Fast Packet Classification on OpenFlow Switches Using Multiple R-Tree Based Bitmap Intersection*

Ding-Fong Huang[1], Chien Chen[12] and Mahadevan Thanavel[3]
[1]Department of Computer Science, [2]Information Technology Service Center and
[3]Department of Electrical Engineering and Computer Science
National Chiao Tung University, HsinChu, Taiwan
artines1@gmail.com, chienchen@cs.nctu.edu.tw, tmahadevan1991@gmail.com

***Abstract*—In order to accomplish a stringent speed requirement for processing internet services such as Access Control List (ACL), Quality of Service (QoS), firewalls, etc., software based OpenFlow switches must have a fast packet classification capability. Even for hardware based OpenFlow switches, a limited size of Ternary Content Addressable Memory (TCAM) in the switch could be only enough for a forwarding table. Therefore, ACL, firewall tables, etc. need to be implemented by using the memory of the switch CPU. However, it has become a great challenge to build extremely effectively for next-generation software based packet classification that supports higher throughput and larger flow entries in OpenFlow switch. This paper first exploits a fast packet classification algorithm that forms a R*-Tree based Bitmap Intersection and secondly discusses an enhanced R*-Tree based Bitmap Intersection by using Bloom Filter and Multiple R*-Tree. The evaluation results show that the performance of the algorithm in OpenFlow switches is 4.42 times of Bitmap Intersection and 5.16 times of R*-Tree algorithm and consumes only 300 KB of memory space, which is much less than that of other methods. Finally, the use of multiple R*-Trees has further improved memory usage by about 30%.**

Keywords- openflow; software defined networking (SDN); packet classification; bitmap intersection; R*-Tree; Bloom filter;

I. INTRODUCTION

While Software-Defined Networking (SDN) technology is evolving, more attention has been placed on improving the performance of OpenFlow switches. Due to the enormous growth of Internet traffic and rule set size, multi-field packet classification has become a majorly challenging part of Software-based OpenFlow switches such as Open vSwitches. It poses a similar challenge as the hardware based OpenFlow switches with limited Ternary Content Addressable Memory (TCAM) to handle switch's line rate moved beyond 40 Gbps. Since the cost and power consumption limit the size of TCAM, the switch can use memory from the switch CPU to store some flow tables to classify packets for Internet services such as Access Control List (ACL), Quality of Service (QoS), firewalls, etc. Conventional packet classification problems can be represented as geometric problems. In multiple-dimensional geometric space, we can represent a rule with a rectangle. It means each field of a rule represents a covered area in a different dimension. Furthermore, packet header information could be represented as a point in the geometric space.

Geometric packet classification is a difficult problem. The traditional solution requires either a large storage or a classification procedure that takes too much time. A great packet classification algorithm must have faster speed and less storage requirement at the same time. We observed that the bitmap intersection [4] packet classification algorithm has great performance for classification speed. Bitmap Intersection can handle different dimensions separately, thus it is possible to implement Bitmap Intersection in hardware to achieve high speed packet classification in parallel. However, it requires a very huge memory space when handling a large rules table. If Bitmap Intersection is implemented in software, the amount of memory access during classification procedure also increases. When memory access raises, it means classification will spend more time on memory access, which could decrease classification speed. In addition, we also observed that the R*Tree algorithm [8] utilizes a tree structure to index objects in geometric space, and has great performance on memory usage. However, the R*-Tree algorithm has slow speed when searching for objects, because it uses linear search in each node. By synthesizing these two different algorithms, we think it is possible to utilize their advantages to conquer each other's drawbacks. To solve such a software based packet classification problem in OpenFlow switch, we propose the R*Tree based Bitmap Intersection algorithm that tries to use the efficient spatial utilization of R*-Tree to conquer the large memory usage issue of Bitmap Intersection, and use the fast matching speed of Bitmap Intersection to improve linear search speed in each node of R*-Tree.

Moreover, we further analyze the performance of our proposed solution, which shows that one of the factors influencing search performance is the number of traversed paths in the R*-tree. It is obvious that when the number of traversed paths increases, the number of R*-tree nodes we need to visit also increases. As the number of traversed nodes increases, it needs to perform more instances of Bitmap Intersection algorithm as the number of visited nodes grows. The cumulative time spent on searching also increases. Therefore, we further propose an Enhanced R*-Tree based Bitmap Intersection to enhance the search performance of the tree so that the traversed paths are categorized into unnecessary and necessary paths. An unnecessary path is minimized by using Bloom Filter (BF). The number of necessary paths is also minimized by using Multiple R*-Tree that categorizes the path with high-priority rules that cover a small area in geometric space and the path with wild card rules that cover a large area in geometric space. Finally, the simulation result shows that the performance of our algorithm in

978-1-5386-3416-5/18/$31.00 ©2018 IEEE

the Software-based OpenFlow switches shows a significant improvement of 4.42 times that of Bitmap Intersection, and 5.16 times that of R*-Tree algorithm. Also in terms of memory, it only requires 300 KB compared to the 420 KB required for R*-Tree and 10 MB required for Bitmap Intersection with 10000 rules. The overall performance of Software-based OpenFlow switch using Multiple R*-Tree is further improved by about 30%.

The rest of paper is organized as follows: section II describes the related works about packet classification based on geometric space and OpenFlow related works. Section III describes one of the first proposed works, which is R*-Tree based Bitmap Intersection. Section IV describes the second proposed work, which is the continuation of the previous section as Enhanced R*-Tree based Bitmap Intersection using Bloom Filter and Multiple R*-Tree. Then we evaluate our algorithm and compare with other algorithms in Section V and conclude this work in Section VI.

II. Related Works

Recently the OpenFlow-based SDN is drawing attention because it can offer an agile and flexible network. When scaling the SDN to a larger size, both control plan and data plan performance of OpenFlow switches has become a concern. Kun et al. [1] propose GFlow, which describes the benefit of the parallelism of GPU resources with CPU for table matching in Software-based OpenFlow Switches. It uses an ItemGraph data structure in order to organize flow entries into a directed acyclic graph (DAG). ItemGraph is able to solve flow table matching issues by working in a parallel manner as well to save table matching overhead. Moreover, increasing the number of match fields has led to a performance bottleneck as GPU has limited memory capacity.

