Hierarchical packet classification using a Bloom filter and rule-priority tries

A.G. Alagu Priya, Hyesook Lim *

Ewha Womans University, 11-1 Daehyun-Dong, Seodaemun-Gu, Seoul, Republic of Korea

1. Introduction

Packet classification [1] identifies flows among a stream of packets that arrives at routers. It is a fundamental building block that enables routers to support access control, quality of service differentiation, virtual private networks, and other value added services. To classify a packet, each packet arriving at a router is compared against a set of pre-defined rules. A rule consists of a set of fields, in which the most common fields are IP source prefix, IP destination prefix, source port number, destination port number, and protocol type and flags in the packet header. Each rule has an associated action, which is usually assigned to define its priority (or cost) among matched rules. Hence, packet classification system must compare the multiple header fields of the packet received on a link against a set of rules and return the identity of the highest-priority (or the least-cost) rule that matches the packet header.

In the past years, many algorithms have been proposed to attain an effective solution for packet classification [2]. TCAM based solutions are the one which offers consistently a high performance independent of the characteristics of the rule set. However, the cost and the high power consumption of TCAM made to explore some other algorithmic solutions. Recently a fast algorithm based on Bloom filters was proposed [3]. Generally, the Bloom filter is used to avoid lookup in some subsets which contain no matching rules and to make a possibility to sustain high throughput.

1.1. Our contributions

In this paper, we proposed a new algorithm based on a hierarchical approach. Generally, a two stage hierarchical approach may employ the same data structure of trie in both the stages to perform packet classification. A trie is a tree-based data structure that is used to store an associative array where the keys are either ‘0’ or ‘1’ to traverse. Trie comprises of many empty internal nodes. Our interesting approach is to perform the packet classification by combining Bloom filter and trie-based search. Since the Bloom filter can be implemented with an on-chip memory or a fast cache, the off-chip memory accesses are not occurred in the first stage of the hierarchical approach in the proposed algorithm while off-chip memory accesses are occurred in every stage in other hierarchical algorithms.
Though the querying process of the Bloom filter is the same as original Bloom filter, we proposed a new way of designing a Bloom filter which accommodates different lengths of prefixes in a single Bloom filter, which is termed as all-length Bloom filter (ALBF). This all-length Bloom filter is employed at the first stage of the hierarchy. The all-length Bloom filter is used to pre-filter the unnecessary prefix lengths of the source address of an input packet which have no match with the source prefixes included in a given rule set. At the second stage of the hierarchy, a new space-efficient trie called rule-priority trie (RPT) is employed to perform match in entire rule fields in a very efficient way. The RPT is constructed based on the destination prefix field, but each node of the RPT includes the complete rule fields so that input packets are compared with the entire fields of rules in the RPT. The RPTs are split into numerous small tries based on the source prefix representation in the ALBF. Hence, in the process of combining an ALBF and RPT, the Bloom filter not only finds the match for the source prefix in the first stage, but also acts as a pointer to the second stage tries. We further refined our search by setting threshold values for all the RPT, so that search can be avoided in a RPT when its threshold value is less than the priority of the already found match. Though ALBF shows a positive, some searches in the second stage RPT can be skipped if its threshold value is less than the already found match. The proposed algorithm supports incremental update of rules. The experimental result shows that our proposed algorithm provides a better performance than many other algorithms in terms of search performance and storage requirement.

The remainder of this paper is organized as follows. We discuss the related work in Section 2. We describe our all-length Bloom filter in Section 3 and our rule-priority trie in Section 4. Our proposed hierarchical packet classification algorithm is described in Section 5. In Section 6, we discuss about the simulation results. Section 7 concludes the paper.

2. Related work

In this section, we discuss various packet classification algorithms by grouping the algorithms into several categories depending on their approaches and characteristics. Most of the algorithms described in this section will be used in the performance comparison in a later section.

2.1. Trie-based algorithms

Hierarchical trie (H-trie) [4] first builds a source prefix trie, and each prefix node of the source trie hierarchically connects a destination prefix trie which comprises of rules with the same source prefix field. For a given input packet, the search is performed in the source prefix trie first. If there is a match with the source prefix, the search traverses to the corresponding destination prefix trie. If there is a match in the destination trie, the rule number is noted down and the search needs to be continued from the node of the source trie from where it traversed to the destination trie. The search is continued until there are no more nodes to proceed in the source trie, keeping track of the highest-priority matching rule. Table 1 shows an example rule set. The H-trie corresponding to the rule set in Table 1 is shown in Fig. 1.

The big trie is the source prefix trie and multiple small tries are destination tries. Dark nodes represent prefix nodes or rule nodes, and white nodes represent empty internal nodes which are not associated with a prefix or a rule. While searching, all the destination trie connected to every matched node of the source trie should be visited in order to determine the highest-priority rule, and this kind of search procedure is called back-tracking. The back-tracking causes the excessive number of memory accesses. The search complexity depends on the number of destination tries visited, and the memory requirement depends on the number of nodes. The search complexity is $O(W^2)$ and the memory requirement is $O(NW)$, where $N$ is the number of rules and $W$ is the maximum length of the prefixes.

The H-trie includes many empty nodes in the path to a prefix node (or a rule node). The empty nodes waste memory space as well as deteriorate the search performance by causing unnecessary memory accesses. Hierarchical binary search tree (HBST) [5] algorithm is the same as the H-trie except that HBST replaces the trie structure in H-trie with the tree structure which does not include empty internal nodes. In HBST, by applying a set of definitions to compare the magnitude of prefixes with different lengths, binary search trees without empty internal nodes are constructed. Each node in the binary search tree for the source prefix field connects hierarchically the destination binary search tree for packet classification. Hence HBST shows better performance than H-trie in search speed and the memory requirement.

