tree by cutting multiple dimensions at a time to obtain smaller tree height, but it suffers from large memory overhead. In this paper, we propose a novel packet classification scheme called Layer based Search Tree (LST). LST improves the existing decision tree based schemes by having two phases, partition phase and classification phase. In the partition phase, rules are partitioned into several buckets which are corresponding to the leaf nodes of binary decision tree. In the classification phase, we consider the leaf nodes of decision tree as sorted elements to construct an approximately balanced binary search tree. LST can use the binary decision tree which has the least number of rule duplications without damaging the search speed because speed of LST depends on the number of leaf nodes rather than height of the decision tree.

Due to the high-speed link rate of router such as OC-768 (40Gbps), software solution is hard to achieve this requirement. The 40Gbps means the router must processes 40 bytes packet every 8 ns. Thus, field-programmable gate array (FPGA) has become a good choice for real-time network processor. Although many existing FPGA-based approaches can over 40Gbps for their throughput, the hardware resource (block RAM) is still a bottleneck that their approaches merely design for smaller or not complex rule table (ACL). We can implement our proposed scheme, layer based search tree, into FPGA with larger tables because our proposed scheme need less memory requirement. In this thesis, we also design a FPGA engine that has low hardware cost and still keep pace with the high throughput.

II. RELATED WORK

We discussed the decision-tree-based approaches HiCuts [9], HyperSplit [20], HyperCuts [23], and EffiCuts [27] because they are related to our proposed scheme. In decision tree based scheme, a pre-computed decision tree is built as follows: Suppose a node *v* in the decision tree contain a set of rules. All the rules in *v* are distributed into child nodes of *v* and some rules may be duplicated in more than one node. The rule distribution process is repeated at each child node

until the number of rules in each node is smaller than the predefined threshold. In search step, by traversing the decision tree we can find a set of candidate nodes. Then, the buckets associated with the candidate nodes can be searched linearly to find the highest priority rules. HyperCuts is an improved version of HiCuts. Each node in HyperCuts selects multiple fields to distribute the rules rather than one field at a time. Thus the height of the decision tree in HyperCuts is shorter than that of HiCuts. But its rule duplication may be larger. HyperSplit and EffiCuts are both proposed to reduce the memory size blowout problem. HyperSplit uses endpoint position instead of bit position which represents the specific endpoint to reduce rule duplication. EffiCuts classifies the rules into some subsets according to the wild cards in each filed and uses equi-dense cut to reduce the number of empty nodes.

There are many hardware based search engines for packet classification. In this section, we will introduce some search engines for Ternary Content Addressable Memory (TCAM) or Field Programmable Gate Array (FPGA). TCAM is a specific hardware in which each entry can store a ternary string containing 0, 1 or * (wildcard). Besides, all TCAM entries can be compared to the input headers in parallel in one clock cycle. But TCAMs suffer from many disadvantages that include high power consumption, range-to-prefix expansion blowout, and high cost. Song et al. proposed an architecture called BV-TCAM [24] that combines the TCAM and the Bit Vector algorithm for multiple matching results.

There are many approaches adopt bloom filter for search engine. NTLMC [8] implement the cross-product algorithm and bloom filter into the search engine. Besides, B2PC [19] uses multi-level bloom filters in decomposing multi-field classification to implement the search engine. 2sBFCE [18] also use bloom filter and it is implemented in ASIC. The above Bloom-Filter based schemes do not achieve the high link rate of OC-768 (40Gbps).

The tree based search engines that use pipeline or parallel architectures can improve their throughputs. Several existing work [14][21][28][6][15] discusses how to implement decision tree based algorithms on FPGA. Two-dimensional Linear Dual-Pipeline [14], BiConOLP [28], and Power Saved HyperCuts [15] implement HyperCuts. Two-dimensional Linear Dual-Pipeline uses several linear pipelines to process internal node buckets. BiConOLP balance the dual port memory in each pipeline stage by bidirectional subtree. Power Saved HyperCuts uses adaptive clocking unit for power saving. HyperSplit on FPGA [21] and Hyper-Cutting scheme [6] both implement endpoint base decision tree algorithm. SPMT [3] and SPSTwB [7] belong to the set-pruning tree based architecture. Rather than binary set-pruning trie [12], SPMT and SPSTwB propose set-pruning multi-bit trie and set-pruning segment tree to reduce the number of memory access. Due to the large rule duplication problem of set-pruning tree base algorithm, both of the schemes partition the rule set into many groups to reduce the memory requirement. All of above tree based search engines with dual-port memory can achieve the high link rate of OC-768. The main reason is that they all adopt the parallel and pipeline architecture.

Table I. Rule table analysis for 10K table

	Table size	ACL	FW	IPC
Number of distinct field values	SA	4473(44%)	8733(87%)	1557(15%)
	DA	595(6%)	3081(30%)	3105(30%)
	SP	1	9	34
	DP	108(0. 1%)	1	54
	Prot	4	5	7
Number of wildcards	SA	35	978(10%)	367(4%)
	DA	56	4959(50%)	225(2%)
	SP	10000(100%)	3497(35%)	8322(83%)
	DP	2792(28%)	10000(100%)	5455(55%)
	Prot	802(8%)	2506(25%)	3311(33%)

III. RULE TABLE ANALYSIS

In recent years, there are so many algorithms for packet classification. Due to the different characteristics of three type tables (ACL, FW, and IPC), none of the existing packet classification schemes can be perfectly suitable for all types of the tables. Thus, we will analyze the three filter sets which are generated by ClassBench [25] in Table I to Table III. We count the number of wildcards for each of the five fields because the wildcards imply the heavy rule duplications that may cause more memory usage in many existing algorithms. Besides, we also calculate the number of distinct field values for each field and use them to determine which filed is a better choice to perform the cutting operation. Because a field that contains a large number of distinct values represents that it has more information to be used for the classification process. The last column in the analysis tables of these figures named Summary shows the ratios of number of distinct field values and total number of rules or number of wildcards and total number of rules. Table I shows the results for ACL. We can see that wildcards account for smaller than 1% of the Source or Destination IP field. The source port fields in ACL are all wildcards. And the number of distinct field value in source and destination field is larger than other three fields. Thus, the source and destination IP field is suitable than other three fields to solve the ACL table because these two fields are outstanding in number of distinct values and wildcards. The source port is not suitable obviously.

