Packet Classification Using Dynamically Generated Decision Trees

Yu-Chieh Cheng, Pi-Chung Wang

Abstract—Binary Search on Levels (BSOL) is a decision-tree algorithm for packet classification with superior speed performance. However, the most decision-tree-based algorithms, like BSOL, may suffer from a memory explosion problem caused by filter replications. In this work, we improve the storage performance of BSOL by employing a scheme, replication control. Our scheme dynamically generates multiple decision trees to eliminate filter replications in BSOL. The experimental results show that the new scheme achieves better performance than the existing decision-tree-based algorithms.

Index Terms—Packet Classification, Firewalls, QoS, Packet Forwarding

1 INTRODUCTION

Packet classification is a process employed by Internet routers to classify packets into network flows based on multiple fields of packet headers specified in predefined filters. The filters for packet classification consist of a set of fields and an action. Each field, in turn, corresponds to one field of packet headers. The value in each field could be a variable-length prefix, range, explicit value or wildcard. The most common fields include a source/destination IP address prefix, a source/destination port range of the transport protocol and a protocol type in a packet header. Formally, we define a filter $F$ with $d$ fields as $F = (f_1, f_2, \ldots, f_d)$. While performing packet classification, a packet header $P$ is said to match a particular filter $F$ if for all $i$, the $i_{th}$ field of the header satisfies $f_i$. Each action has a cost that defines its priority among the actions of the matching filters, and the classifier only applies to the least-cost action from the matching filters. The actions of packet classification are used to fulfill many services such as firewall packet filtering and quality of services [1].

Software algorithms have better scalability; however, their data structures for storing filters significantly affect the performance. Decision tree has been regarded as an efficient data structure for packet classification.

The existing decision-tree algorithms, including Hi-Cuts [6], Modular Packet Classification [7], HyperCuts [8] and CubeCuts [9], proposed different approaches to partition a set of filters. For example, HiCuts selects one field to divide address space into equal partitions and HyperCuts may use more than one field for space partitioning. CubeCuts separates one hyperrectangle space from the rest. The filters occupying more than one space partition are replicated. As a result, the cut rules for space partitioning affect the degree of filter replications.

To alleviate filter replication, both BSOL [10] and EffiCuts [11] employ multiple decision trees to store filters. BSOL constructs a specialized decision tree which can apply an algorithm, binary search on prefix lengths [12], to improve the search performance. The time complexity of BSOL is $O(W \log W)$, where $W$ is the total number of bits of all inspected fields in a packet header. Because BSOL may suffer from the filter replication problem, a refinement, namely BSOL2, is presented to store filters in two decision trees in a static manner. However, the static filter arrangement may not perform well for some filter sets. EffiCuts [11] generates multiple decision trees according to the wildcard distribution of filters. In the worst case, 15 decision trees are generated to result in poor search performance. Selective tree merging is then proposed to dynamically merge two trees which meet some constraints. Since at most two trees can be merged, the search performance of EffiCuts may suffer from accessing more than five decision trees.

In this work, we propose an enhanced scheme based on BSOL by employing a replication control scheme to alleviate memory overhead of BSOL without performance penalty. Unlike previous schemes which generate multiple decision trees based on wildcards or short prefixes, our scheme dynamically generates new decision trees according to the degree of filter replication. The experimental results show that the memory requirement is reduced with the same speed performance as compared to BSOL. Our scheme also achieves better speed performance and memory requirement than EffiCuts.

The rest of the paper is organized as follows. Section
2 describes the scheme of BSOL. Then, we point out the problem of BSOL in storage performance and present our scheme to alleviate the problem in Section 3. We also present a refinement to further improve the performance. In Section 4, we present the experimental results by comparing the memory requirements and accesses with those of BSOL2 and other existing decision-tree algorithms. Finally, Section 5 provides a summary of the current work.

2 Binary Search on Levels

We generate multiple decision trees based on BSOL due to its superior speed performance. In this section, we briefly describe the algorithm of BSOL. We divide the construction of BSOL into two parts: decision tree construction and hash building.

