2013 IEEE 37th Annual Computer Software and Applications Conference Workshops

Packet Classification Using Multi-Iteration RFC

Chun-Hui Tsai, Hung-Mao Chu, Pi-Chung Wang
Department of Computer Science and Engineering
National Chung Hsing University
Taichung 402, Taiwan
hwei519@gmail.com, chuhungmao@gmail.com, pcwang@nchu.edu.tw

Abstract—Packet Classification is an enabling technique for the future Internet by classifying incoming packets into forwarding classes to fulfill different service requirements. It is necessary for IP routers to provide network security and differentiated services. Recursive Flow Classification (RFC) is a notable high-speed scheme for packet classification. However, it may incur high memory consumption in generating the pre-computed cross-product tables. In this paper, we propose a new scheme to reduce the memory consumption by partitioning a rule database into several subsets. The rules of each subset are stored in an independent RFC data structure to significantly alleviate overall memory consumption. We also present several refinements for these RFC data structures to significantly improve the search speed. The experimental results show that our scheme dramatically improves the storage performance of RFC.

Keywords-packet classification; packet forwarding; firewalls; QoS

I. INTRODUCTION

Packet classification identifies flows among a stream of packets that arrive at routers. It is an enabling technology for the future Internet to support access control, quality of service guarantees and differentiated services. Packet classification is essentially a problem of multidimensional range matching, which compares the header fields of incoming packets against a set of pre-defined rules. The header fields include source IP prefix, destination IP prefix, source port range, destination port range, and protocol number. Each rule indicates an action for processing the matching packets and a priority value to indicate its precedence among the matching rules. For an incoming packet, packet classification yields the matching rule with the highest priority. The performance of a packet classification algorithm is measured by its memory consumption and number of memory accesses to accomplish a classification.

Currently, packet classification algorithms have different tradeoffs between storage and speed performance. For example, the algorithms based on hash tables have superior space performance, but their speed performance cannot be guaranteed [5][6]. Decision-tree-based algorithms use a decision tree to divide rules into multiple linear-search groups [7][8]. The speed and the storage performance would vary according to the characteristics of rule databases. EffiCuts uses multiple decision trees to control memory consumption, but also degrades the speed performance [9]. Ternary content addressable memories (TCAMs) have been widely used to lookup rules. However, TCAMs cannot store ranges; thus, range-to-prefix transformation is required to degrade storage efficiency [11][12]. RFC [2] is a notable high-performance algorithm based on cross-producting. It uses multiple iterations of cross-producting to classify packets. While RFC outperforms the existing cross-producting algorithms [2][3][4] in search performance, it is not feasible for large rule databases since high memory consumption in generating the cross-product tables is incurred.

In this paper, we propose a new packet classification scheme based on RFC. Our scheme partitions a rule database into several subsets where the rules of each subset are stored in a RFC data structure. By controlling the number of rules in a subset, our scheme can avoid generating huge cross-product tables. Consequently, RFC is executed in a recursive fashion to determine which subsets should be accessed and which rules in a subset are matched. We also present several refinements to improve both storage and speed performance. The experimental results show that the new scheme significantly improves the feasibility of RFC for large rule databases while maintaining the superior speed performance.

The rest of paper is organized as follows. Section II and III presents the proposed scheme and refinements, respectively. The experimental results are show in Section IV; a summary is given in Section V.

II. PROPOSED SCHEME

We aim at improving the storage efficiency of RFC by employing multiple RFC instances. First, we partition a rule database into several subsets. The rules of each subset are stored in an independent RFC data structure. Each subset is then represented by an index rule and each index rule points to the corresponding RFC data structure. Thus, if we partition the database into k subsets, then k index rules are created. These index rules are stored in another RFC data structure (index RFC). With the index RFC, we can determine which subsets an incoming packet matches. Next, the corresponding RFC data structures are accessed to determine the matching rules.