Table II. The length distribution of source and destination IP field

	Length	ACL-10K	FW-10K	IPC-10K
SA	0	7	5672(60.9%)	556(6.2%)
	1-8	14	28	0
	9-16	0	71	457(5.1%)
	17-20	0	0	133(1.5%)
	21-24	781(8.1%)	255(2.7%)	3547(39.1%)
	24-28	0	484(5.2%)	814(9%)
	28-31	584(6%)	372(4%)	95(1.1%)
	32	8217(85.6%)	2429(26%)	3435(38%)
DA	0	26	2361(25.4%)	461(5.1%)
	1-8	109(1.1%)	0	0
	9-16	164(1.7%)	0	681(7.5%)
	17-20	0	0	9
	21-24	1073(11.2%)	236(2.5%)	2543(28.1%)
	24-28	51	243(2.6%)	1102(12.2%)
	28-31	223(2.3%)	122(1.3%)	66
	32	7957(82.9%)	6349(68.2%)	4175(46.2%)

Compared with ACL table, FW table has more number of wildcards. In ACL table, the ratios of wildcards in source and destination IP fields do not reach 1%. Different from ACL tables, the ratios of wildcards for FW table are about 10% in source IP field and 50% in destination IP field. The ratios of distinct field values in source and destination IP fields are about 80% and 30% and the ratios of the other three fields are less than 1%. According to the number of distinct field values, the source and destination IP field is more important than the other three fields to solve the FW table. But these two fields also have too large number of wildcards. In order to solve the problem in these two fields, we can partition the rule set according to wildcard of source and destination IP fields. Each group of rule set can be easily solved by decision tree method or other algorithms.

As the ACL tables, the ratios of wildcards in both source and destination IP fields are smaller than other three fields. Besides, the ratios of these two fields are larger than ACL but smaller than FW. Furthermore, the pattern of IPC for number of distinct field values is similar to that of ACL. As in ACL and FW, the source and destination IP fields obviously have more number of distinct field values than the other fields. So, we also focus the source and destination IP field in IPC. According to the above analysis, IPC rule table is similar with ACL rule table that they both have less number of wildcard and larger number of distinct field value in source and destination field. But, memory usage of decision tree scheme for IPC is worse than ACL even if we divide the rule table into some subgroups by the wildcards for source and destination field. The result let us know that it surely has other factors which influence the memory usage in source and destination IP field. In fact, the prefix length of source and destination field is the key factor for why IPC still need more memory than ACL.

Because the rule set has much more number of distinct values in source and destination IP fields than other three fields, we would usually select the source and destination IP fields to partition. And the length distribution for source and destination IP field is shown in Table II. The value in parentheses is the ratio of number of rules which has that prefix length in that field and total number of rules. The 10K rule tables of Table IV are public available in [13]. In ACL-10K, we find out that the proportion of prefix length "32" in source and destination IP field is 85.6% and 82.9%, respectively. But, they are 38% and 46.2% in IPC-10K. No matter the rules is in source IP or destination IP field, the length of prefix in ACL is larger than 80%. But, they are smaller than 50% in IPC. This characteristic can be verified why IPC has more memory usage than ACL. Decision scheme for ACL rule table has less memory usage because most rules in source and destination IP field would not cause any duplication. Although the proportion of length "32" in source and destination IP field is 26.6% and 68.2% in FW-10K, the ratio of wildcard are 60.9% and 25.4%. For source and destination IP field, total rules in wildcards and length "32" are 87.5% and 93.6%. This analysis shows that the rules in FW are almost wildcard or length "32" prefix. This is why grouping the rule set by wildcard field can easily reduce the memory usage in FW.

IV. PROPOSED SCHEME

Many exist works [2][5][12][19] shown that simple groups method can reduce rule duplication efficiently. EffiCuts [19] solved the memory overhead problem by dividing rules into some groups according to the wildcard (or almost wild card) fields. Our scheme also categorized the rules into subgroups. Because the classification capability of prefix fields is more effective than range and protocol fields, the criterion of our grouping method is only based prefix fields. There are four groups in the proposed scheme：

group 1: rules with wildcards in both source and destination address fields.
group 2: rules with wildcard only in source address field.
group 3: rules with wildcard only in destination address field.
group 4: rules without wildcards in both source and destination address fields.

We select either group 2 or group 3 that has more rules than the other to merge with group 1. As a result, we have three groups left. Then we construct a search tree for each of the three groups independently. The search tree construction has two phases, namely *partition phase* and *classification phase*. In the partition phase, we construct the binary decision tree to partition the rules in the group into several buckets which exist in the leaf nodes of the binary decision tree. The bucket size is limited by a predefined threshold *T*. In the classification phase, all the leaf nodes in binary decision tree are used to build the approximately balanced search tree. Rather than searching from the root to a leaf node, searches on this tree can stop when an internal node may contain matched rules. Finally, we linearly search the bucket which is pointed by the matched node to find the best matching rule. To illustrate our proposed scheme, an example of two-dimensional (2-D) classifier is shown in Figure1. Each rule $R_i = (R_i[1], R_i[2])$ is a rectangular area in the 2-D address space.

A. Partition phase

In this phase, a binary decision tree is used to divide the rules into several buckets. The binary decision tree is built by

Rule R_i	R_i[1]	R_i[2]
R_1	[24, 31]	[24, 31]
R_2	[0, 7]	[16, 23]
R_3	[0, 7]	[0, 7]
R_4	[12, 15]	[0, 7]
R_5	[0, 15]	[12, 15]
R_6	[24, 27]	[16, 19]
R_7	[12, 15]	[16, 31]
R_8	[24, 31]	[4, 7]

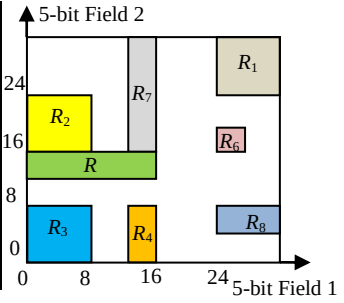

Figure 1. An example 2-D classifier with eight rules in $2^5 \times 2^5$ address space

cutting each node independently. Each node v in the binary decision tree is associated with the following attributes:

1. $sp(P_1[v],\ldots, P_d[v])$: the address space covered by node v, where d is the number of fields of the rules. The root node covers the entire d-dimensional address space.
2. $rule(v)$: the set of rules that intersect $sp(P_1[v], \ldots, P_d[v])$.
3. $field(v)$: the selected field to cut the address space of v.
4. $right(v)/left(v)$: the right and left child pointers of v.
5. $LF(v)$: the layer to which node v belongs.
6. $L(v)$: the set of labels from the first layer to $LF(v)$.