Assume that each filter has \( d \) fields, and each field can be an arbitrary range. Each filter can be treated as a hyper-rectangle in a \( d \)-dimensional address space, where each side is equal to the corresponding range. BSOL generates a binary decision tree. In the decision tree, each node is associated with a space, where the root corresponds to the whole \( d \)-dimensional space. The space of a node is equally shared by its two child nodes. Both child nodes have one side which corresponds to upper or lower half of a selected dimension of their parent node. In a decision tree, each node has a filter list where each filter in the list overlaps the space of the node. Initially, all filters are associated with the root node of the decision tree. While the number of filters in a node is greater than a predefined bucket size, two child nodes are generated. The filters in the parent node are inserted into a child node according to whether their space overlaps the space of the child node. If a filter overlaps both child nodes, it will be inserted into both nodes to activate filter replication, which is the main reason of memory expansion for a decision-tree algorithm. To minimize the number of replicated filters, it is necessary to properly select the dimension for dividing the space of a node. Each cut rule of BSOL minimizes the number of replicated filters. BSOL also constraints the same cut rule when applying on nodes with the same tree height. This restriction is designed for storing the same-height nodes in a hash table.

We use an example with twelve five-field filters (\( F_0 \) to \( F_{11} \) in Table 1) to explain the procedure of generating a BSOL decision tree, where each leaf node can store up to four filters. As shown in Fig. 1, initially, all filters are listed in a root node (dark-gray node). Next, a cut rule is generated to partition the rules in the root node into two groups. A cut rule using field \( f_5 \) is produced, because it does not incur any filter replication. Next, field \( f_2 \) is used as the cut rule for the second-level nodes, because only two rules, \( F_{10} \) and \( F_{11} \), are replicated. Node space partitioning stops when all leaf nodes (as designated as bolded nodes) have less than or equal to four filters. After completing the BSOL decision tree, there are 23 rules stored in leaf nodes (i.e., a replication ratio of 1.9).

\[
\begin{array}{cccccc}
\text{Filter} & f_1 & f_2 & f_3 & f_4 & f_5 & \text{Action} \\
F_0 & 000* & 111* & [10:10] & * & UDP & act_0 \\
F_1 & 000* & 111* & [01:01] & [10:10] & UDP & act_0 \\
F_2 & 000* & 111* & [10:10] & [01:01] & TCP & act_1 \\
F_3 & 000* & 111* & [01:01] & * & TCP & act_2 \\
F_5 & 0* & 111* & [10:10] & UDP & act_0 \\
F_6 & 0* & 111* & [01:01] & UDP & act_0 \\
F_7 & * & 01* & * & TCP & act_2 \\
F_8 & * & 0* & * & [01:01] & UDP & act_0 \\
F_9 & * & 0* & [01:01] & UDP & act_0 \\
F_{10} & * & * & * & UDP & act_3 \\
F_{11} & * & * & * & TCP & act_3 \\
\end{array}
\]

Fig. 1. Decision Tree of BSOL. (Bucket Size: 4)

After constructing a decision tree, the nodes with the same tree height are specified by the same-length prefixes due to the restriction of BSOL cut rules. Accordingly, a set of hash tables is generated to store the leaf nodes, where each hash table corresponds to a height of the decision tree with leaf nodes. For the example in Fig. 1, four hash tables are generated. The markers are then inserted into the hash tables for enabling a binary search procedure [12]. To perform packet classification, the binary search procedure upon the hash tables is performed to yield the matching leaf node, without traversing a decision tree from root to leaf nodes. All filters in the leaf node are compared to determine the matching filter. By making use of hash tables, the search performance of BSOL mainly ties to the bucket size.

3 Replication Control

A proper field selection for space partitioning can alleviate the phenomenon of filter replication for most filters. However, there are still a few filters with more than one wildcard that cannot avoid replication. For small filter sets, filter replication is acceptable since their decision trees are usually short. In a high decision tree, the number of replicated filters is usually unacceptable. As a result, a single decision tree may not be scalable for large filter sets. In [10], the authors present BSOL2 by dividing the filter set into two subsets. A subset stores the filters whose length of source IP address prefix is...
less than five. The remaining filters are stored in another subset. Each subset has a decision tree for storing its filters. The source IP address is selected since there are usually the most wildcards or short prefixes.

In some cases, the approach of BSOL2 can significantly reduce memory requirement. However, it lacks of flexibility and may lead to worse results for some cases. Hence, we propose a new scheme, replication control, to solve the filter replication problem and a refinement to further improve the search performance.