To avoid the back-tracking of the H-trie, hierarchical set-pruning trie [4] copies all the rules included in the ancestor nodes of a descendant node in the source trie into the destination trie of the descendant node. By this set-pruning, all possible matching rules are gathered into the destination trie of the descendant node. While searching, the longest matching prefix in the source trie

<table>
<thead>
<tr>
<th>Rule</th>
<th>Src prefix</th>
<th>Dest prefix</th>
<th>Src port</th>
<th>Dest port</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>010*</td>
<td>10*</td>
<td>0, 65535</td>
<td>25, 25</td>
<td>6</td>
</tr>
<tr>
<td>R1</td>
<td>101*</td>
<td>001*</td>
<td>53, 53</td>
<td>443, 443</td>
<td>4</td>
</tr>
<tr>
<td>R2</td>
<td>*</td>
<td>10*</td>
<td>53, 53</td>
<td>1024, 65535</td>
<td>17</td>
</tr>
<tr>
<td>R3</td>
<td>*</td>
<td>01*</td>
<td>53, 53</td>
<td>443, 443</td>
<td>4</td>
</tr>
<tr>
<td>R4</td>
<td>1*</td>
<td>1*</td>
<td>53, 53</td>
<td>25, 25</td>
<td>4</td>
</tr>
<tr>
<td>R5</td>
<td>1101*</td>
<td>001*</td>
<td>0, 65535</td>
<td>2788, 2788</td>
<td>17</td>
</tr>
<tr>
<td>R6</td>
<td>110100*</td>
<td>11*</td>
<td>53, 53</td>
<td>5632, 5632</td>
<td>6</td>
</tr>
<tr>
<td>R7</td>
<td>*</td>
<td>11*</td>
<td>53, 53</td>
<td>25, 25</td>
<td>6</td>
</tr>
<tr>
<td>R8</td>
<td>111*</td>
<td>01*</td>
<td>67, 67</td>
<td>5632, 5632</td>
<td>17</td>
</tr>
<tr>
<td>R9</td>
<td>010*</td>
<td>10*</td>
<td>0, 65535</td>
<td>0, 1023</td>
<td>6</td>
</tr>
<tr>
<td>R10</td>
<td>111*</td>
<td>1*</td>
<td>67, 67</td>
<td>25, 25</td>
<td>4</td>
</tr>
<tr>
<td>R11</td>
<td>1101*</td>
<td>001*</td>
<td>53, 53</td>
<td>2788, 2788</td>
<td>4</td>
</tr>
<tr>
<td>R12</td>
<td>111*</td>
<td>*</td>
<td>1024, 65535</td>
<td>5632, 5632</td>
<td>4</td>
</tr>
</tbody>
</table>
is determined and the search traverses to the corresponding destination trie which has all the possible matching rules in it. Therefore, the back-tracking is avoided. The search performance is improved from $O(W^2)$ to $O(W)$. However, because of the rule duplication, the memory requirement is $O(N^d)$ for $d$-dimensional classification.

Grid-of-Tries [6] overcomes the drawback of the back-tracking and the memory overhead by constructing the trie using the pre-computation of switch pointers. In Grid-of-Tries, when there is no node to traverse in the destination trie, a pre-computed switch pointer will lead to a node, which may have the next possible match in the next ancestor’s destination trie. However, Grid-of-Tries skips over all rules in the next ancestor’s destination trie whose destination fields are shorter than the current destination match. In other words, the switch pointer is computed in such a way that some rules could not be traversed, since pointer jumps to a lower level in the next destination trie. Hence, it is assumed that a rule with a longer destination prefix length has a higher priority than a rule with a shorter destination prefix length. In other words, Grid-of-Tries is restricted only for 2-dimensional (2-D) classifiers, since it fails to traverse all possible matching rules. On the whole, the search performance of Grid-of-Tries is $O(W)$ and the memory requirement is $O(NW)$ by avoiding back-tracking and rule copying, but it does not provide the incremental update of rules because of the pre-computation of switch pointers.

Area-based quad-trie (AQT) [7] recursively partitions a 2-dimensional plane composed of source and destination prefixes, and each partitioned area is mapped to a quad-trie node. Hence, AQT is a 2-dimensional trie which examines both the source and the destination prefixes at the same time. The entire plane is mapped to the root node of the quad-trie, and four equal-sized areas partitioning the entire plane are mapped to four children of the root node, and so on. Each rule in a given rule set can be expressed as a rectangle in the 2-dimensional plane. Rules in a crossing filter set (CFS) of an area are defined as the rules included in an area, among which one of its dimension entirely crosses the area. Rules included in a CFS are stored into the corresponding quad-trie node. For a given input packet, starting from the root node, by examining the first bits of the source and the destination addresses of the input packet, search moves to one of its children. If the search encounters a node storing rules, input headers are compared with the corresponding fields of the stored rules.

Priority-based quad-trie (PQT) [8] algorithm is based on the AQT algorithm, but empty nodes in the AQT are completely avoided. By relocating the highest-priority rule among rules included in the sub-trie rooted by each empty node into the empty node, PQT completely removes empty nodes. The relocated rule is named a priority rule. Search procedure in the PQT is basically same as the AQT, but it is more efficient since search can be finished if a given input header matches a priority rule in a node. Since it is known that the priority rule has the highest priority among the rules included in the sub-trie of the node, if the input matches the priority rule, it is the highest-priority rule matching the input. Hence search does not need to be continued in lower levels.

Binary search on prefix length (BSL) was applied first by Lim and Mun for packet classification [9]. Since the binary search on length cannot be directly applied to tries with prefix nesting relationship, the algorithm splits the area-based quad-trie (AQT) into multiple quad-tries based on prefix nesting levels so that prefixes in each quad-trie do not have the prefix nesting relationship. In performing the binary search on length, to utilize the heuristics in real packet classification database, in which the longer in prefix length has the higher in priority, the quad-trie of prefixes with the longest length is searched first in the algorithm.

2.2. Bit-vector based algorithms

Bit-vector algorithm [10] first searches for rules that match each field of a given input packet and represents the result of the search as a bit-vector of rules. Then the rules that match the full header can be found by taking the intersection of the bit-vectors for all relevant fields. The bit-vectors have $N$ bits in length, where $N$ is the number of rules in a given rule set. If $M$ is the size of a word of memory, the bit operations are responsible for $(N \times d)/M$ memory accesses in the worst case, where $d$ is the number of header fields. Aggregated bit-vector algorithm [11] adds two new ideas to the bit-vector algorithm: recursive rule aggregation of bit-vectors and rule rearrangement to reduce the size of bit-vectors based on the fact that the set bit in the bit-vector produced by each field is very sparse.