Maciej. K et al. [13] describe the performance characteristics of flow tables updates with three hardware OpenFlow switches. They present experiments where they monitor rule updates in the control plane and send traffic to the updated rules. To measure the affected rule update speed, they quantify the tradeoff between the rule update rate and servicing delay. The above-discussed state of art induced us to propose a concise scheme to solve a packet classification problem on Software-based OpenFlow switches.

The packet classification problem is a long existing problem, and there are plenty of solutions that have been proposed. Our R*-Tree based Bitmap Intersection belongs to the category of the geometric algorithm, so we concisely introduce previous works in this field. Other details about packet classification are given in these papers [6] [7] [12].

HyperCuts [11] is very similar to HiCuts, but its partition procedure is different from HiCuts. For one partition, HiCuts cuts one dimension at a time, whereas HyperCuts cuts multiple dimensions and also uses a decision tree to index blocks, but uses a multi-dimension array to store multi-dimensional space information in a node of the tree. It has a great performance for classification and doesn't require large memory space, but it cannot handle wild card rules very well.

Figure 1 shows the Bitmap Intersection scheme [4], which uses the geometrical space decomposition approach to project every rule on each dimension. For N rules, a maximum of 2N+1 non-overlapping intervals are created on each dimension. Each interval is associated with an N-bits bit vector. Bit j in the bit vector is set if the projection of the rule range corresponding to rule j overlaps with the interval. On packet arrival, for each dimension, the interval to which the packet belongs is found. Taking the conjunction of the corresponding bit vectors in each dimension as a resultant bit vector, the highest priority entry in the resultant bit vector can be determined. The Bitmap Intersection scheme has a fast matching speed, and it can be speed up by parallel hardware computing, but it requires a huge memory space when the size of the rule table grows large and the space complexity is $O(dN^2)$, where d represents a number of dimensions and N represents a number of rules. It states that as the rule number grows linearly, the requirement of memory space grows quadratically. Besides, when the rule number becomes large, the length of the bit vector also grows, requiring more memory access to load the bit vector, so the matching speed also becomes slower. On the other hand, Figure 2 shows that the R*-Tree packet classification algorithm [8] utilizes a tree structure to index objects in the geometric space. In R*Tree, spatial objects are considered a Minimal Bounding Region (MBR), such as the four MBRs a, b, c, and d in figure 2. Several MBRs (e.g. a, b, c, and d) are assembled into a bigger MBR (e.g. B). R*-Tree uses a tree structure to retain geometric information of MBRs. Every node in R*-Tree indicates one MBR, and children nodes represent lower layers of MBRs. It has great performance on spatial utilization, since as the number of rules grows linearly, the memory requirement also grows linearly. But it has slow matching speed since it uses a linear search to find overlapping Minimal MBRs when searching a R*-tree node.

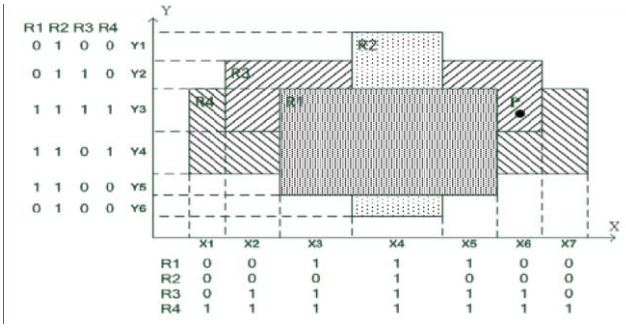


Fig. 1. Bitmap Intersection

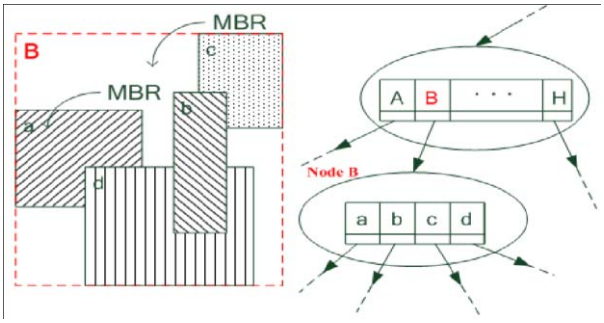


Fig. 2. R*-Tree node structure

III. R* -Tree Based Bitmap Intersection

Currently switches are enabled with OpenFlow protocols [2], [3] that provide network virtualization and bring programmability to the network infrastructure. The part of the OpenFlow switch involving the most processing is packet classification, where up to 12-tuple header fields of each packet are matched against all the rules [3]. Next-generation packet classification could be viewed as an extension from traditional

packet classification, whose solutions have been extensively studied in the past decade. The following work looks into next generation packet classification problems using geometric packet classification in terms of memory access and search performance for Software-based OpenFlow switches.

Both R*-Tree algorithm and Bitmap Intersection could solve the geometric packet classification problem exclusively. However, they have different drawbacks that may hamper their performance. In order to solve these problems, we implement a R*-Tree based Bitmap Intersection algorithm that combined two different algorithms together. We try to use the benefit of spatial utilization of R*-Tree to conquer the memory issue of Bitmap Intersection, and use the fast matching speed of Bitmap Intersection to improve the search speed of R*-Tree. This approach is based on the R*-Tree packet classification algorithm, and utilizes Bitmap Intersection on every node of R*-Tree. We observe that it searches one point for every search in geometric packet classification problem. We consider every node an independent geometric space. This space contains sub MBRs that belong to this node. If we regard these MBRs as rules, then we can utilize Bitmap Intersection on the node, instead of the linear search procedure of the R*-Tree algorithm. Besides, because the number of rules inside a node will be bound, it does not require much memory space for each node. Even though the number of MBRs will be greater than the number of rules, they are going to be divided into multiple small subsets, where each subset is represented by very short bit vectors. Hence the total memory requirement is much less than that of Bitmap Intersection.