Replication control adjusts the filters stored in a decision tree by utilizing the information of filter replication during tree construction. Let \( n \) be the number of original filters to be stored in a decision tree. Filter replication ratio threshold, \( r _ { r t h r e s h } \), is defined to limit the number of replicated filters, where \( r _ { r t h r e s h } \times n \) is the maximum allowable number of filters in a decision tree (i.e., the total number of filters including replicated filters after completing tree construction must be less than or equal to \( r _ { r t h r e s h } \times n \)). Initially, we employ the BSOL procedure to construct a decision tree. For each filter, we track the tree level where its first filter replication takes place, referred to as replication level. When a set of nodes is partitioned in the same tree level, the filter replication number is also counted. If the total number of the stored filters in a decision tree is greater than the upper bound, the filters with the smallest replication level are removed from the decision tree. The procedure of tree construction proceeds for other filters. If the number of stored decision tree is once again higher than the filter threshold, then the above procedure of removing filters repeats until the decision tree is completed. We choose filters with the smallest replication level since these filters usually occupy the largest space. Filters removed from the decision tree are inserted into a new decision tree to repeat the above steps. If another set of filters is removed in the construction procedure, the third decision tree is constructed and so on. Finally, we may have multiple decision trees according to the characteristics of the original filters. Because our scheme may lead to more decision trees to degrade the search performance, there is a tradeoff between storage efficiency and speed performance. We also note that filter removal may reduce the number of filters in the buckets and degrade the storage efficiency. If a constructed tree is too high to search, we need to reconstruct the decision tree to improve the storage efficiency. Since we have removed the filters which may cause severe replication, the tree reconstruction can achieve a better storage efficiency.

We use previous example to illustrate the differences between BSOL2 and BSOL-RC. For both schemes, the bucket size is reduced to two. Let us consider the decision trees of BSOL2 first. Because \( f _ 1 \) cannot be used to distinguish any filters, filters are categorized into two subsets based on the criteria whether their \( f _ 1 \) values are wildcards. The decision trees corresponding to both filter subsets are depicted in Fig. 2, where the number of filters stored in both decision trees is reduced to 17.

Next, the decision trees based on the procedure of replication control are shown in Fig. 3. The \( r _ { r t h r e s h } \) value is set to one so that any filters causing replication in a decision tree are removed to the next decision tree. Initially, all filters are stored in the root node of the first decision tree to start constructing a BSOL decision tree. In the second level, both \( F _ { 0 } \) and \( F _ { 1 } \) cause replication so that they are removed from the decision tree. Similarly, the replication level of \( F _ { 2 } \) is the fourth level and that of \( F _ { 3 } \) is the fifth level. The filters removed from the first decision tree are denoted by bold fonts. They are stored in the root of the second decision tree for construction. There is no filter replication in both BSOL decision trees with replication control. The replication control scheme adaptively removes filters to alleviate the storage expansion; thus, it can accommodate different types of filter sets to achieve better storage efficiency.
Using multiple decision trees to store filters would degrade the search performance since one linear search is performed for each decision tree. We make a refinement to improve the search performance by halving the bucket size of the second and later decision trees. Baboeescu et al. have observed that in real-world classifiers, the maximum number of filters whose source and destination IP address fields match a packet is less than 20 [13]. Accordingly, the bucket size of BSOL is set to 20. A larger bucket size may have less filter replication and less tree height, but the cost of linear search in a bucket increases. Since we have divided the original filters into multiple sets, the number of filters in the second or later decision trees that match any packet may decrease. Thus, it is reasonable to decrease bucket size to lower the cost of linear search. Although the tree height would increase with a smaller bucket size, we observed that the number of filters in the second or later decision trees are usually relatively small. Furthermore, we use binary search to access the decision tree in the search procedure, and the increase of tree height is less influential than the increase of bucket size. As a result, the total search cost is still decreased with lower-cost linear searches.

4 Performance Evaluation

We evaluate the performance of our scheme, BSOL-RC, with several filter sets generated by ClassBench [14]. ClassBench provides twelve seeds to generate different types of filter sets. For each seed, 16K filters with five fields, two IP address prefixes, two port ranges and a protocol number, are generated. The performance metrics include the required memory in kilobytes (KB) and the number of memory accesses for a packet classification in the worst case. The value of \( rrthresh \) is fixed to two to maintain the memory consumption.

Fig. 4 shows memory requirements of BSOL2 and our scheme. For most filter sets, our scheme consumes less memory than BSOL2 does. BSOL2 may need significantly more memory than BSOL-RC for some filter sets due to the lack of flexibility for generating more than two decision trees. In these filter sets, wildcard appears in the fields other than source IP address field to result in a significant increment of filter replication. Since BSOL-RC can dynamically store replicated filters in a new decision tree, it features relatively consistent memory requirements.