Basically, the proposed decision tree is constructed based on Hicuts [6]. For each node v to be partitioned in the decision tree, we select the field denoted by $field(v)$ such that the resulting number of rule duplications is the least compared to cutting the node along any other fields. Fig. 2 shows an example in 3-bit address space for a node v containing two rules, R_1 = ([000, 001], [110, 111]), R_2 = ([000, 110], [000, 011]). If node v is partitioned along field 1 at the first bit, the left child node u of v will contain two sub-rules R_1 = ([**0**00, **0**01], [110, 111]), R_2 = ([**0**00, **0**11], [000, 110]) and the right child node of v will contain only one sub-rule R_2 = ([**1**00, **1**10], [000, 011]). However, if v is partitioned along field 2, both the left and right child nodes u and z of v will contain only one rule which are R_2 = ([000, 011], [**0**00, **0**11]) and R_1 = ([000, 001], [**1**10, **1**11]), respectively. As a result, partitioning node v along field 2 is preferred. The node cutting process continues until it contains no larger than T rules, where T is a predefined threshold.

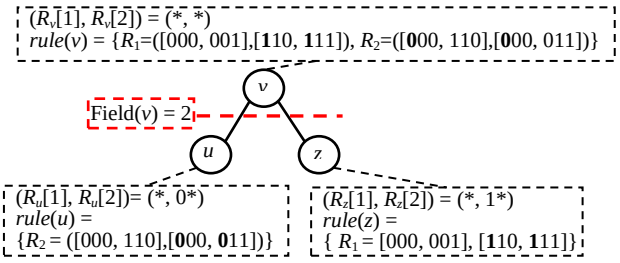

Figure 2. Node v is divided along field 2 into two child nods, u and z.

We define a parameter, called *layer* such that all the nodes in the same layer are partitioned along the same field, called layer field (*LF*). The root of the decision tree is initially defined to be in layer 0. As in the example of Fig. 2, nodes u and z are partitioned along the same field and thus they belong to the same layer. If the partition field of node u is also field 2, both of node u's child nodes are also collected in the same layer of u. In Figure 3, the layer numbers are shown with different colors based on the layering process described above. As we can see, nodes b, c, and f are classified in layer 1 because they are obtained by cutting their parent nodes along field 1. Similarly, nodes g, d, h, e, k, o, l, and p belong to layer 2 because they are obtained by cutting their parent nodes along field 2. Next, we will describe how to compute the label set $L(v)$ for node v which is the key factor in classification phase. Initially, the label set of root node is set to empty, $\emptyset$. If a node and its two child nodes are in the same layer, its label set inherits from its parent node. Nodes c, d, and l in Fig. 3 are the examples for this case. If a node v is not in the same layer as its child nodes, its label set will be set to its parent's label set appended to by field value of covered space in field $field(v)$. For example, the label set of node f, $L(f)$, is $L(c) + P_1[f]$ = 11* because $L(c) = L(a) = \emptyset$ and $P_1[f]$ = 11*. Finally, if there are n layers in the decision tree and the label set of a leaf node v is of size k, then $n - k$ *'s need to be appended. For example, $L(c)$ is {11*, 11*, *} where the last don't-care is finally added.

Label set is the construction basis of classification phase. Besides, the rules in each bucket are arranged from high

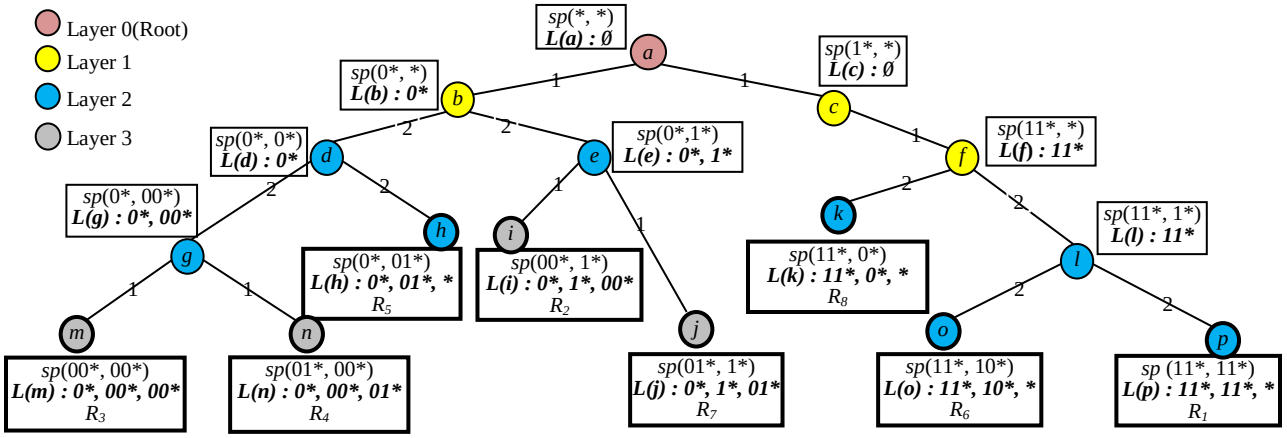

Figure 3. A binary decision tree built according to rules in Table I, where the predefined threshold T = 1.

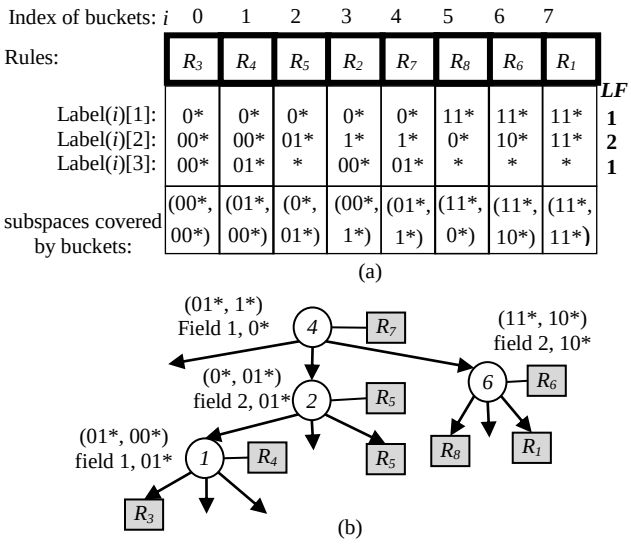

Figure 4. (a) Bucket array corresponding to the leaf nodes in Figure 3 (b) A binary search tree built according to (a)

priority rule to low priority rule. Once the some rule is matched, it guarantees that the rule is the highest priority matching rule in that bucket.