2.3. Cutting based algorithms

Cutting based algorithms [12,13] partition a multi-dimensional space composed of each rule field based on the heuristics of rules in a given rule set, and each partitioned space is mapped to a node of a decision tree. For constructing the decision tree, HiCuts uses a local optimized decision at each node in determining the dimension (or field) of the cuts and the number of cuts to be made in the chosen dimension [12]. The criterion is to balance the storage requirement and search speed. The HiCuts algorithm uses two parameters, $binfo$ and $spfic$, in tuning the heuristics, which trade off the depth of the decision tree against the memory space. For balancing the storage with time, a small amount of linear searching is established. The $binfo$ is the pre-determined number of rules included in the leaf nodes of the decision tree for linear searching. The field in which a cut may be executed is chosen to minimize the maximum number of rules in any partition, and the $spfic$, a function of space measure, is used in determining the number of cuts.

While HiCuts algorithm only considers one field at a time in selecting the dimension of cuts, HyperCuts [13] algorithm considers multiple fields at a time. In selecting the dimension of cuts, the ratio of the number of distinct elements to the total number of possible values representing the dimension is considered. If $C_i$ is the number of cuts in field $i$, the number of all possible cuts is $\prod C_i$, for $i = 1 \ldots d$, where $d$ is the number of header fields. When compared with HiCuts decision tree, the decision tree of HyperCuts has smaller depth, as multiple fields are used at the same time in a single node, and thus, the search speed is improved. However, the number of entries is more in HyperCuts, due to the generation of many unnecessary entries. Several techniques have been proposed to refine or optimize the algorithm, such as node merging, rule overlapping, region compaction, and pushing common-rule subsets upwards [13]. These refinements cause pre-processing overhead.

2.4. Tuple space based algorithms

In tuple space algorithm [14], each rule in the rule set is specified as a pair of prefixes. The lengths of the prefix pair in the rule is defined as tuple and denoted as $(i, j)$, where $i$ is the length of the source prefix and $j$ is the length of the destination prefix. Even though there could be 1089 different tuples, which is 33 multiplied by 33, there is more possibility for many empty tuples. A packet arriving at a link is queried with all the non-empty tuple spaces by extracting $i$ bits from source IP address and $j$ bits from destination IP address. Since it is not practical to afford that many queries in all the existing tuples to classify a packet, a simple optimization called tuple space pruning [14] was introduced. In the tuple space pruning, the longest prefix matching (LPM) lookup
is performed first to reduce the number of tuple spaces to be searched. By performing the LPM for the source IP address, if it is known that there is no match for the source IP address at length 'i', there is no need to perform tuple space search with any of the tuples that has the source prefix length as 'i'.

Coarse-grained tuple space [15] algorithm partitions the tuple space into subsets that are close in tuple space and use the information from single field lookups to limit the number of subsets that must be searched. For the longest prefix matching for the single field lookups, a new algorithm combining the multi-bit trie with multi-level jump tables was proposed. Chained path Bloom filters are used to jump fast the search to the target segment when doing the longest prefix matching.

2.5. Cross-product based algorithms

In the cross-product algorithm [6], the rule database is sliced into columns, with the ith column storing all distinct prefixes in field i. A cross-product table composed of all possible cross-product is built. The highest-priority rule matching each cross-product is pre-computed. Given a packet P, the LPM for each field is determined separately, the results of the LPM lookups on individual fields are combined, and a single compound lookup is performed in the cross-product table. Hence the highest-priority matching rule is determined by d LPM operations plus a single direct lookup of the cross-product table, where d is the number of fields. However, this algorithm suffers from a memory explosion problem: the cross-product table can have \( N^d \) entries in worst case, where N is the number of rules and d is the number of fields.

In fast packet classification using Bloom filter [3], rules in a set are split into non-overlapping multiple subsets in order to reduce the required number of rules for the cross-product table. The LPM operation using Bloom filters is done separately on each field against a pre-computed bit-vector table, each entry of which indicates the lengths of the prefixes included in each subset. The result is used to form a set of tuples, each of which indicates how many bits should be used for constructing keys corresponding to that subset. The keys are queried in Bloom filters first and the keys matched in Bloom filters are used to probe the hash table of corresponding rule subset kept in the off-chip memory. In this algorithm, pre-computation is required to construct non-overlapping multiple subsets and the corresponding bit-vector table, and hence the algorithm does not provide incremental update. Each LPM operation of the algorithm requires at least an off-chip memory access, and the rule table lookup for the subsets turned out to be positive also requires several off-chip memory accesses. Moreover, every distinct combination of source prefix and destination prefix requires a Bloom filter. The total number of Bloom filters can be large and the item distribution can be highly skewed. In reducing the total number of Bloom filters, the algorithm proposed to use prefix-expansion, but the prefix expansion deteriorates the performance in terms of memory requirement and incremental update.

3. All-length Bloom filter

In this section, we briefly describe the theory behind the Bloom filter and our unique method of designing all-length Bloom filter. Bloom filter conceived by Burton H. Bloom [16] in 1970, is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. A Bloom filter is basically a bit-vector of length m, which represents the membership of a set of elements.

IP address lookup algorithms using Bloom filter [17,18] assign each distinct prefix length a Bloom filter. Hence, when programming a Bloom filter, a set of hash functions which is compatible for the corresponding distinct length of prefixes is selected and programmed. Each distinct prefix length requires a Bloom filter, and hence the total number of Bloom filters can be large. Although the total number of elements programmed in these Bloom filters is equal to the total number of prefixes, the item distribution among these Bloom filters can be highly skewed. Our interesting approach is to design a Bloom filter which can accommodate various lengths of prefixes in it.

3.1. Hash generator

The all-length Bloom filter accommodates prefixes of different lengths in it, and hence it requires a hash generator which generates hash indices for various-length inputs. Any random hash generator which provides multiple hash indices for an input with an arbitrary length can be used. As a universal hash generator for any prefix lengths, CRC generator has excellent characteristics to be used as a hash generator for our purpose. CRC generators scramble bits of a given input and produce a fixed-length binary sequence known as a CRC code regardless of the input length. Any number of hash indices can be easily obtained from the generated CRC code by selecting several different combinations of bits [19]. CRC generators are simple to implement in hardware and easy to analyze mathematically. Hence it is well suited for the hash generator for our proposed all-length Bloom filter which requires multiple hash indices for a given input with an arbitrary length.