To construct R*-Tree based Bitmap Intersection, we translate rules into MBRs first, which means we map rules into geometric space. MBR contains the start and end positions for every dimension, and it also retains information corresponding to the rule (i.e. priority, action). Next, we use an original R*-Tree construction algorithm to insert these MBRs (i.e. rules) into R*-Tree until they have been all inserted into R*-Tree. And then we use Depth-First-Search or Breadth-First Search to visit the nodes sequentially. On each node, we use Bitmap Intersection to translate MBRs into bit vectors and record these bit vectors back into a node until we reach a leaf node. Figure 3 shows the node structure which is constructed, where we use Bitmap Intersection to translate four MBRs into bit vectors and store these bit vectors into node B.

When a packet comes in, Software-based OpenFlow switches first extract header information from the packet and map it into a point, then query the R*-Tree that we built before with this point. It starts from a root node and uses Bitmap Intersection to find MBRs that overlap with the query point, meaning that it is different from the original linear search algorithm. We use Bitmap Intersection to find MBRs inside node B quickly. Its search speed is much faster than that of linear search. The MBRs that correlate with the bits that have been set to 1 in the resultant bit vector are overlapped with the query point. After finding overlapping MBRs, we then keep visiting nodes corresponding to these MBRs until reach a leaf node. We use the same approach on a leaf node to find overlapping MBRs. These MBRs represent matching rules, and the one with the highest priority is going to be the final matching result.

We further analyze this problem and observe that a major factor influencing searching performance is the number of traversed paths. When the number of traversed paths grows, the time spent on search also increases. It is because as the number of visited nodes increases, it needs to perform more Bitmap Intersection algorithms as the number of visited nodes grows, so it takes more time to finish. Thus we need to reduce the number of traversed paths in terms of improving search performance on Software-based OpenFlow switches to support a large number of header fields that would be depicted in the following sections.

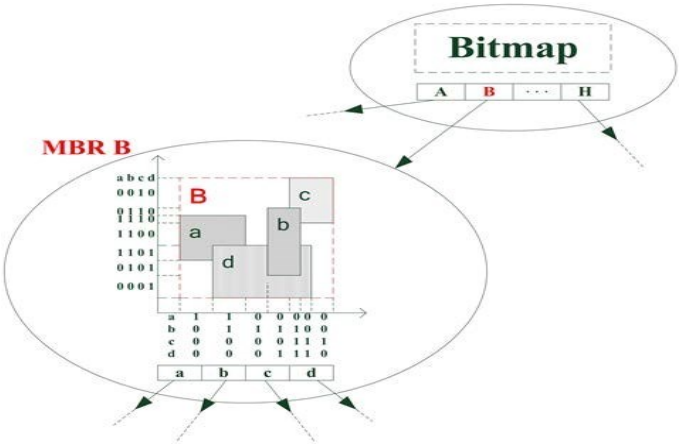


Fig. 3. The structure of R* -Tree based Bitmap Intersection

IV. ENHANCED R*-TREE BASED BITMAP INTERSECTION

In order to enhance the above searching speed of the R*-tree based Bitmap intersection for packet classification while applying to Software-based OpenFlow switches that necessitate to further reduce the number of traversed paths. From the above approach, traversed paths can be categorized into two different types: necessary and unnecessary paths. Necessary paths indicate the paths that will lead to a result, these paths are necessary to be traced for results during the search procedure, so the costs of necessary paths can't be avoided. Unnecessary paths indicate the paths that will not lead to a result. Even if it will not get a result, we can only know this after we finish tracing this path. Thus the cost spent on the unnecessary path is a waste, and will decrease search performance. As an example, see the necessary path and unnecessary path in Figure 4, which contains one R*-Tree with tree height 2, and shows the traversed paths for searching packet P. We can see that it finds the results from nodes D and G individually, so the paths that traversed to these two nodes are necessary paths. And we could also find that a path that traversed to node C is an unnecessary path, because we do not acquire matching rules from this path, so actually this path is not required to be traced.

We can improve the searching performance very efficiently by reducing the number of paths during the search procedure. Here we propose two different enhanced approaches that try to improve searching performance through reducing necessary paths and unnecessary paths respectively. We try to reduce unnecessary paths through Bloom Filter, which can filter unnecessary paths out. And we use multiple R*-Trees to separate necessary paths into multiple R*-Trees, which will decrease the number of necessary paths as we search one R*Tree.

A. Bloom Filter

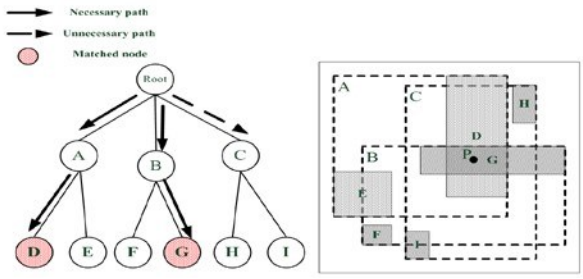


Fig. 4. R*-tree contains Necessary path and unnecessary path

The Bloom Filter is a special data structure that can check whether an element belongs to a set. It has very good space utilization, and it takes a small time to check one element. One Bloom Filter consists of a one-bit vector with length N and M hash functions as shown in Figure 5. In Bloom Filter, false positive matches are possible, but false negatives are not. In our case, a query returns either "possibly in the path" or "definitely not in path".

We can reduce the number of unnecessary paths through Bloom Filter. The basic idea is that we utilize Bloom Filter to check whether the path it traces right now will get a result or not. We utilize Bloom Filter to filter unnecessary paths out during searching in a higher level. Then we can discard those paths, which will save the time spent on those unnecessary paths. We utilize Bloom Filter at higher level nodes, such as level one or level two nodes. After R*-Tree construction is finished, for each higher level node, we employ a Bloom Filter at this node, and program it with all the rules which are below this node, and utilize the programmed Bloom Filter in this node. When traversing a higher level node, it first utilizes the Bloom Filter which is stored in this node to check whether the searching packet belongs to the rules that are below this node. If it is positive, then we keep trace this path. If it is not, then we can discard this path to save the additional cost of this unnecessary path.