B. Classification Phase

After the partition phase, the leaf nodes of binary decision tree are taken to construct the search tree which is built as balanced as possible. First, the buckets of leaf nodes are arranged in an array. Each bucket has a multi-field label set which is associated with the corresponding leaf node. The bucket array is ordered by projected position of leaf nodes in the binary decision tree on the horizontal line. Each bucket contains a set of rules, a label set containing k labels, and the address space covered by this bucket. Let *L(i)[k]* denote the k^{th} label of the i^{th} bucket. *L(0)[j]* to *L(n-1)[j]* is called label array j, where n is number of buckets and $j \le k$. Figure 4 (a) shows the result of bucket array from Figure 3. The original motivation is that we consider the label set, *L(i)[0]* to *L(i)[k]*, as a wide identifier to be searched, where k is number of layers in binary decision tree and i is the index of buckets. Bucket array then can be binary searched by label set array. However, the comparison between each label set is too complex in the worst case because the number of labels for each bucket is about 2 to 9. Therefore, we propose the layer partitioned search tree (LPST) to solve such problem. Instead of comparing all labels in a label set, LPST only compares a single label field and sometimes it needs to check whether the node is matched. Each node of LPST is corresponding to a bucket, so the number of nodes is the same as the number of buckets. Besides, each node has three pointers to point to the subtrees that contains labels smaller than, equal to, and larger than the label of the node. Assume L, R, and M are the leftmost, rightmost, and middle indices of bucket array. For the creation of each node, the label array must be checked first. Then one of buckets in bucket array is picked to construct the LPST according to this label array. From $j = 1$ to k, the following operations are performed. We select the smallest j such that that *L(R)[j]* to *L(L)[j]* are not all identical. Next, we find two indexes *El* and *Er* such that *L(i)[j] = L(M)[j]* for $i = El$ to Er and $L \le El \le M \le Er \le R$. Thus, buckets bucket[$R$] to bucket[$L$] are divided into four parts: (1) smaller buckets bucket[*L...El*–1], (2) the selected bucket[*Em*], any one from buckets bucket[*E1…Er*], identity buckets bucket[*E1…Em*–1,*Em*+1…*Er*], and larger buckets bucket[*Er*+1…*R*]. Notice that the spaces covered by buckets bucket[*E1…Er*] are disjoint and so if we can find a match in any one, this match must be the only match in buckets bucket[*E1…Er*]. Then, a node is created to record bucket[*Em*], label field *L(Em)*[j], and the subspace covered by bucket *Em*. This node creation process is repeated recursively so that the left pointer pointing to the subtree created from bucket[*L...El*–1], middle pointer pointing to the subtree created from bucket[*E1…Em*–1,*Em*+1…*Er*], and right pointer pointing to the subtree created from bucket[*Er*+1…*R*]. Figure 5 shows the LPST construction algorithm.

```
// Construction of LST
LPST_create(Node, L, R) {
01  j = LabelArrayFind(L, R); //Inspect label arrays and return the
        smallest number j such that L(R)[j] to L(L)[j] are not all identical;
02  M = (L+R+1)/2;
03  Find two indexes El and Er such that L(El)[j] to L(Er)[j] are all
        identical. (L ≤ El ≤ M ≤ Er ≤ R).
04  Node.index = Em such that bucket[Em] has the highest priority
        among all buckets bucket[El..Er];
05  Node.label = bucket[El].label(j);
06  Node.region = bucket[El]. bucket_region;
07  if (El > L)    LPST_create (N->left, L, El – 1);
08  if (Er > El)   LPST_create (N->middle, El, Er);
09  if (R > Er)    LPST_create (N->rightt, Er+1, R);
}
```

Figure 5. Algorithm to construct LST.

Figure 4 (b) shows the LPST built according to the buckets array in Figure 4 (a). The index of middle bucket is 4 and this bucket will be the selected bucket (root). Then, the first label of bucket 4 (0*) is used to look for the smaller array, identical array, and larger array. Since the bucket whose first label is smaller than selected bucket's does not exist, the left pointer of root is NULL. Identical buckets (bucket[0…3]) which contains the identical label 0* and the remaining buckets are larger buckets (bucket[5…7]). Then middle subtree is created by middle buckets bucket[0…3] and right subtree is created by larger buckets bucket[5…7]. The middle child of root is node 2 because it is middle bucket in bucket[0…3]. And the subtree of node 2 is constructed by bucket[0,1,3]. Then, right child node of root node is node 6 and the subtree of node 6 is constructed by bucket[5,7]. Similar operations are performed to recursively construct the whole search tree.

For each incoming packet, the packet headers are used to traverse the search tree. For each search step, the corresponding field is compared with the label. If the corresponding field is matched, the subspace of this node is checked to see whether the node is matched or traversal goes ahead to the middle direction. Otherwise, if the

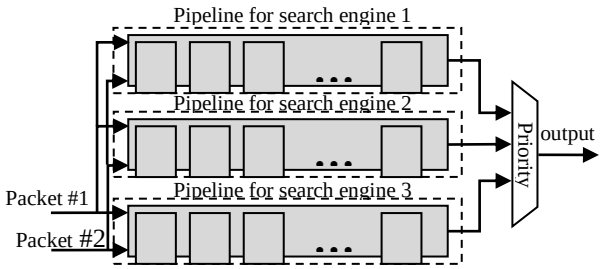

Figure 6. System block diagram for parallel search engines

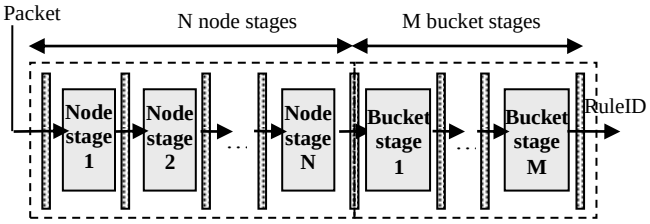

Figure 7. The architecture for each search engine.

corresponding field is larger or smaller than the label of the node being checked, the direction of traversal would be right or left, respectively. If the node is matched, the rules associated with this node are linearly searched until a matched rule is found. For example, the packet header (00010, 10010) is used to search in Figure 4 (b) In root, the field 1 header matches the label (0*), but it doesn't match the *sp*(01*, 10*). So the node 2 is searched next because the label of 0* matches 00010. The field 2 header 10010 doesn't match the label (01*). Since 10010 is larger than its label 01*, node 3 is the next node to search. The label and subspace of node 3 both match the packet headers. As a result, the bucket of node 3 is linearly searched and R_2 is the final result.

As mentioned earlier, LPST has three groups which are classified by wild card. Searching three trees sequentially decreases the overall search performance. We can increase the overall search performance by maintaining the highest-priorities in the second and third groups. Thus, most of the searches can be completed without searching the next group if the priority of matched rule in the prior group is higher than the next group. Besides, the decreasing order of rules in each bucket can also save some search times if the matching rule has higher priority than other rules in the bucket.

C. Hardware Architecture

By our grouping method, we construct the three separable search trees. Each search tree would be mapped to a search engine. Then, we parallel the search engine and use pipeline technique to increase the throughput. Due to our linear pipeline structure, it has many advantages:

1. Each packet can be delay in a constant time.
2. The order of packet can be maintained.
3. One clock cycle can resolve a packet.