3.2. Programming

An all-length Bloom filter representing a set \( S = \{x_1, x_2, \ldots, x_n\} \) of n elements which may be of different lengths, is described by an array of m bits, initially all set to 0. Once an element in the set \( S \) enters to the CRC generator one bit at a time, a fixed-length CRC code is obtained. From the CRC code, by selecting k different combinations of bits, k hash indices to program the Bloom filter for that specific element are obtained. The k different hashing indices are obtained in such a way that the resulting hash index \( j' \) is of the range \( 0 \leq j < m \). All the bit-locations corresponding to k hash indices are set as ‘1’ in the Bloom filter. On programming all the elements in the set \( S \), some elements may result in repeated occurrence of an index. In other words, if a bit-location was already set as ‘1’ and if the same hash index is resulted for another element, that bit-location is not altered.

In the proposed packet classification algorithm, all the distinct source prefixes in a given rule set are identified and programmed to an all-length Bloom filter. For the example rule set shown in Table 1 in Section 2, assuming that the maximum size of the source prefixes is 6 for simplicity, a CRC-6 generator is used as a hash generator in this example. The block diagram of the CRC-6 generator is shown on Fig. 2 [20]. For every distinct source prefixes with non-zero length shown in Table 1, Table 2 shows the generated CRC codes and the corresponding two 4-bit hash indices, which are chosen from the first 4 bits and the last 4 bits of the CRC code.

Fig. 3 shows our 16-bit all-length Bloom filter programmed by two hash indices for the distinct source prefixes given in the example rule set. After programming all the distinct source prefixes, the distinct lengths of prefixes that were programmed into the all-

![Fig. 2. Block diagram of the CRC-6 generator.](image-url)
length Bloom filter, which is \{1, 3, 4, 6\} in this example, are noted down.

### 3.3. Querying

To test the membership of an input element from the Bloom filter, the input is entered to the same CRC generator at a bit at a time, and a fixed-length CRC code is generated. K hash indices are obtained from the same combinations of bits of the CRC code which were used in programming. Bit locations in the Bloom filter corresponding to the hash indices are checked. If at least any one of the locations was set only by that element under testing and there is a possibility to encounter a ‘0’. Hence, it is perfectly justified that it is not a member of the set. On the other hand, if all the hash value locations were set as ‘1’, it does not mean that all those bit-locations were set only by that element under testing and there is a possibility that those locations would have been set by some other elements in the set. This type of positive result is termed as “false positives.” Considering the above fact, we come to a conclusion that when all the bits-locations of the hash indices are ‘1’, it may be a member of the set S. On the whole, Bloom filter may produce false positives but not false negatives. By properly controlling the false positive rate of the Bloom filter, the Bloom filter at the first stage of the hierarchical packet classification can filter out effectively the source prefix lengths that do not have matches in a given rule set.

In the proposed packet classification algorithm, for a given input packet, all the possible source prefix lengths are queried to the all-length Bloom filter. For example, assuming a given input packet (110101, 101000, 53, 25, 4), the source address ‘110101’ is considered. The distinct prefix lengths are \{1, 3, 4, 6\} which was noted during programming the Bloom filter. For the longest distinct length 6, ‘110101’ is fed into the hash generator and the generated CRC code is 001001. Two hash indices, which are 2 and 1, are checked in the Bloom filter of Fig. 3. It gives a negative result. For the next distinct length 4, which is the bits ‘1101’, the generated CRC code is 100011 and the hash indices are 8 and 3. The Bloom filter gives a positive result. For the next distinct length 3, which is the bits ‘110’, the generated CRC code is 100100 and the hash indices are 9 and 4. The Bloom filter gives a negative result. Next, for length 1, which is the bit ‘1’, the CRC code is 110001, and the hash indices are 12 and 1. It is a positive result. The reason to examine the prefixes from the longest to the shortest will be discussed in a later section.

On the whole, for a given source address 110101, among four possible source prefixes, two prefixes, 110101 and 110, turned out to be negative, and the prefixes with negative results are filtered out. No further comparison is required for these prefixes. A serial search for all the existing prefix lengths is required for a given input. An alternative method is proposed by Song et al. [21] to overcome the drawback of serial searches by performing a parallel search, and the method increases the speed of the algorithm and achieves a high throughput. However, to perform the parallel search, a single Bloom filter must be partitioned into equal-sized distributed and load-balanced Bloom filters, and a set of hash functions must be assigned for each unique length prefix groups. Moreover, while the Bloom filters proposed in [21] require large amount of memory to control the false positive rate close to zero, the proposed all-length Bloom filter requires a small amount of memory which can be stored in an on-chip memory or a fast cache, since the Bloom filter only works as a pre-filter in the proposed approach. Serial searches to an on-chip memory or a fast cache do not deteriorate the overall search performance. Hence we employ an all-length Bloom filter without any partitioning and also use a single CRC hash function for implementation simplicity.

### 4. Rule-priority trie

As described in Section 2, most of the trie-based packet classification algorithms have many empty internal nodes which waste memory spaces and memory accesses. The trie-based algorithms compare a given input with shorter prefixes of the rules first and even if a match was found, the search has to be continued until a leaf is visited, since there may be a possibility for a higher-priority rule at lower levels. The priority trie proposed for IP address lookup removes empty internal nodes by replacing the longest prefix among the prefixes belonged to the sub-tree rooted by each empty node [22].

Our proposed rule-priority trie (RPT) is based on the priority trie, but we relocate the highest-priority rule instead of the longest-prefix rule. The relocated rule is termed as a priority rule, and a rule stored in its own level is termed as an ordinary rule. RPT can be constructed based on either the source prefix field or the destination prefix field, and nodes in the RPT will have information about entire rule fields. On searching, a match with a node is determined if it has prefix match with the source and the destination prefixes, exact match with the protocol, and range match with the source port and the destination port. Search in the proposed algorithm is immediately finished without searching the complete trie if an input packet matches a priority rule. This property effectively improves the search performance.