Fig. 5 shows the number of memory accesses per lookup in the worst case for BSOL2 and BSOL-RC. The results indicate that even with the storage reduction, the speed performance using our scheme is similar to that of BSOL2 for most cases. There are few cases that our scheme is slower than BSOL2. This is due to a tradeoff between storage and speed performance, where our scheme generates more decision trees to result in less memory requirements for these cases (fw1, fw3, fw4, and fw5). Nevertheless, BSOL-RC benefits from the tradeoff to improve feasibility by moderating memory consumption and providing comparable speed performance.

We further use ClassBench to generate 64K-filter sets to test the scalability of our scheme. As shown in Fig. 6, BSOL2 may suffer from divergent storage requirements for large filter sets. BSOL-RC allows memory consumption to be linearly proportional to the number of filters. Moreover, the speed performance can be maintained at the same level as 16K-filter sets, as shown in Fig. 7.

We use three 100K filter sets in [11] to compare BSOL-RC with several decision tree algorithms. Table 2 shows both speed and storage performance of four decision-tree algorithms, including HyperCuts, EffiCuts, BSOL2, and our scheme. The result shows that the algorithms with multiple decision trees have much better storage performance than HyperCuts, which only has one decision tree. While the multiple-decision-tree algorithms improve storage performance, the search performance might degrade as a tradeoff for storage efficiency. For example, EffiCuts significantly reduces memory requirement from HyperCuts for ACL and FW, but also requires extra memory accesses. For IPC, however, both speed and storage performance is improved by EffiCuts simultaneously because HyperCuts generates a high decision tree. BSOL2 avoids the problem of high decision tree by using the technique of binary search on prefix lengths [12]; thus, BSOL2 usually has better speed performance than EffiCuts. The storage efficiency of BSOL2 would
TABLE 2
Performance Comparisons of the Existing Decision-Tree Algorithms with 100K Filter Sets.

<table>
<thead>
<tr>
<th>Filter Set</th>
<th>HyperCuts</th>
<th>EffiCuts</th>
<th>BSOL2</th>
<th>BSOL-RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL 100K</td>
<td>Memory accesses: 37, Memory requirement: 1084 MB</td>
<td>Memory accesses: 83, Memory requirement: 5.3 MB</td>
<td>Memory accesses: 25, Memory requirement: 19.1 MB</td>
<td>Memory accesses: 20, Memory requirement: 3.2 MB</td>
</tr>
<tr>
<td>FW 100K</td>
<td>Memory accesses: 48, Memory requirement: 2433 MB</td>
<td>Memory accesses: 53, Memory requirement: 3.7 MB</td>
<td>Memory accesses: 24, Memory requirement: 9.4 MB</td>
<td>Memory accesses: 25, Memory requirement: 2.1 MB</td>
</tr>
<tr>
<td>IPC 100K</td>
<td>Memory accesses: 24, Memory requirement: 575 MB</td>
<td>Memory accesses: 17, Memory requirement: 5.5 MB</td>
<td>Memory accesses: 17, Memory requirement: 1.9 MB</td>
<td>Memory accesses: 14, Memory requirement: 2.3 MB</td>
</tr>
</tbody>
</table>

Fig. 6. Storage Performance for 64K-filter Sets.

Fig. 7. Speed Performance for 64K-filter Sets.

degrade due to two limitations. First, to construct a decision tree for applying binary search, BSOL2 must limit the nodes at the same level equally partitioned by the same field. Second, BSOL2 always generates two decision trees. As a result, BSOL2 may consume more storage than EffiCuts. BSOL-RC could reduce the memory requirement by employing replication control. By removing the filters which cannot be properly categorized, the height of the resulted decision tree is significantly decreased as well as the number of markers. The speed performance also benefits from a shorter decision tree. BSOL-RC also outperforms EffiCuts in both storage and speed performance. EffiCuts uses decision trees that must be traversed from the root node; thus, the intermediate nodes are stored and accessed in the search procedure. Although BSOL-RC stores extra markers, it reduces the tree height to minimize the number of markers. In sum, BSOL-RC can promote storage efficiency while maintaining comparable or better speed performance than the prominent decision-tree algorithms.

5 CONCLUSIONS

In this work, a new algorithm for packet classification, BSOL-RC, is presented. While the BSOL algorithm generates fixed number of decision trees, the new algorithm generates decision trees according to the number of replicated filters. By controlling the number of replicated filters, both memory consumption and height of a decision tree can be manipulated. As a result, both space and speed performance can be improved. The experimental results show that BSOL-RC can reduce more than 50% memory consumption from BSOL2 for some filter sets. For large filter sets, the storage saving of BSOL-RC is more significant. In brief, BSOL-RC has a performance advantage over the existing decision tree algorithms.

REFERENCES