Figure 6 shows the system block diagram of basic architecture with dual port Block RAMs. Two packets can be input at the same time because dual port Block RAM allows two packets read the data from the same Block RAM module simultaneously. Each packet would traverse all the three search engines in parallel. The packets input to the three search engine simultaneously and they also departure from the search engines at the same time. After the packet departure, each three search engines would output a rule ID. If some search engine doesn't have match rule, it will output the default ID. The priority resolver would select the highest priority rule to output.

The Figure 7 shows the architecture for each search engines. If the height of search tree is N and bucket size is M, the corresponding search engine will be N node stages and M bucket stages. For the search engine, the packet needs to go through N node stages. The pipeline bucket address must be transmitted along linear pipeline stages from first stage to stage N. Any stages have the chance to modify the pipeline bucket address for transmission. If the matching rule store in this search engine, one of the node stage would modify the correct pipeline bucket address. The following stages would not allow modifying the transmission bucket address because the packet would match only one node. If the matching rule do not store in the search engine, the wrong pipeline bucket address also need to be transmitted to node stage N. Then, packet goes through M bucket stages according to the pipeline bucket address and it will output the default rule ID. In each bucket stage, the packet compares a rule which is at the pipeline bucket address and output the match rule.

In the node stages, we need to map the search tree node into pipeline memories and design the delicate hardware circuit for dimension selection and label comparator.

We show our data format before we explain how to design the node pipeline stages. The node data format is shown in Figure 8. We will mention the node data format in the following description:

(1) Rule space covered by node: Because the nodes of

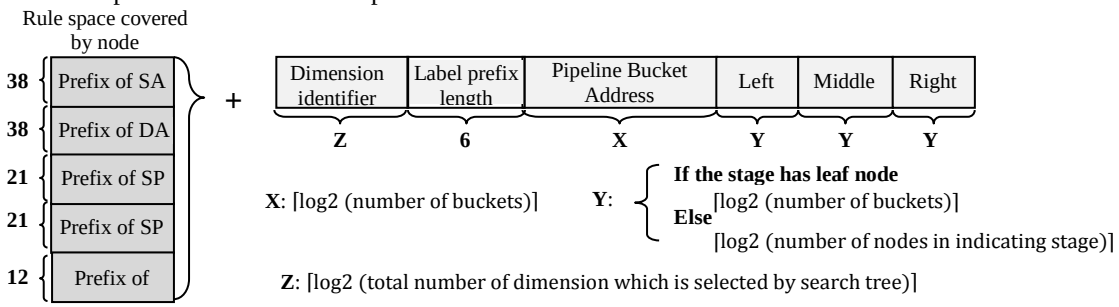

Figure 8. Node data format

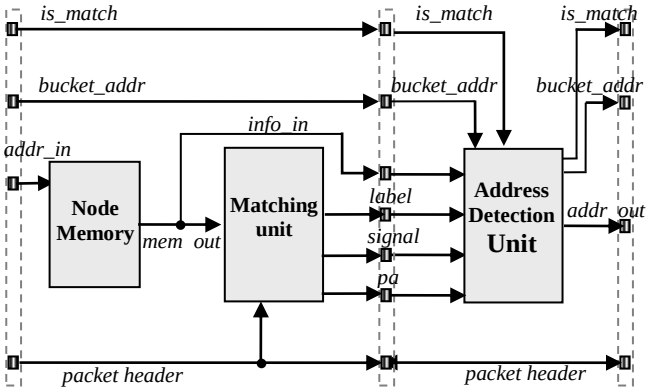

Figure 9: The architecture of node stage

LST in partition phase are cut by bit, the rule space covered by node is consisted by prefix set. The prefix set are prefix of SA, prefix of DA, prefix of SP, prefix of DP, and prefix of Protocol. For each separable tree, some dimension may never be selected that the corresponding prefix of each node would be wildcard. For example, the third tree of LST that all rules have wildcards in both source and destination address field would never select the source and destination address field to cutting. Thus, if the nodes of search tree all contain the wildcard prefixes with same dimension, we can consider the wildcard prefixes as the default prefixes and do not need to record them. The SA and DA of prefix both contain 32 bit value and 6 bit mask. The SP and DP of prefix both contain 16 bit value and 5 bit mask. The Protocol of prefix is 8 bit value and 4 bit mask.

(2) Pipeline bucket address: Due to all the nodes of search tree are leaf nodes of binary decision tree, each node of search tree records a bucket address. Each pipeline stage will inspect whether the node is matching or not, the pipeline bucket address which is contained by matched node will transmit to first bucket stage by stage. The address bits are corresponding to the number of buckets. The size is $\lceil \log_2(\text{number of buckets}) \rceil$.

(3) Left, Middle, and Right: They are the pointer addresses of left, middle, and right child node. Due to these three pointers in leaf node maybe be used to point to the bucket. Thus, the Right, Middle, and Left address bits are corresponding to the number of nodes in indicating stage or number of buckets. If the pipeline stage contains leaf node, the size of Left, Middle, and Right in this pipeline stage is $\lceil \log_2(\text{number of buckets}) \rceil$. Otherwise, the size is $\lceil \log_2(\text{number of nodes in indicating stage}) \rceil$.

(4) Dimension identifier: It is a single value to indicate the selected dimension. If the search tree only select SA, DA, and DP to construct, the value 0~2 for dimension identifier represents SA, DP, and DP, respectively.

(5) Label prefix length: The format of label prefix is "value/length". The label prefix length is the mask of label prefix. Because the value of label prefix can be obtained from "rule space covered by node", we only store the prefix mask. The size is 6 bits because the largest prefix value is 32 bits that the length can be 0 to 32.

Figure 9 shows the block diagram of single node stage. In order to speed up the clock rate, the operation of node is partitioned into two pipeline stages. Although using two stages to deal with one search tree level, we still can accept the total stages because height of search tree is not too large. Each node operation has three steps:

1. Access node memory: Read memory by the input address (*addr_in*) and output *mem_out* which is the corresponding node data structure. The format of *mem_out* is shown in Figure 8. Besides, *info_out* is part of *mem_out* that they are **Pipline bucket address**, **Left**, **Middle**, and **Right**. The characteristic of *info_out* is that it doesn't need to be compared in matching unit.

2. Matching unit: Each field of packet header is taken to match the corresponding prefix in **"Rule space covered by node"**. *signal* is the bit-vector that each bit records whether the corresponding field prefix is matched or not. If some prefix matches the corresponding field of packet header, the corresponding bit would be 1. Otherwise, the bit of bit-vector is 0. Matching unit also output *label* and *pa*. *label* is generated by extracting the value in **"Rule space covered by node"** according to the Dimension identifier and right shifting this value according to the Label prefix length. *pa* is generated by fetching one field value from the packet header according to the dimension identifier and also right shifting this value according to the Label prefix length.