#### 4.1. Building RPT

The proposed RPT can be constructed by using the method of insertion and pushing repeatedly. Assuming that RPT is constructed based on destination prefixes, the destination prefixes and the corresponding priorities are used to determine the path of the trie. Initially, for the first rule, its priority information and the complete rule are inserted at the root node. To insert the second rule, its priority is compared with the root node, and if it has a higher-priority than the rule in the root node, it replaces the root node rule. The previous rule at the root node finds its new position among the prefixes belonged to the sub-tree rooted by each empty node [22].
On inserting a new rule into the trie, if the destination prefix reaches its own position, in which the node position corresponds to the destination prefix, even if the node had a higher-priority rule, the rule has to be replaced. In other words, when the original destination prefix of that corresponding node enters the trie, it is not an empty node any more, and hence the higher-priority rule has to be replaced and find the next possible position. If more than a rule is mapped to the same node, they are assumed to be connected by a linked list in the order of decreasing priority.

For the example rule set in Table 1, the proposed RPT, which is constructed based on the destination prefix field, is shown in Fig. 4. In Fig. 4, all the rules are stored in the trie in a sequential order of increasing source prefix and decreasing destination prefix. The rules R1 to R11 are shown in the example rule set. The proposed hierarchy is shown using destination prefixes. Rule R5 (1101*, 001*) is the best match of the current input. The rule R10 (111*, 1*) is not necessarily compared since it has a lower priority than the current best match. The rule R10 which is stored in the same node is not necessarily compared since it has a lower priority than the current best match. The search moves to the left child since the second bit of the destination address is 1, search moves to a node, the input packet does not match R12 in port numbers. Since the stored rule R0 and R2 have higher priorities than the current best match, the input is compared with them. The input does not match them. The rule R9 is not compared since it has a lower priority. Now the search is over and the best match of the current input is R4.

The search procedure in the proposed RPT is basically same as the search in the binary trie. Search proceeds to the left or right according to the sequential inspection of destination address bits starting from the most significant bit. If there is a match with all the fields in a node, it is considered as a match and its priority number is remembered. Search will be stopped immediately in case if it ends with a match with a priority rule. Search in the proposed algorithm is finished either at a match with a priority rule or at a leaf while it is always finished at a leaf in other trie-based algorithms.

For example, assume a given input packet (110101, 101000, 53, 25, 4), the destination address ‘101000’ is considered. At the root node, the input packet does not match R12 in port numbers. Since the first bit of the input destination address is 1, search moves to a right child of the root node. The input is compared with R4 and ends up with a match. Hence R4 is remembered as the current best match. The rule R10 which is stored in the same node is not necessarily compared since it has a lower priority than the current best match. Search moves to the left child since the second bit of the destination address is 0. Since the stored rule R0 and R2 have higher priorities than the current best match, the input is compared with them. The input does not match them. The rule R9 is not compared since it has a lower priority. Now the search is over and the best match of the current input is R4.

The proposed RPT itself can be used for packet classification. However, the RPT is constructed based on a single field. If more than a rule has the same prefix in that field, the rules are mostly likely to be stored in the same node if they are not stored in other nodes as a priority rule. Those rules are linearly compared. The linear search in a node deteriorates the search performance. In our proposed hierarchical packet classification algorithm, many small RPTs are constructed as the same as the destination tries in the H-trie, and hence the search is more efficient than that in a single RPT.

5. Our packet classification approach

Our hierarchical packet classification approach follows the observation from real databases that any packet matches only a small number of distinct source-destination prefix pairs [23]. For a given input packet, if rules are filtered out using the source and destination addresses, only a small number of rules are left as possible matching candidates. A linear search to the set of matching candidates for complete rule fields can be performed. In this way, the port range specifications stay as ranges without the blowups associated with range translation.

Our proposed approach follows the hierarchical approach of packet classification but has an efficient combination of different data structures, while the same data structures are employed in other hierarchical approaches. In our proposed hierarchical approach, a Bloom filter is used for the source prefix field to filter out rules that do not match to a given input address, and rule-priority tries constructed as the basis of the destination prefix trie are used to determine the best matching rule in entire fields. Since the Bloom filter searches can be performed in an on-chip memory or a fast cache, off-chip memory accesses are occurred only for the second stage of hierarchical approach. The proposed rule-priority trie for the second stage is an efficient data structure in both the search performance and the memory requirement, in which it compares with the higher priority rules earlier than the lower priority rules and avoids empty internal nodes.

As described in earlier section, the hierarchical trie (H-trie) examines all the destination tries which are connected to every matching prefix of the source trie. The set-pruning trie avoids this by rule duplication. Our proposed packet classification approach can follow either the H-trie approach or the set-pruning approach. In this paper, we show the packet classification following the H-trie approach, in which every possible length for a given input source address is examined in the ALBF to determine whether the specific length of the source address has a possible match in the rule set. For all the lengths which have a positive result from the ALBF, a rule-priority trie (RPT) is examined to determine whether the rule with the source prefix matches the given input in entire rule fields. Since our proposed all-length Bloom filter (ALBF) used for the source prefix field has a simple structure and requires a small amount of memory, it can be implemented by an on-chip memory and provides fast result. If we follow the set-pruning approach, the longest prefix length matching to a given source address needs to be searched out from the ALBF, and the best matching rule will be determined from

![Fig. 4. Proposed rule-priority trie constructed using destination prefixes.](image-url)
the RPT of the longest matching prefix. In this case, the search performance will be improved at the cost of memory requirement.

5.1. Building

There are two factors affecting the performance of a Bloom filter: the size of the Bloom filter and the number of hash indices. To determine the size $m$ of the ALBF, the number of non-wild source prefixes $n$ in the rule set is counted and it is rounded to the nearest two powered value ($2^z$). In other words, the size of the Bloom filter is $2^z$, where $2^{z-1} < (n \times i) < 2^z$, where $i$ is a multiplication factor and $z > 0$. If the size $m$ of the ALBF is to be incremented as two times as $n$, the multiplication factor $i$ is set as 2 and the size $2^z$ is determined.

For each source prefix, $k$ different hash indices of $z$ bits should be chosen. Throughout all our experiments, two hash indices are used to program and probe the ALBF. The packet classification performance depending on the number of hash indices is beyond the scope of this paper. In selecting two hash indices from a CRC code, we consistently selected the first $z$ bits and the last $z$ bits for two hash indices, $k_1$ and $k_2$. The bit-locations $k_1$ and $k_2$ in the ALBF are set as 1. The complete rule is stored into the RPT-$k_1$, in which $k_1$ indicates the position of the RPT to be stored. Hence the RPT-$k_1$ has all rules which map to the bit-location of $k_1$ by the first hash index of its source prefix field. After all the distinct source prefixes are programmed to the ALBF and all the rules are stored into the corresponding RPT, all the distinct lengths of source prefixes stored in the ALBF, which is $m = \{u_1, u_2, \ldots, u_n\}$, are noted down, where $n$ is the number of distinct lengths.