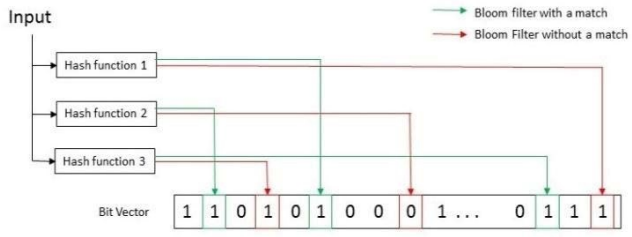


Fig. 5. Bloom Filter with a match and non-match element

For practical implement, if we want to add a Bloom Filter into one node, we first gather all the rules below this node, these rules located at leaf nodes below this node, then program a Bloom Filter with these rules. One node will contain several Bloom Filters, each Bloom Filter corresponding to one field of the rule. When we program a Bloom Filter, we use data inside these fields to program their corresponding Bloom Filters individually. When it traces to the node with Bloom Filters, we first get fields from the header information, and then we test these fields with their corresponding Bloom Filters separately. If all Bloom Filters have a positive result, it represents this packet belonging to the rules. If one or more Bloom Filters show a negative result, then it indicates that this packet does not belong to the rules.

But there is a problem that needs to be solved here. Since the fields of rules are stored in prefix form, and the fields inside packet are represented by a value, it is not possible to use the Bloom Filters which are programmed by prefixes to handle the packet information in value form directly. To solve this issue, we can employ prefix expansion to expand the prefix into values, then program the Bloom Filter with these expanded values. We can use this kind of Bloom Filter to test whether a packet belongs to the set of rules. But it causes another problem, where the prefix expanding could generate huge values. Assume there is a field ready to be expanded, and it is an IP prefix with prefix length 2. Fully expanding this prefix is going to generate 230 values, about one billion values. It is too huge to handle with a Bloom Filter.

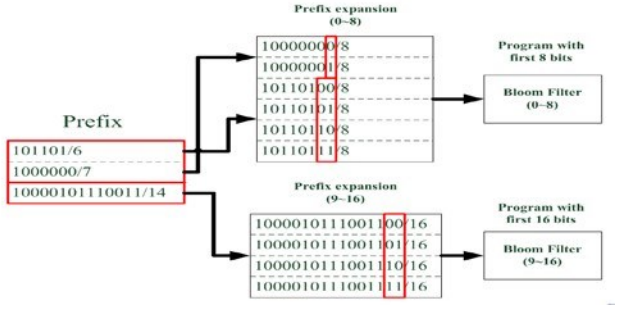


Fig. 6. Prefix expansion with 8-bits and 16-bits

For solving this issue, we divide the prefix into a few subsets according to prefix length, each subset aggregates prefixes which its prefix length within a range. For an instance, we divide the prefix into four subsets, the first subset encloses prefixes with prefix length between 0 and 8, and the second subset with prefix length between 9 and 16, the third subset with prefix length between 17 and 24, and the fourth subset with prefix length between 25 and 32. In this way, we don't have to fully expand prefixes for each subset, we only need to expand to the highest length in the subset. It can reduce the impact of prefix expanding efficiently. We employ four Bloom Filters to handle these four prefix subsets individually. When handling a packet, these four Bloom Filters should be used to test the same field in this packet. We use the first 8 bits in the field to test the first Bloom Filter, use the first 16 bits for the second Bloom Filter, and so on. If one of these four Bloom Filters shows a match, it means this field belongs to the corresponding field in the rule. If there is no Bloom Filter showing match, it means this field does not belongs to the rules. In Figure 6, there is an example of using a prefix for another expansion, considered three prefixes, where their prefix lengths are 6, 7, and 14 respectively. Here, we divide these three prefixes into two subsets, one for prefix length between 0 and 8 and another for prefix length between 9 and 16. Then we expand the prefixes in the first subset into prefix length 8 and expand prefix in second subset into prefix length 16. After prefix expansion, we use two Bloom Filters to handle these two subsets respectively. We program these Bloom Filters with their corresponding prefixes in the subsets. For the subset with prefix length 8, we use the first 8 bits of prefix to program its Bloom Filter and use the first 16 bits. But there are some drawbacks when we use Bloom Filter. First, Bloom Filter requires executing the hash function many times while testing whether one element belongs to a set. These executions also spend time, so there will be an additional cost. If the cost of these executions is greater

than the cost we saved, it will make searching performance decrease. Second, it also requires more memory space for Bloom Filters, so the memory requirement will be enlarged after we utilize Bloom Filter.

B. Multiple R-Tree*

Next, we try to reduce the number of necessary paths. We first observe the characteristics that necessary paths have during a searching packet. We find out that necessary paths can be generally categorized into two types. One is a necessary path that will lead to a high priority rule that covers a small area in geometric space normally. Another is a necessary path that will lead to a wildcard rule which has low priority and covers a large area in geometric space. In general, the packet matches with a number of high priority rules are going to less than those with wildcard rules. In other words, most of the necessary paths traced are for low priority wildcard rules. It is because wildcard rules cover a wide area in geometric space, so it is easier to get wildcard rules during searching. Thus, it is possible that utilizing these characteristics to reduce the necessary paths which will lead to wildcard rules will further improve the searching performance.

Thus we separate these wildcard rules from the rules table, and use an extra R*-Tree to handle these wildcard rules. So there are two R*-Trees, one for normal rules, another one for wildcard rules. Figure 7 shows an example of multiple R*-Trees, where we split the rule table into two tables depending on priority before building multiple R*-Trees. For instance, we separate the rules which are located at the first 80% of priority into first table and remaining 20% of rules into the second table. It can efficiently separate wildcard rules, because wildcard rules usually have low priority.