3. Address Detection Unit: Address detection unit has two parts. One part determines the child address (*addr_out*) for the next node stage by comparing the *label* with *pa*. Another one part determines the pipeline bucket address (*bucket_addr*) to output. If *is_match* shows the node is match, the output *bucket_addr* would be the same as input *bucket_addr*. Otherwise, we we must identify output *bucket_addr* that *bucket_addr* is Pipeline bucket address which is fetched from *info_in* or *addr_out*. According to the singnal_in, we identify the bit vector whether the node is matched or not. If the node is matched, *addr_out* must be pipeline bucket address which transmits form *info_in*. And *match_out* must output 1. Otherwise, *addr_out* would be *addr_out* and *match_out* output 0.

We take two adjustments to improve the bucket stage. One adjustment is bucket merging. Even if we use binth to restrict the maximum number of rules in each bucket, each bucket still has different number of rules. Because the packets need to traverse from the first stage to the last stage, each stage needs to consume one clock cycle even if no rule needs to be compared in that stage. Thus, we can merge some buckets so that total rules still do not exceed the size of bucket. First, the bucket must sort according to the number of rules. The condition of bucket merging is that number of rules for each bucket can't exceed the bucket size after the bucket merging. The buckets which contain the rule duplication need to be merged first. We can remove the duplication rules from the merging bucket. Then, we merge the remaining buckets. We inspect the bucket from first to last to merge with the following buckets. After the bucket merging, we need to sort priority of all the rules for each

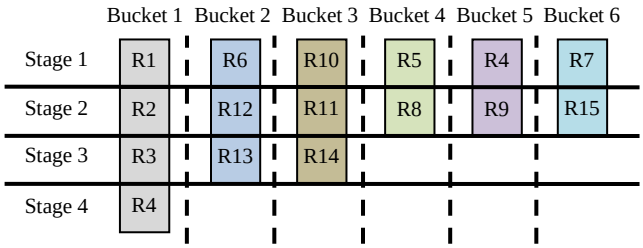

Figure 10. The Rule lists.

bucket in decreasing order of the priority. For this design, we do not need to compare the priority in each bucket stage because we can guarantee that the last matched rule has the highest priority.

Another one is bucket size increasing with no responds time increasing. For each search tree, the number of total stages is not the same. The search engine which contains the less number of stages needs to wait for the one containing more stages. A block RAM size is restricted by 1024 entries. If some stages for the smaller search engine contains more than 1024 entries, we maybe can increase the bucket size to decrease the number of block RAM usage and the total stages with bucket size increasing is no larger than the largest one.

Figure 10 shows pipeline stage with simple mapping for six uneven buckets. The method of simple mapping is that we sort the bucket by the size of buckets and then directly map the buckets in the pipeline stage in order. For this example, we need three bit bucket address to record the six buckets. Figure 11 shows the bucket pipeline mapping with adjusting the buckets in Figure 10. We assume that bucket size is five after the bucket increasing. We first find the buckets which contain the duplication rule to merge that bucket 1 and bucket 5 both have R4. Besides, the number of rules would be five after the bucket merging. So we merge the bucket 1 and 5. Then, we inspect remaining bucket to merge that bucket 2 and 3 can be merged with bucket 4 and 6. After the bucket merging, the bucket address only needs 2 bits instead of 4 bits.

V. EXPERIMENTAL RESULTS

In this section, we will show our hardware performance and compare with other hardware based packet classification schemes. Our scheme is implemented in FPGA by verilog

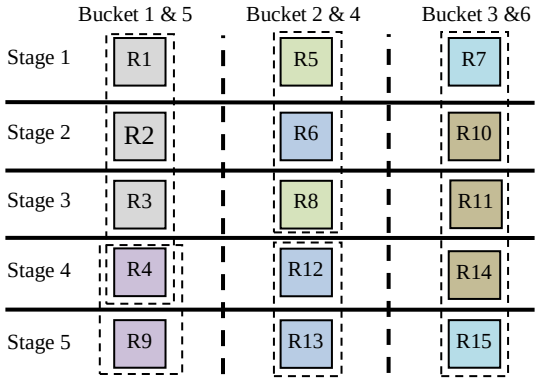

Figure 11. Mapping with adjustment.

with Xilinx ISE 12.2 development tools. The target device is Xilinx Virtex-5 XC5VFX200T [28] with '-2' speed grade.

The bucket size of each search tree in different rule tables would be variable. We will discuss the influence of Block Ram usage in different bucket sizes. Table II shows the usage of Block Ran with variable bucket size in FW1 10K table. The second row is the number of rules with variable bucket size. We can see that the large bucket size have the less rule duplication and our proposed scheme cause less rule duplication. Even if the bucket size for each search engine is set to 4, the number of rules in our proposed only causes more 10% than number of rules in original rule table. For each search engine, the different bucket size leads to different number of Block Ram. If we decrease the bucket size, pipeline stages for bucket search would be less. But the number of node pipeline stages maybe becomes larger. Besides, a Block Ram has 18 bits bandwidth and 1K entries. Bucket size decreasing may cause more entries for each pipeline stages. In other word, it is possible to increase the number of Block Ram in each stage. For Table 6-7, bucket size 5, 7, and 7 for each search engine have the least number of Block Rams. So we choose this three bucket size to implement FW1 10K tables.

Table IV shows the performance of rule duplication and bucket merging for 10K rule tables. The duplication ratio of # of rules in original rule table and # of rules in LST is 1.06~1.68. For FW_10K, the merging ratio of (# of buckets before merging - # of buckets after merging) and # of buckets before merging is lower because buckets before merging are almost full.

Table II. Number of Block Ram with different bucket size and number of rules in FW1 10K rule table

Bucket size		4	5	6	7	8	9	10	11	12
Number of Rules (original: 9311 rules)		10307	1005	9901	9861	9807	9769	9719	9669	9615
Search engine 1	Number of 18K Block Ram	117	90	108	108	117	117	126	135	135
Search engine 2	Number of 18K Block Ram	112	112	104	96	104	120	128	136	144
Search engine 3	Number of 18K Block Ram	78	78	78	72	78	78	78	84	90

Table IV. The performance of rule duplication and bucket merging for 10K rule tables

	ACL1_10K	FW1_10K	IPC1_10K
# of rules in table	9603	9311	9037
# of rules in LST	11098	9867	15141
Duplication ratio	1.16	1.06	1.68
Bucket Size for each search tree	12, 7, 2	5, 7, 7	13, 10, 3
# of buckets before merging	2768	2099	3245
# of buckets after merging	964	1939	1251
Merging ratio	0.65	0.08	0.61

Table V The detailed performance statistics of the proposed scheme for 10K rule table