Since a rule is inserted into a particular RPT based on the $k_1$ value generated by the source prefix, the number of RPTs can be equal to the number of set bits of the ALBF in worst case. Since the rules with source prefix as wild-card (*) cannot be programmed in the ALBF, a separate RPT is created for those rules, and it is termed as RPT-wild. After constructing every RPT, we denote the threshold value for each RPT, where the threshold value of a RPT is equal to the highest-priority rule included in that RPT.

The basic architecture for the proposed packet classification algorithm is shown in Fig. 5, which is constructed for the example set in Table 1. Since the number of rules with non-wild source prefix is 10, the size of the Bloom filter is selected as 16, in which the multiplication factor $i$ is 1. The remaining three rules with a wild source prefix are inserted in the RPT-wild. All those bit-locations were initially set as 0. Entire rule fields are stored in each node of rule-priority trie, but for simplicity, the nodes are represented in the format of “rule number (source prefix, destination prefix)”. Dark nodes are the ordinary nodes and white nodes indicate the priority nodes. The distinct lengths of the source prefixes programmed in the Bloom filter, which is $\{1, 3, 4, 6\}$, are noted down.

If we compare the H-tree shown in Fig. 1 with the proposed packet classification architecture shown in Fig. 5, the source prefix trie of the H-tree is replaced by an ALBF. We can find out one-to-one correspondence between the destination tries of the H-tree and the RPTs of the proposed architecture. For example, referring the hash indices shown in Table 2, the destination trie of a source prefix ‘111’ is the RPT-5 and the destination trie of a source prefix ‘1101’ is RPT-8, and so on. Two different source prefixes, ‘010’ and ‘110100’, have the same $k_1$ value, and hence two destination tries in the H-tree are merged into one RPT, which is RPT-14. Each empty node in the destination tries of the H-tree is replaced by a rule with the highest priority in the sub-trie rooted by the empty node.

5.2. Searching

To find the best matching rule for an incoming packet, the 32-bit input source address is considered first. From the source address, the longest distinct length of source prefix existing in the ALBF, which is $u_i$ bits, is extracted. The extracted bits are serially entered to the CRC generator. From the CRC generator, a fixed-length CRC code is generated and two hash indices $k_1$ and $k_2$ are selected. If any or both of the bit-locations $k_1$ and $k_2$ are set as ‘0’ in the ALBF, there is no match for the sub-string with the length $u_i$ of the source address, and hence there is no need to search for rule match against the rules stored in RPT.

If both the locations $k_1$ and $k_2$ are set as ‘1’, it is considered that there may be a match for the sub-string with the length $u_i$, and hence the search proceeds further to find the match with all the five fields. As we programmed, if the sub-string produces hash indices $k_1$ and $k_2$, the corresponding rule will be stored in the RPT-$k_1$. Therefore, the rule search is performed in the RPT-$k_1$ and if there is a match, it is noted. Once a RPT is accessed, then that trie will be disabled, which means hereafter that RPT should not be accessed to find a rule match for the current input packet.

The search procedure is repeated for the sub-string of the source address with the next smaller distinct length, and if the Bloom filter gives a positive result, the search will be continued in the corresponding RPT-$k_1$. In case if the RPT was searched for a match before, it is not accessed, since it has been disabled. In case if the RPT was not accessed so far, before start searching in that trie, the threshold value of that trie, which is the highest-priority rule number included in that trie, is compared with the already found match. If the already found match has a higher priority than the threshold, that trie is not necessarily searched. The reason is that even if the Bloom filter output may be a true positive and hence we end up with a match in that trie, it is not going to be a higher-priority rule than the already found match which was inferred from the threshold comparison. Hence, it is not necessary to access the trie and the trie is disabled.

This procedure is repeated for all the distinct lengths of prefixes from the longest to the shortest in the ALBF. Since the Bloom filter and all RPT represents only the rules whose source prefix length is longer than zero, we need to search for the possibility of a higher-priority rule in RPT-wild. The RPT-wild is accessed only if its threshold value has a higher priority than the already found match. Finally, the best matching rule (BMR) for the input packet is determined.
The reason to examine the sub-string of the input source address from the longest to the shortest is as follows. If the longer source prefixes have the higher priorities, which is likely to be true for real rule databases, more RPT search will be skipped by the threshold comparison, and hence the search speed will be accelerated. In case that there are many rules with a wild-card in both the source and the destination prefix fields, the rules are mapped to the root node of the RPT-wild and stored by a link-list in the order of decreasing priority. Since those rules generally have low priorities, the comparison on those rules is avoided by the priority comparison with the already found match. Hence the search performance is not deteriorated for the rule set with many wild-card rules if a rule set follows the general heuristics.

In this algorithm, unlike the standard Bloom filter, the proposed ALBF is not only used to find the membership queries, but also acts as a pointer to the second stage of searching and all the different lengths of prefixes are accommodated in a single Bloom filter. The search procedure can be described by a pseudo-code as follows:

Search (in_packet, src_addr, src_length) {
// src_length is a 32-bit vector which was set for the distinct source lengths in the building procedure
BMR = N-1;
// (N-1) is the index of the lowest priority rule
for (i = 31;i >= 0;i--) {
  if (src_length[i] == 1) {
    // (i+1)-bit prefix of the source address exists
    in the rule set
    crc_code = crc_gen (src_addr, i+1);
    // (i+1)-bit source address is entered to CRC generator and a 32-bit crc_code is returned
    k1 = crc_code[0:21]; the first 21-bit index
    k2 = crc_code[22:31]; the secondz-bit index
    ALBF_ret = probe_ALBF (k1, k2);
    if ((ALBF_ret == POS) && (RPT[ALBF_ret].mode==DISABLE))
      & & (RPT[ALBF_ret].threshold < BMR))
      BMR = search_RPT(in_packet, k1, BMR);
    RPT[ALBF_ret].mode = DISABLE;
  }
}
if (RPT-wild-ptr < BMR)
  BMR = search_RPT(in_packet, RPT-wild-ptr, BMR);
return BMR;
}