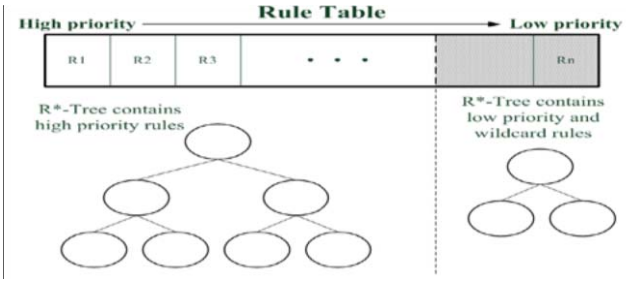


Fig. 7. Build multiple R*-Tree

After splitting the rule table with this approach, the necessary paths are going to be separated into two R*-Trees. This will decrease the necessary paths we take while searching a single R*-Tree. And since the priority in the first R*-Tree is always greater than that of the second R-Tree, we don't have to search the second R*-Tree if we get a result from the first R*Tree. Most of the packets get a result from the first R*-Tree, which means we can save the cost spent on necessary paths in the second R*-Tree. This is going to improve the searching performance of our approach. Obviously, this way would lead to improving the performance of Software-based OpenFlow switches in next-generation packet classification networks.

V. PERFORMANCE EVALUATION

The performance of Software-based OpenFlow switches are evaluated by our approaches in the following in terms of the different rules tables, different node size, the node size indicating the number of MBRs in one R*-Tree node, as well as using Bloom Filter and Multiple R*-Tree respectively. In this paper, we employ ClassBench [5] to evaluate the performance of our approach. ClassBench is a suite tools for benchmarking packet classification algorithms. ClassBench consists of a rules table generator and a flow generator. ClassBench also provides three different types of configuration files, which reflects three different type rules tables. These three types of rules table are listed below: Access Control List (acl), Firewall (fw), and IP Chain (ipc). The rules, which we use to evaluate our approach have five dimensions, each dimension corresponds to one of the five fields in header information (Source/Destination address, Source/Destination Port, Protocol). The following performance results are averaged through 100000 simulations runs.

A. Memory Access and Memory Usage

We first compare the performance in the following order such as Bitmap Intersection, R*-Tree packet classification, and R*-Tree based Bitmap Intersection respectively. For each algorithm, we show memory access and memory usage with different types of rules table such as Access Control List (ACL), Firewall, and IP Chain (acl, fw, and acl) and different sizes of rules tables.

Figures 8, 10, and 12 show the number of memory access for acl, fw, and ipc rules table respectively. For each figure, it shows memory access of these three algorithms for different table sizes. Memory access represents the average number of memory access for performing one packet classification in Software-based OpenFlow switch. We first observe figure 8, the result of the acl rules table, which shows that the performance of our approach is better than others as a small table size and Bitmap Intersection also shows great performance which is quite same as our approach. And the R*-Tree algorithm has the worst performance. When the size of rule table grows, it is obviously that Bitmap Intersection's performance is decreasing. It is because the bit vector length increases as the table size grows, thus it requires more memory access to access the bit vector, which decreases the performance of Software-based OpenFlow switches. Comparing our approach with Bitmap Intersection, it shows that the memory access of our approach also increases as table size grows, but the increase in ratio is much less than that of Bitmap Intersection. When the number of rules grows to 10000, the performance of our algorithm is 3.95 times of Bitmap Intersection, and 5 times that of R*-Tree algorithm. Besides, there is an interesting phenomenon, which happens on our algorithm and R*-Tree algorithm, where the memory access decreases as table size increases for some cases. It happens because the major factor affecting performance of our algorithm is not the number of rules, but the distribution of our algorithm is not the number of rules, but the distribution of rules in geometric space is the true factor that affects performance. If rules overlap with each other in geometric space, the searching should take more paths, which need more memory access. In general, the spatial objects are going to overlap with each other as we put more objects in the space. But things are different if we consider the distribution, for the following two situations, a few objects concentrated in a small area and many objects separated in a wide area. The objects are going to overlap with each other in the first case, so searching is going to spend more time. In the second case, even though there are many objects, they are separated in a wide area. They are not going to overlap, thus searching will take less time in this case. This is the reason that the phenomenon happens.

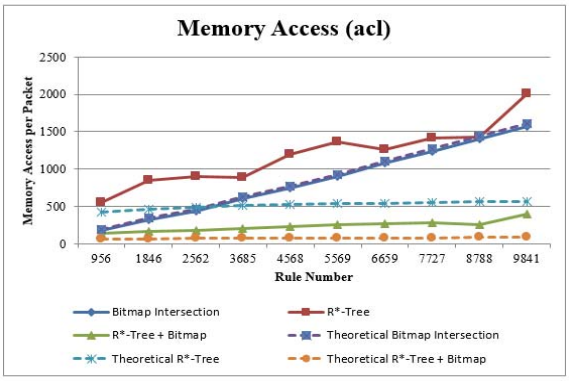


Fig. 8. Memory access (acl rules table)

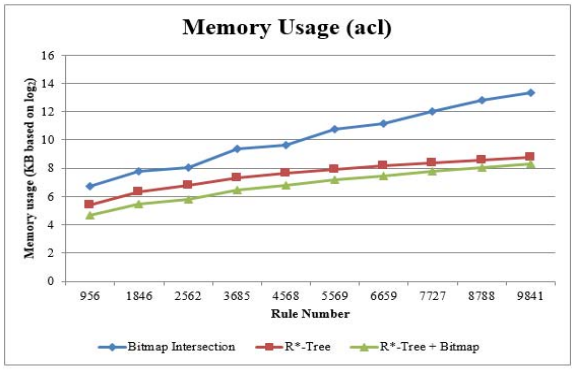


Fig. 9. Memory usage (acl rules table)

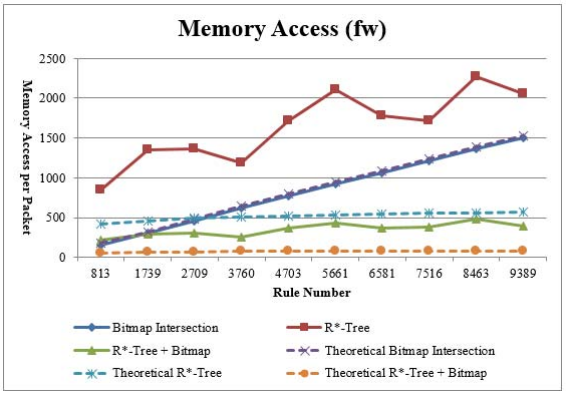


Fig. 10. Memory access (fw rules table)