We describe the reasoning for partitioning a rule database by using an example. Table I is a rule database with six two-field rules. In the source address (SA) field, there are five combinations: 0* (R3,R6) , 010* (R3,R4,R6), 1* (R2,R6) , 1100 (R1,R2,R6), and 1110 (R2,R5,R6),where the identifiers shown in the parenthesis are the matching rules for the corresponding prefix. In the destination address (DA) field, there are six combinations: * (R5), 110* (R5,R6), 1011 (R1,R5), 0* (R4,R5), 010* (R2,R4,R5), and 00* (R3,R5). As a result, there are 30(=6*5) entries after cross-producting both fields. The number of cross-product entries can be reduced by partitioning a rule database. Fig. 1 illustrates these rules geometrically. These rules are divided into three subsets, (R1,R5,R6), (R2,R5), and (R3,R4). The number of cross-product entries for each subset is 9(=3*3), 4(=2*2), and 4(=2*2). As a result, the total number of cross-product entries is reduced to 17. As compared with the original cross-product table, database partitioning can effectively reduce the cross-product entries. Accordingly, we improve the storage efficiency of RFC by using database partitioning.

978-0-7695-4987-3/13 $26.00 © 2013 IEEE
DOI 10.1109/COMPSACW.2013.87

IEEE computer society

TABLE I. A TWO-FIELD RULE DATABASE.

	SA	DA		SA	DA
R1	1100	1011	R4	010*	0*
R2	1*	010*	R5	1110	*
R3	0*	00*	R6	*	110*

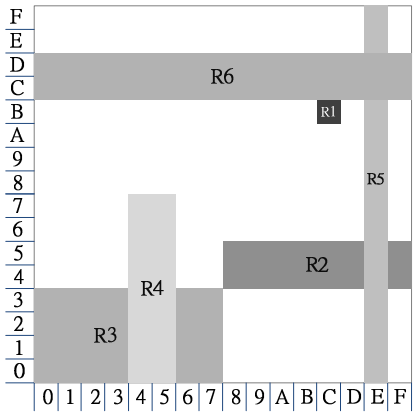

Figure 1. Geometric illustration of rules in Table I.

An effective partitioning algorithm should meet several requirements. First, the rules which are geometrically close to each other should be categorized in the same subset. This requirement can avoid the search procedure to access all subsets. Second, each rule should reside in exact one subset. A less efficient partitioning technique may incur replicated rules to result in extra storage. Third, the number of rule subsets should be adjustable to accommodate different rule databases. In the following, we investigate the current techniques for partitioning a rule database.

The idea of tuple space divides a rule database into tuples based on the number of bits specified in each field. Each tuple corresponds to a prefix-length combination of all inspected fields, and the resulting set of tuples is called tuple space [1]. For example, a five-dimensional tuple, (8,16,7,0,8), collects the rules whose first field is an 8-bit prefix and the second field is a 16-bit prefix and so on. Since each rule has only one prefix-length combination, tuple space does not incur any rule replication. However, a prefix-length combination does not imply any geometric relations, tuple space cannot meet our first requirement. It is also difficult to adjusting the number of tuples due to the high cost of prefix expansion. A similar idea of tuple space is proposed by using the nested-level tuple, where the length of each field is defined as the number of nested levels for the corresponding prefix. Although the number of nested-level tuples is significantly less than the number of tuples, the first requirement is still not supported. A greedy approach is proposed to reduce the number of nested level tuples by using the technique of cross-producting, but the problem of rule replication remains. Nested-level tuples also do not support updates since inserting a rule with a new prefix can change the nested levels of all rules.

The decision-tree algorithms can divide a database into subsets by using field attributes of a rule. When the attribute used for partitioning a database is the field values, decision tree provides a geometrical approach that the rules in the same subset are close to each other. As a result, only one subset of a decision tree is accessed while performing packet classification. The number of subsets can be controlled by adjusting the number of rules in a leaf node. However, rule replication is a persistent problem of a decision tree. Since wildcard specification is common in a rule database, a geometrical approach to partition a rule database can only minimize the replicated rules, rather than avoid rule replication. In the previous work, several approaches to minimizing replicated rules are proposed [8]. Several algorithms use multiple decision trees to improve the efficiency of rule partitioning [9]. The other algorithms exploit different attributes for partitioning a rule set. None of these approaches can completely avoid the problem of rule replication with a reasonable cost.