For a search example, assume an input packet (110101, 101000, 53, 25, 4). The src_length was set as 10101 during the building procedure, which means the given rule set has the distinct source prefix lengths 6, 4, 3, and 1. For the source address ‘110101’, we have positive result on lengths 4 and 1 as described in Section 3. For the 4-bit source prefix 1101, the hash indices are k1 = 8 and k2 = 3, and hence RPT-8 is searched. The input packet does not match R5, which is stored at the root node of RPT-8. The first bit of the destination address is 1, but there is no right child in RPT-8, and hence search in RPT-8 is over. The RPT-8 is now disabled. For the 1-bit source prefix 1, the hash indices are k1 = 12 and k2 = 1, and hence RPT-12 is searched, where it ends up with a rule match of R4. The RPT-12 is now disabled. Finally, the current best match R4 is compared with the threshold value of RPT-wild, which is R2. Since R2 has a higher priority than the current best match R4, the search in the RPT-wild is performed. The input packet does not match R2 and search moves to the right child of R2. Since R7 has a lower priority than the current best matching rule, which is R4, the comparison is avoided. Finally, R4 is returned as the best matching rule.

6. Simulation results

Simulations using C++ have been performed for rule sets created by Classbench [24–26]. Since rule sets created by the Classbench are known to have the characteristics similar to actual rule sets used in backbone routers, it is widely used in evaluating the performance of packet classification algorithms [2,3,5,8,9,15]. Three different types of rule sets, access control list (ACL), firewall (FW), and IP chain (IPC), are created with sizes of about 1000 and 5000 rules, each. The basic block diagram of a 32-bit CRC generator to provide hash indices for all-length Bloom filter (ALBF) is shown in Fig. 6 [20]. For any length of input prefix, universally a 32-bit CRC code is available. From this 32-bit CRC code, by selecting some combination of bits, any number of hash indices is easily generated [19].

The performance of Bloom filter is highly dependent on the number of hash functions and the size of Bloom filter when a fixed number of elements are given. The main motivation of this paper is to propose a new hierarchical approach and to show the implementation feasibility of our proposed approach. Hence finding the optimal Bloom filter in terms of the number of hash functions and the size of Bloom filter is beyond the scope of this paper.

The performance of the ALBF was primarily evaluated by increasing the multiplication factor, I, as 1, 2, and 4. For all the cases, only two hash functions are used for simplicity. The experimental results are shown in Tables 3 and 4. As shown in the memory requirement, the required amount of memory for ALBF is just sev-
eral hundred bytes, which is very small. The false positive rate of the Bloom filter has the trade-off with the required memory amount for the Bloom filter. While a large amount of memory is used in [21] to control the false positive rate close to zero, our approach is to achieve a high throughput with comparatively a lower memory requirement. Our proposed approach requires 20–32 times less memory than the Bloom filters required in [21].

The experimental results show that as increasing the size of the Bloom filter, the percentage of negative result is increased and the percentage of false positive result is decreased. In these tables, the percentage of true results is the summation of the percentage of negatives and the percentage of true positives (which was confirmed through RPT search).

The false positive rate highly depends on the characteristics of rule sets. ACL1k and ACL5k rule sets have many rules with a 32-bit source prefix field, and hence the CRC provides sufficient randomization so that the percentage of false positive is very low. On the other hand, the FW1k rule set has many rules with a short prefix, and hence the CRC does not provide sufficient randomization so that the percentage of false positives is high. It would be better to have a hash generator showing consistent false positive rate for all types of rule sets, but finding out better hash functions for all types and any-length prefixes is beyond the scope of this paper. The proposed ALBF provides very good scalability, which can be inferred from the fact that the larger the number of rules relatively the better performance in false positive rate is shown in all type of rule sets. Generally, the ALBF works efficiently in the first stage of hierarchical packet classification to eliminate the inputs that have no match with the source prefix in the rule sets, which is revealed by the percentages of the negatives of the ALBF, and among the positives shown by the ALBF, the percentage of false positives is relatively small.

A C++ model of our proposed packet classification algorithm was developed to analyze the average number of memory accesses and the maximum number of memory accesses required to perform packet classification and to calculate the memory requirement. For all the analysis, the number of hash functions to probe the Bloom filter is two and we set the size of the Bloom filter to be equal to the size of the rule set, i.e. the multiplication factor $l$ is set as ‘1’. The number of memory accesses is the number of rules accessed and compared with the given input packet. The maximum number of memory accesses and the average number of memory accesses required to find a best matching rule are obtained.

For performance comparison with other packet classification algorithms, extensive simulations for H-trie [4], set-pruning trie [4], HBST [5], AQT [7], PQT [8], BSL [9], bit-vector (BV) [10], HiCuts [12], and HyperCuts [13], are performed using the same database. The results are shown in Fig. 7 through Fig. 12. The performance of cutting algorithms such as HiCuts [12] and HyperCuts [13] depends highly on $binth$, which is the number of rules for linear search in a leaf node of the decision tree, and the number of cuts made in each dimension. The simulation results shown in Figs. 7–12 are the case that both the $binth$ and the number of cuts in each dimension are set to 4.

In the average number of memory accesses shown in Figs. 7 and 8, the proposed algorithm is distinctly better or similar to other algorithms. For all the rule sets, the proposed algorithm has up to 25 times better performance when compared with other algorithms. Since both the HBST [5] and our proposed packet classification algorithm follow the hierarchical approach for packet classification, they show similar performance. The H-trie and the set-pruning trie [4] also follow the hierarchical approach, but the search performance is inferior because of the memory accesses to empty nodes. The BSL [9] shows similar performance in the average search performance, but the worst-case search performance of the BSL for some rule sets is poor because of a long linked list mapped to a single node. The PQT [8] shows slightly better performance than the AQT [7] since each empty node in the AQT is replaced by the highest-priority rule among the rules in the subtree rooted by the empty node. In the worst case performance shown in Figs. 9 and 10, the proposed algorithm and the HBST [5] are also far better than other algorithms, where it shows up-to 11 times a better performance. Note that the search performance of the proposed algorithm can be controlled by controlling the false positive rate of the ALBF while it is not possible for other packet classification approaches.