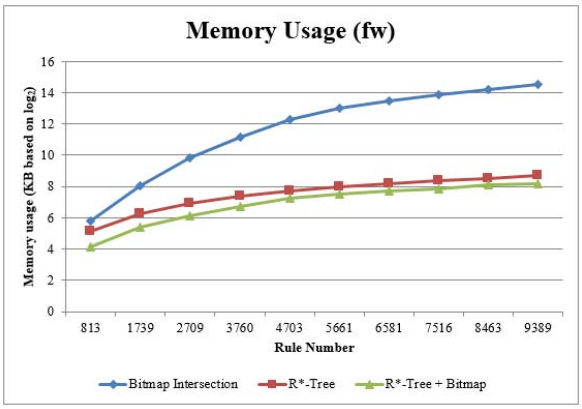


Fig. 11. Memory usage (fw rules table)

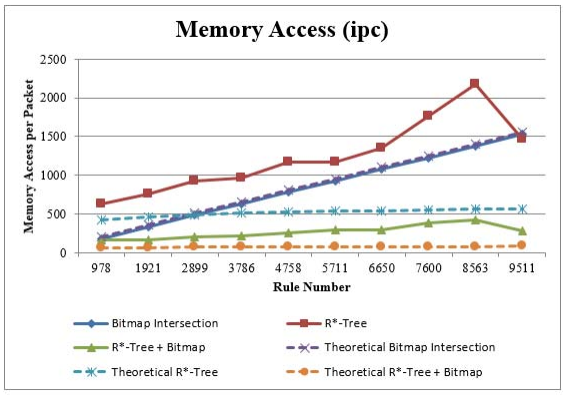


Fig. 12. Memory access (ipc rules table)

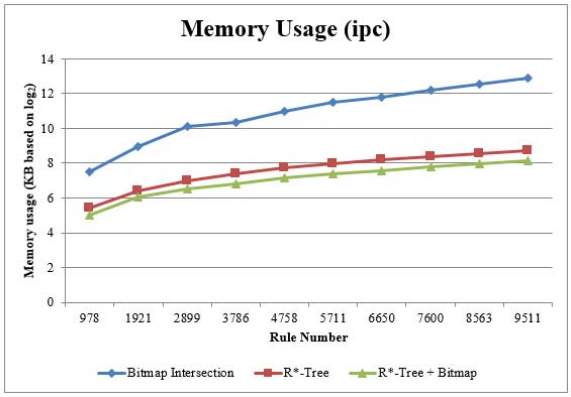


Fig. 13. Memory usage (ipc rules table)

Then we observe figure 10, the simulation result of fw rules table. The result is different from acl rules table. When the table size is small, the Bitmap Intersection has the performance, while our algorithm's performance is slightly worse than Bitmap Intersection's. When table size increases, it shows the same phenomenon as acl rules table, where the memory access of Bitmap Intersection increases as table size grows. Our algorithm's memory access increases as well, but the increase in ratio is also less than that of Bitmap Intersection. When the number of rules grows to 10000, the performance of our algorithm is 3.8 times that of Bitmap Intersection, and 5.18 times that of R*-Tree algorithm. As we compare the results between fw rules table and acl rules table; we find out that our algorithm has poor performance in fw rules table. The reason is that the rules are going to overlap with each other in fw rules table, which will decrease searching performance. But Bitmap Intersection's performance does not decrease in fw rules table, because the major factor that affects the performance of Bitmap Intersection is the number of rules, thus its performance is quite similar for different distributions.

Observe figure 12, the simulation result of ipc rules table. The results shown here are similar to the results of acl rules table. When table size is small, our algorithm has the greatest performance, and the performance of Bitmap Intersection is smaller. The memory access also increases as table size grows, and the increase in ratio of our algorithm is the smallest. The memory access also increases as table size grows, and the increase in ratio of our algorithm is the smallest. When the number of rules grows to 10000, the performance of our algorithm is 5.5 times that of Bitmap Intersection, and 5.3 times that of R*-Tree algorithm. Summarizing these three simulation results, our algorithm's performance is similar to Bitmap Intersection when