		ACL1_10K	IPC1_10K	FW1_10K
Engine 0	Bucket Size	12	13	5
	Tree height	12	13	10
	Tree Memory(KB)	37.64	38.13	12.8
	Bucket Memory(KB)	181.17	212.69	63.1
	Total Memory(KB)	218.81	250.82	75.9
Engine 1	Bucket Size	7	10	7
	Tree height	8	12	9
	Tree Memory(KB)	1.29	7.23	11.89
	Bucket Memory(KB)	6.63	36.55	69.73
	Total Memory(KB)	7.92	43.78	81.62
Engine 2	Bucket Size	2	3	7
	Tree height	2	4	9
	Tree Memory(KB)	0.017	0.19	1.84
	Bucket Memory(KB)	0.026	0.33	5.28
	Total Memory(KB)	0.043	0.52	7.12
Total Memory usage(KB)		226.78	295.13	164.64

In table V, we show the memory usage of our proposed scheme for ACL1 10K, IPC1 10K and Fw1 10K table. Taking FW_10K for example, we can see that our proposed scheme needs 164.64 KB. According to the Table 6-7, the estimated number of Block Ram is 258 and the Block Ram we used is 580.5 KB (258*18Kbit). That is to say the 71.6% memory is empty. Thus, we use distributed Ram which doesn't restrict the size of bandwidth and number of entries instead of Block Ram for some less entries stages to increase the efficiency of memory usage. Table VI shows the FPGA resources utilization and throughput for 10K rule tables. All the schemes use the Xilinx Virtex-5 XC5VFX200T with -2 speed grade and Dual-Port memories. For 40 byte per packet,

Table VI. The FPGA resources utilization and throughput for 10K rule tables

Scheme	Rule Table	Slices Used/ Available	Block RAMs Used/ Available	Frequency (MHz)	Throughput (Gbps)
Our proposed scheme	**ACL1 10K**	898/30720 (3.2%)	95/456 (20.8%)	194.64	62.28
	IPC1 10K	924/30720 (3.0%)	121/456 (26.5%)	192.02	61.45
	FW1 10K	827/30720 (2.7%)	103/456 (22.6%)	196.95	63.05
Hyper-Cutting scheme [6]	**ACL1 10K**	3290/30720 (10.7%)	171/456 (37.5%)	161.76	50.55
	IPC1 10K	6041/30720 (19.6%)	318/456 (69.7%)	161.63	50.51
	FW1 10K	3613/30720 (11.7%)	269/456 (58.9%)	161.20	50.38
SPSTwB [7]	**ACL1 10K**	1576/30720 (5.2%)	182/456 (39.9%)	207.98	64.99
	IPC1 10K	1250/30720 (4.1%)	243/456 (53.3%)	208.46	65.14
	FW1 10K	623/30720 (2.1%)	114/456 (25%)	212.6	66.44

our throughput is about 60Gbps which can achieve OC-768 40Gbps. Our proposed scheme use about 3% Slices utilization and 20% Block Ram. A Block Ram unit showed in Table VI is 36 Kbit which contains the two smallest units. To compare with other two schemes, our scheme requires the least FPGA resources expect for FW 10K. Although the throughput of our scheme is a little smaller than SPSTwB [7], our scheme can implement the larger rule table than other two schemes. Hyper-Cutting [6] at most implement in ACL 50K, FW 25K, and IPC 20K. SPSTwB is ACL 30K, FW 15K, and IPC 20K. Our scheme can fit 50K for all the three tables, shown as Table VII.

In Table VII, we adopt original the four groups, classified in section 4.2, for the FW and IPC 50K because merging group 2 or 3 to group 1 would cause large rule duplication. The more search engine waste more Block Ram that IPC 50K uses more Block Ram than ACL 50K even if IPC 50K require less memory usage.

In Table VIII, we also compare our design with some existing schemes which use the Xilinx Virtex-5 XC5VFX200T device and dual port memory [3][14] [28][6][7]. Because some of them can't fit in IPC1_10K and FW1_10K, we only show the result for ACL1_10K rule table. The slices are quite sufficient for each approach. And the bottleneck of the rule table size is Block Ram utilization. Thus, we introduce a new metric, Efficiency, as the ratio of throughput and the number of used Block RAMs to have a fair comparison. For the efficiency, our approach is the best.

The Table IX shows the throughput comparison of many existing engines for packet classification. The second column lists the platform of each approach. Due to the rule table of each approach is not the same, we show number of rules used for each approach in third column. And the last column lists the throughput. Although throughput of our approach is

Table VII. The detailed performance for three rule 50K tables

		ACL1 50K	FW1 50K	IPC1 50K
Engine 1	Tree Memory (KB)	100.89	65.76	48.27
	Bucket memory (KB)	1006.23	777.11	590.16
Engine 2	Tree Memory (KB)	1.93	23.19	42.97
	Bucket memory (KB)	12.92	173.07	224.2
Engine 3	Tree Memory (KB)	0.13	82.43	13.76
	Bucket memory (KB)	0.79	630.66	73.24
Engine 4	Tree Memory (KB)	-	3	0.41
	Bucket memory (KB)	-	9.2	1.23
Total Memory usage(KB)		1122.88	1761.36	994.22
Slices Used/Available (utilization)		1476/30720 (4.8%)	2053/30720 (6.7%)	1270/30720 (4.1%)
Block RAMs Used/ Available (utilization)		393/456 (86.8%)	449/456 (98.5%)	439/456 (96.2%)
Frequency (MHz)		190.56	189.32	191.72
Throughput (Gbps)		60.98	60.59	61.35

Table VIII. The comparison in Virtex-5 environment with ACL_10K rule table.

Approaches	Slices Used/ Available	Block RAMs Used/ Available	Frequency (MHz)	Throughput (Gbps)	Efficiency (Throughput/ Block Rams)
Our approach	1796/30720 (6.4%)	95/456 (20.8%)	194.64	124.57	1.311
SPSTwB[7]	3152/30720 (10.3%)	182/456 (39.9%)	207.98	129.99	0.714
Hyper-Cutting scheme[6]	7044/30720 (22.9%)	173/456 (37.9%)	161.76	101.1	0.584
Two-dimensional Linear Dual-Pipeline[14]	10307/30720 (33.5%)	407/456 (89.2%)	125.4	78.37	0.192
BiConOLP[28]	6611/30720 (21.5%)	208/456 (45.6%)	143.4	44.81	0.215
SPMT[3]	6584/30720 (21.4%)	429/456 (94.1%)	173.02	108.14	0.252

smaller than SPSTwB, the efficiency of our approach is larger than SPSTwB, shown as Table 6-12.