### Table 3
Performance of all-length Bloom filter – 1k rule sets.

<table>
<thead>
<tr>
<th>Rule type (no. of rules)</th>
<th>Size</th>
<th>Memory (Kbyte)</th>
<th>No. of rules</th>
<th>% of negatives</th>
<th>% of false positives</th>
<th>% of true results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL1k (958)</td>
<td>n</td>
<td>0.125</td>
<td>958</td>
<td>70.1</td>
<td>0.1</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td>2n</td>
<td>0.25</td>
<td>958</td>
<td>70.1</td>
<td>0.1</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td>4n</td>
<td>0.5</td>
<td>958</td>
<td>70.1</td>
<td>0.1</td>
<td>99.9</td>
</tr>
<tr>
<td>FW1k (870)</td>
<td>n</td>
<td>0.125</td>
<td>871</td>
<td>78.4</td>
<td>15.4</td>
<td>84.6</td>
</tr>
<tr>
<td></td>
<td>2n</td>
<td>0.25</td>
<td>871</td>
<td>81.2</td>
<td>12.6</td>
<td>87.4</td>
</tr>
<tr>
<td></td>
<td>4n</td>
<td>0.5</td>
<td>871</td>
<td>81.3</td>
<td>12.5</td>
<td>87.5</td>
</tr>
<tr>
<td>IPC1k (988)</td>
<td>n</td>
<td>0.125</td>
<td>988</td>
<td>80.2</td>
<td>12.7</td>
<td>87.3</td>
</tr>
<tr>
<td></td>
<td>2n</td>
<td>0.25</td>
<td>988</td>
<td>84.1</td>
<td>8.7</td>
<td>91.3</td>
</tr>
<tr>
<td></td>
<td>4n</td>
<td>0.5</td>
<td>988</td>
<td>84.2</td>
<td>8.7</td>
<td>91.3</td>
</tr>
</tbody>
</table>

### Table 4
Performance of all-length Bloom filter – 5k rule sets.

<table>
<thead>
<tr>
<th>Rule type (no. of rules)</th>
<th>Size</th>
<th>Memory (Kbyte)</th>
<th>No. of rules</th>
<th>% of negatives</th>
<th>% of false positives</th>
<th>% of true results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL5k (4659)</td>
<td>n</td>
<td>0.625</td>
<td>4660</td>
<td>75.8</td>
<td>1.2</td>
<td>98.8</td>
</tr>
<tr>
<td></td>
<td>2n</td>
<td>1.25</td>
<td>4660</td>
<td>76.9</td>
<td>0.2</td>
<td>99.8</td>
</tr>
<tr>
<td></td>
<td>4n</td>
<td>2.5</td>
<td>4660</td>
<td>76.9</td>
<td>0.1</td>
<td>99.9</td>
</tr>
<tr>
<td>FW5k (4343)</td>
<td>n</td>
<td>0.625</td>
<td>4351</td>
<td>88.0</td>
<td>5.8</td>
<td>94.2</td>
</tr>
<tr>
<td></td>
<td>2n</td>
<td>1.25</td>
<td>4351</td>
<td>88.0</td>
<td>5.8</td>
<td>94.2</td>
</tr>
<tr>
<td></td>
<td>4n</td>
<td>2.5</td>
<td>4351</td>
<td>87.9</td>
<td>6.3</td>
<td>93.7</td>
</tr>
<tr>
<td>IPC5k (4467)</td>
<td>n</td>
<td>0.625</td>
<td>4468</td>
<td>86.4</td>
<td>5.9</td>
<td>94.1</td>
</tr>
<tr>
<td></td>
<td>2n</td>
<td>1.25</td>
<td>4468</td>
<td>86.5</td>
<td>5.8</td>
<td>94.2</td>
</tr>
<tr>
<td></td>
<td>4n</td>
<td>2.5</td>
<td>4468</td>
<td>86.5</td>
<td>5.8</td>
<td>94.2</td>
</tr>
</tbody>
</table>
The comparison on memory requirement for the proposed algorithm with other algorithms is shown in Figs. 11 and 12. In the proposed packet classification algorithm, rules are stored once without any duplication and the RPT does not require empty nodes and the memory overhead is only for the single all-length Bloom filter which requires the negligible amount of memory, and hence the
The memory requirement for the proposed algorithm is close to the linear search structure which has the optimum memory requirement. Trie-based algorithms show relatively better performance than bit-vector algorithm [10] or cutting-based algorithms [12,13] in memory requirement even though the trie-based algorithms have empty nodes. However, the memory requirement for the set-pruning [4] is
very high, because of the rule duplication. The bit-vector algorithm requires a huge amount of memory, especially for large rules sets, because of the bit-vectors stored in every node of each 1-dimensional trie. The cutting-based algorithms also require a large amount of memory because of the rule duplication caused when a rule belongs to multiple leaf nodes of a decision tree.

7. Conclusion

A Bloom filter is a simple, space-efficient, randomized data structure for concisely representing a data set in support of membership queries. For an efficient hierarchical packet classification, our algorithm proposes to use a Bloom filter at the first stage and the rule-priority trie at the second stage. The 1-dimensional match for all the different lengths of prefixes can be performed using a single all-length Bloom filter. At the first stage of pre-filtering, the all-length Bloom filter efficiently narrows the scope of the search. Usually, a Bloom filter will be used to answer for membership queries, but in our algorithm, the Bloom filter not only performs membership queries but also acts as a pointer to the second stage of the search for all the positive results of Bloom filter as seen in the hierarchical packet classification algorithms.

At the second stage, many small rule-priority tries are constructed in order to narrow down the search. In the rule-priority trie, the empty nodes are replaced by the highest-priority rule belonging to the sub-trie rooted by the empty node. Search in the rule-priority trie can be finished either at a leaf or at a match to a priority rule. Hence, the performance of the rule-priority trie in terms of memory access is very high with a less memory requirement. The incremental update is also possible in the proposed algorithm. The simulation results show that the proposed algorithm shows better performance in terms of the average memory accesses, the maximum number of memory accesses, and the memory requirement when compared with other packet classification algorithms.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (2010-0000483). This research was also supported by the MKE (The Ministry of Knowledge Economy), Korea, under the HRNC (Home Network Research Center) – ITRC (Information Technology Research Center) support program supervised by the National IT Industry Promotion Agency (NIPA-2010-C1690-1011-0010).

References