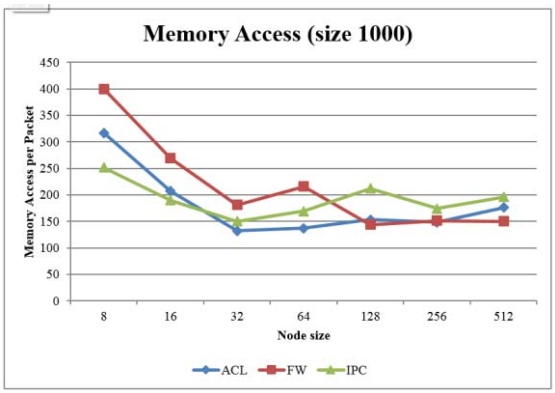


Fig. 14. Memory access of different node size with 1000 rules

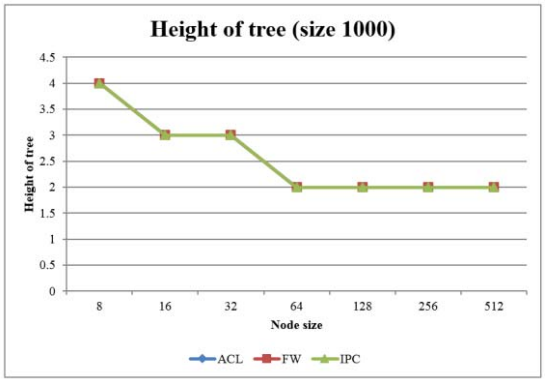


Fig. 15. Tree height of differ ent node size with 1000 rules

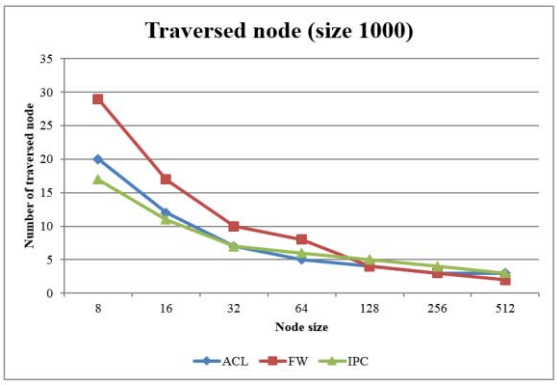


Fig. 16. Traversed node of different node size with 1000 rules

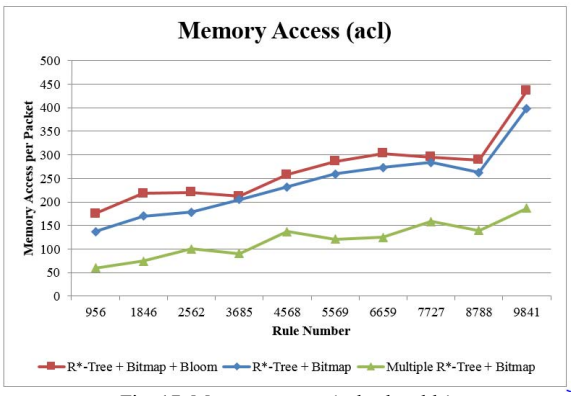
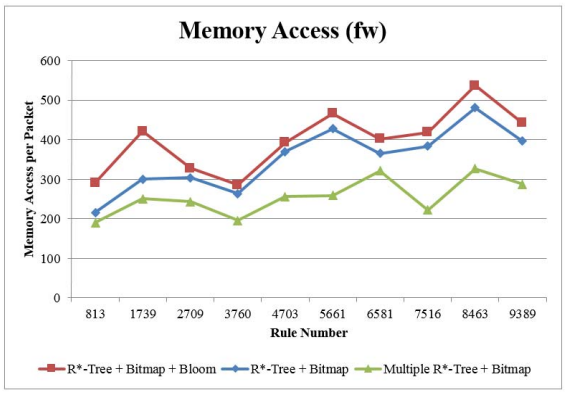


Fig. 17. Memory access (acl rule table)



Fi g. 18. Memory access (fw rule table)

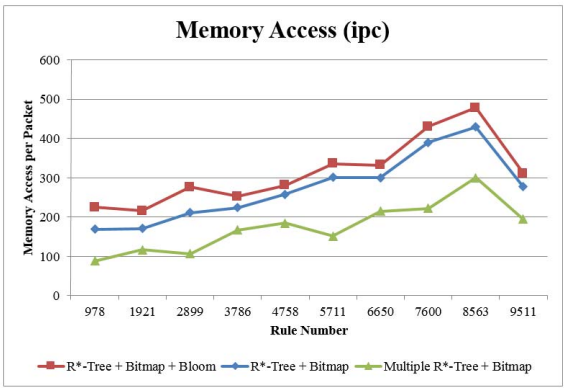


Fig. 19. Memory access (ipc rule table)

the table size is small. But our performance becomes better than others as the number of rules grows to 10000, the performance of our algorithm is 4.42 times that of Bitmap Intersection, and 5.16 times that of R*-Tree algorithm.

The simulation results of memory usage for different rules table are represented in figure 9, 11, and 13 respectively. Because the difference between these three algorithms is huge, we take a $\log_2$ here for ease of observing the data. For different types of rules tables, the memory usages of our algorithm are very similar, and have the same trend. But Bitmap Intersection's results are very different between rules tables. When the number of rules is 10000, Bitmap Intersection requires 10 MB for acl rules table, 20 MB for fw rules table, and 7 MB for ipc rule tables. The memory usage of our algorithm and R*- Tree algorithm are similar. Our algorithm requires about 300 KB and R*-Tree requires about 420 KB for these three different rules tables. It is obvious that our algorithm has the greatest performance on memory usage while simulating with Software-based OpenFlow switches. And the simulation proves that the memory usage of Bitmap Intersection grows quadratically and our algorithm and R*Tree algorithm are grown linearly as the number of rules increases.

B. The influence of R-Tree node size*

We will simulate our algorithm with different sizes and types of rules table, to observe memory access, tree height, and number of traversed nodes, as well as discuss the influence of node size.

The simulation results of memory access, tree height, and traversed nodes are given in figures 14, 15, and 16 respectively, and the size of the rules tables is 1000. We find out that the node size affects performance very much, the greatest performance is 2.78 times the worst performance in fw rules table, and the

difference is about 2 times for other rules tables. And the tree height affects performance very much, it takes more memory access when tree height is higher. The reason of this phenomenon we can understand through the result of traversed nodes. When the tree height is 4 (i.e. node size 8), its average traverses 29 nodes for each packet classification. And its average traverses 16 nodes when the tree height is 3 (i.e. node size 4). It is obvious that the number of traversed nodes decreases as tree height decreases if the memory access spent in one node is equal with fewer traversed nodes representing better performance. Then we compare node size 32 with node size 64. Node size 64 gets lower tree height and smaller traversed nodes, but the simulation shows that node size 32 gets better performance on memory access. It is because the memory access spent in one node increases as the node size increases. So node size 64 takes more memory accesses even though it performs better on tree height and traversed node. And then, we discuss the influence of different node sizes with the same tree height. We observe the results of node size 64, 128, 256, and 512, where their tree height is 2.

The number of traversed nodes decreases, but the memory access increases as node size grows for the reason that we mentioned above. When the rules number is 1000, it shows the greatest performance with node size 32.