VI. CONCLUSION

In this paper, we propose a scheme, named layer based search tree (LST), to solve multi-field packet classification problem. There are two phase, partition phase and classification phase. In partition phase, we map the given classifier into the binary decision tree by layering process. In classification phase, we collect all the leaf nodes of binary decision tree to construct the binary search tree. For each incoming packet, search in LST is immediately finished without searching the entire search tree if the packet matches

Table IX. The throughput comparison of many existing engines for packet classification.

Approaches	Platform	*# of rules*	*Throughput (Gbps)*
SPSTwB[7]	Virtex-5 XC5VFX200T	9,603	129.99
Our approach	Virtex-5 XC5VFX200T	9,603	124.57
SPMT[3]	Virtex-5 XC5VFX200T	9,603	108.14
Hyper-Cutting scheme[6]	Virtex-5 XC5VFX200T	9,603	101.1
Two-dimensional Linear Dual-Pipeline[14]	Virtex-5 XC5VFX200T	9,603	78.37
HyperSplit on FPGA[21]	Virtex-6 XC6VSX475T	9603	72.12
BiConOLP[28]	Virtex-5 XC5VFX200T	9603	44.81
B2PC in ASIC[19]	ASIC	3300	13.6
NTLMC[8]	Evaluate	12507	12.16
Power Saved HyperCuts[15]	Cyclone EP3C120F484C8 & Stratix EP3SE260F1152C47	25,000	10
BV-TCAM[24]	Virtex-E XCV2000E	222	10
2sBFCE[18]	Virtex-4 4vfx40ff672	4,000	1.88

the space covered by the node. To further improve the average classification speed in multiple groups, we use highest-priority variable to skip the search in some groups. Although software solution for packet classification problem is more flexible, the throughput is hard to keep pace with the rapid growth of Internet traffics that routers needs to achieve the high link rate such as OC-768 (40Gbps). Thus, we design the hardware search engine with pipeline and parallel architecture for the LST. Based on Xilinx Virtex-5 FPGA device, our search engine can support 50k rule table for ACL, FW, and IPC. Besides, we can achieve over 120 Gbps throughput with dual port memory.

REFERENCES

[1] H. J. Chao, "Next generation routers," *Proceedings of the IEEE*, vol. 90, no. 9, pp. 1518-1558, Sep. 2002.

[2] Y. K. Chang, "Efficient Multidimensional Packet Classification with Fast Updates," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 463-479, Apr. 2009.

[3] Y. K. Chang, Y.-S. Lin, and C.-C. Su, "A High-Speed and Memory Efficient Pipeline Architecture for Packet Classification," *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.215 - 218, 2010.

[4] Y. K, Chang and H.-C. Chen, "Layered Cutting Scheme for Packet Classification," The IEEE 25th International Conference on Advanced Information Networking and Applications (AINA-2011), 2011.

[5] Y, K. Chang and H.-M. Chen, "Set Pruning Segment Trees for Packet Classification," The IEEE 25th International Conference on Advanced Information Networking and Applications (AINA-2011), 2011.

[6] H. C. Chen, "Recursive Endpoint Based Hyper-Cutting Scheme For Packet Classification," Unpublished thesis for degree of master of computer science and information engineering, National Cheng-Kung University, Tainan, Taiwan, R.O.C, 2011.

[7] H. M. Chen, "Partitioned Set-Pruning Segment Trees For Packet Classification," Unpublished thesis for degree of master of computer science and information engineering, National Cheng-Kung University, Tainan, Taiwan, R.O.C, 2011.

[8] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast Packet Classification Using Bloom Filters," *Proc. ACM/IEEE ANCS*, pp. 61-70, 2006.

[9] P. Gupta, and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34-41, Jan.Feb. 2000.

[10] P. Gupta, and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24-32, Mar.-Apr. 2001.

[11] F. Geraci, M. Pellegrini, and P. Pisata, "Packet classification via improved space decomposition techniques," in *Proceedings of 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2005, pp. 304-312 vol. 1.

[12] V. Srinivasan, G. Varghese, S. Suri et al., "Fast and scalable layer four switching," in Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication, 1998, pp. 191-202.

[13] http://www.arl.wustl.edu/~hs1/PClassEval.html

[14] W. Jiang and V. K. Prasanna. "Large-Scale Wire-Speed Packet Classification on FPGAs," *Proc. FPGA*, pp. 219-228, 2009.

[15] A. Kennedy, X. Wang, Z. Liu, and B. Liu, "Low Power Architecture for High Speed Packet Classification," *Proc. ACM/IEEE ANCS*, pp. 131-140, 2008.

[16] T. V. Lakshman, and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, 1998, pp. 203-214.

[17] H. B. Lu and S. Sahni, "O(log W) multidimensional packet classification, " *IEEE-ACM Transactions on Networking*, vol. 15, pp. 462-472, Apr 2007.

[18] A. Nikitakis and I. Papaefstathiou, "A Memory-Efficient FPGABased Classification Engine," *Proc. IEEE FCCM*, pp. 53-62, Apr. 2008.

[19] I. Papaefstathiou and V. Papaefstathiou, "Memory-Efficient 5D Packet Classification at 40 Gbps," *Proc. IEEE INFOCOM*, pp. 1370-1378, 2007.

[20] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet Classification Algorithms: From Theory to Practice, " in Proceedings of the 28th IEEE Conference on Computer Communications (INFOCOM'09), 2009, pp. 648 – 656.

[21] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li and V. K. Prasanna, "Multi-dimensional Packet Classification on FPGA: 100 Gbps and Beyond," Proc. *Field-Programmable Technology*, pp. 241-248, Dec. 2010.

[22] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. in Proceedings of the ACM SIGCOMM'99 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99), 1999, pp. 135 – 146.

[23] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 213–224.

[24] H. Song and J. W. Lockwood, "Efficient Packet Classification for Network Intrusion Detection Using FPGA," *Proc. ACM FPGA*, pp. 238-245, 2005.

[25] D. E. Taylor, and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE-ACM Transactions on Networking*, vol. 15, no. 3, pp. 499-511, Jun. 2007.

[26] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238-275, Sep. 2005.

[27] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. "EffiCuts: Optimizing Packet Classification for Memory and Throughput," in *Proceedings of the 2010 conference on Applications, technologies, architectures, and protocols for computer communications*, 2010, pp. 207-218

[28] Jeffrey M. Wagner, Weirong Jiang and Viktor K. Prasanna, "A SCALABLE PIPELINE ARCHITECTURE FOR LINE RATE PACKET CLASSIFICATION ON FPGAS," *Proc. Parallel and Distributed Computing and Systems*, 2009.

[29] Xilinx, "Virtex-5 Family Overview", Product Specification, DS100 (v5.0), Feb. 6, 2009, at http://www.xilinx.com.