C. The influence of Bloom Filter and Multiple R-Tree*

We employ a Bloom Filter to reduce unnecessary paths and use multiple R*-Trees to reduce necessary paths. And we will use acl, fw, and ipc three rules tables, to observe the difference in memory access. The simulation results of acl, fw, and ipc rules tables are represented in figures 17, 18, and 19 respectively. From these results, we find out that they all have the same trend for these three different rules tables. The performance decreases with Bloom Filters, and improves with multile R*-Trees. First, we discuss the reason that performance decreases with Bloom Filters. We find out that unnecessary path has a very low ratio of all paths. Actually, we could avoid these unnecessary paths through Bloom Filter, but the cost spent on the execution of Bloom Filter exceeds the cost which is saved from discarding necessary paths. So overall, the memory access increases. Then when observing the simulation result of multiple R*-Trees, we find out that performance gets lot of improvement. On average, the performance improves by about 30%, which means multiple R*-Trees actually separates necessary paths into two trees, and if packets get a result from the first tree, then it is not necessary to search the second tree. This will reduce necessary paths during performing packet classification in Software-based OpenFlow switches, which decreases the amount of memory access efficiently. Summarizing all the simulation results, the speed of packet classification in our algorithm has the same performance as Bitmap Intersection when table size is small. But when the table size is large, the performance of our algorithm is better than others. And the memory usage of our algorithm is much better than Bitmap Intersection, so our algorithm requires less memory access than Bitmap Intersection for packet classification. In summarizing, our algorithm is better than Bitmap Intersection and R*-Tree in the aspects of matching speed and memory usage. And the multiple R*-Trees actually improved the performance of our algorithm.

VI. CONCLUSION AND FUTURE WORK

We consider the next-generation packet classification problems for Software-based OpenFlow switches where more than 5-tuple packet header fields would be classified. We analyze and compare our proposed approaches with Bitmap Intersection and R*-Tree in time and space complexity. We observe that the major influence in searching performance is the number of traversed paths. When the number of traversed path grows, the time spent on search also increases, so it takes more time to finish a packet match processing. In order to enhance the memory performance of Software-based OpenFlow switches, we furthermore proposed two enhanced approaches to improve our algorithm. Firstly, we use a Bloom Filter to reduce the number of unnecessary paths and secondly, we use multiple R*-Trees to separate necessary paths into R*Trees. Simulation results show that the average performance of our algorithm is 4.42 times that of Bitmap Intersection, and 5.16 times that of R*-Tree algorithm. For memory usage, our algorithm only requires 300 KB, R*-Tree requires 420 KB, and Bitmap Intersection requires 10 MB with 10000 rules. Our algorithm has the greatest performance both on a classification of the speed and memory usage while using Software-based OpenFlow switches. For the two enhanced approaches, simulation results show that the Bloom Filter will decrease performance, but multiple R*-Trees improve performance greatly, by about 30%. In future works, we will still examine the Bloom Filter approach with the situation where the size of rules table is huge and the rules overlap with each other very much, to find out if the Bloom Filter approach could improve performance in this situation.

ACKNOWLEDGMENT

This work is particularly supported by the Chunghwa Telecom Laboratories and "Aiming for the SPROUT Project – Center for Open Intelligent Connectivity" of National Chiao Tung University and Ministry of Education, Taiwan, R.O.C.

REFERENCES

[1] K. Qiu, Z. Chen, Y. Chen, J. Zhao and X. Wang, "GFlow: Towards GPU based High-Performance Table Matching in OpenFlow Switches", IEEE ICOIN 2015.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69–74, 2008

[3] OpenFlow Foundation, "OpenFlow Switch Specification, Version 1.0.0," 2009. [Online]. Available: http://www.openflowswitch.org/ documents/openflow-spec-v1.0.0.pdf

[4] D. Taylor, "Survey & Taxonomy of Packet Classification Techniques," ACM Comput. Surv., vol. 37, no. 3, pages 238–275, 2005.

[5] D.E. Taylor and J.S. Turner, "ClassBench: A Packet Classification Benchmark," In proceedings of IEEE INFOCOM, 2005.

[6] G. Varghese, "Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices," Morgan Kaufmann publishers, 2005.

[7] H. J. Chao and B. Liu, "High Performance Switches and Routers," Wiley-IEEE Press, 2007.

[8] C. Maindorfer and T. Ottmann, "Is the Popular R*-Tree Suited for Packet Classification?," In proceedings of IEEE NCA, pages 168-176, 2008.

[9] Gibb, G., Varghese, G., Horowitz, M., and Mckeown, N., "Design Principles for Packet Parsers" In Architectures for Networking and Communications Systems (ANCS), ACM/IEEE Symposium, pp. 13-24, 2013.

[10] Y. Luo, P. Cascon, E. Murray, and J. Ortega, "Accelerating OpenFlow switching with network processors," In Proc. ANCS, pp. 70–71, 2009.

[11] I. Sourdis, "Designs & algorithms for packet and content inspection" Ph.D. dissertation, Comput. Eng. Div., Delft Univ. Technol., Delft,The Netherlands, 2007. [Online]. Available: http://ce.et.tudelft.nl/publicationfiles/1464_564 sourdis phdthesis.pdf

[12] H. Yu and R. Mahapatra, “A power- and throughput-efficient packet classifier with n bloom filters,” IEEE Trans. Comput., vol. 60, no. 8, pp. 1182–1193, Aug. 2011.

[13] M.Kuzniar, P.Peresini, and D.Kostic, “What you need to know about SDN flow tables”, In PAM 2015.

[14] M.Kuzniar, P.Peresini, and D.Kostic, “What you need to know about SDN control and data planes”, Technical Report EPFL_REPORT_199497, EPFL, 2014.

[15] B. Lantz, B. Heller, and N. McKeown, “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks”, In ACM SIGCOMM HotNets Workshop, 2010.

[16] Open Networking Foundation (ONF), “Software defined networking: the new norm for networks,” 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/openflow/wpsdn-newnorm.pdf

[17] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “OFLOPS: An open framework for openflow switch evaluation”, In PAM, 2012.

[18] D. Y. Huang, K. Yocum, and A. C. Snoeren, “High-fidelity switch models for software-defined network emulation”, In HotSDN, 2013.

[19] P. Peresini, M. Kuzniar, M. Canini, and D. Kostic, “ESPRES: Transparent SDN Update Scheduling”, In HotSDN, 2014.

[20] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, “A clean slate 4D approach to network control and management”, SIGCOMM Comput. Commun. Rev., 35(5):41–54, 2005.

[21] Steven J. Vaughan-Nichols, “OpenFlow: The Next Generation of the Netwok? “, IEEE Computre Society, 2011.

[22] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, “OpenFlow Switching: Data Plane Performance”, IEEE ICC proceedings 2010.