As compared with tuple-based partitioning approaches, rule replication is only one problem to overcome for using decision tree to partition a rule database. We use an on-demand approach to avoid the problem of rule replication. Our approach first generates a balanced binary decision tree where each internal node divides the associated rules into two subsets. In the procedure of constructing a decision tree, any replicated rules are removed. All rules which are removed from the first decision tree are then stored in the second decision tree. The second decision tree is then constructed based on the above procedure. Any replicated rules in the second decision tree are then moved to the third decision tree, and so on. After generating all decision trees, the rules in a leaf node are inserted into an RFC data structure. Thus, the number of RFC data structures is equal to the total number of leaf nodes in all decision trees.

The detailed procedure of rule partitioning is described below. We first define a bucket size to limit the number of rules stored in an RFC data structure. All rules are associated with the root node of the first decision tree. If the number of rules is larger than the bucket size, then the rules are divided into two subsets. To partition a rule set, we select a field which can effectively distinguish these rules. We calculate the number of distinct prefixes of each field for the rule set and choose the field with the largest number for rule partitioning. For the selected field, we further determine an address point which can equally divide the rule set into two parts. We calculate the number of rules whose end points of the selected field are less than or equal to a given address point and number of rules whose starting points of the selected field are larger than the given address point for each end point. The end point whose numbers are the closest is selected. With the selected address point, we can divide the rule set into three subsets: the rules whose ranges are lower than the selected address point, the rules whose ranges are higher than the selected address point, and the uncategorized rules whose ranges are across the selected address point. The first two sets can be further divided until each generated subset has less number of rules than the bucket size by repeating the above steps. All uncategorized rules in the decision tree are inserted into the root node of the next decision tree for further partitioning.

We illustrate the above procedure by using an example with seventeen five-field rules, as listed in Table II. We set the bucket size to four. In the first iteration of rule partitioning, there are the most distinct prefixes in source address field; thus, we select this field to partition the rules into three subsets, where each subset corresponds to a node of the tree. As shown in Fig. 2, the left child of the root node stores the rules whose source address field is lower than the selected address, and the right child of the root node stores the rules with larger source address field. The middle child of root node stores the uncategorized rules. Because both left and right child nodes have more than four rules, they should be further partitioned to generate smaller subsets. All the uncategorized rules in the decision tree (including R15, R16, R12, and R13) are removed to the second decision tree. Due to the size of the subset corresponding to the root node is not larger than the threshold, the second decision tree only has one node.

TABLE II. THE FIVE FIELD RULE-DATABASE.

	Source address	Destination address	Source port	Destination port	Protocol
R0	100.100.198.45/32	128.17.88.0/24	[0:1024]	[80:80]	0x06/0xFF
R1	69.250.70.0/32	255.255.255.255/32	[151:151]	[81:81]	0x06/0xFF
R2	128.23.128.0/24	128.17.88.0/23	[80:80]	[150:160]	0x07/0xFF
R3	32.9.136.0/25	200.16.14.0/24	[79:~80]	[80:80]	0x01/0xFF
R4	88.79.0.0/13	192.192.69.69/32	[514:514]	[36:136]	0x06/0xFF
R5	200.55.0.0/13	128.17.88.0/24	[2435:2436]	[55444:55444]	0x03/0xFF
R6	64.63.0.0/16	128.17.88.0/24	[0:1024]	[0:65535]	0x00/0x00
R7	128.128.0.0/16	0.0.255.255/32	[5000:6000]	100 ~ 100	0x07/0xFF
R8	100.100.0.0/16	79.0.0.0/8	[514:514]	[120:120]	0x06/0xFF
R9	5.64.0.0/10	32.0.128.0/10	[35543:65535]	[83:83]	0x00/0x00
R10	224.145.0.0/9	0.0.0.8/32	[80:80]	[0:1024]	0x03/ 0xFF
R11	130.87.0.0/10	64.64.90.0/18	[3680:6887]	[0:1024]	0x03/0xFF
R12	128.34.0.0/24	0.0.0.0/0	[1025:1025]	[1221:1228]	0x03/0xFF
R13	254.80.0.0/16	0.0.0.0/0	[1025:1025]	[55444:55666]	0x08/0xFF
R14	254.80.0.0/16	180.37.0.0/16	[17:30]	[55222:55333]	0x08/0xFF
R15	0.0.0.0/0	128.128.64.0/24	[98:98]	[135:135]	0x00/0xFF
R16	0.0.0.0/0	0.0.0.0/0	[0:65535]	[0:65535]	0x00/0x00

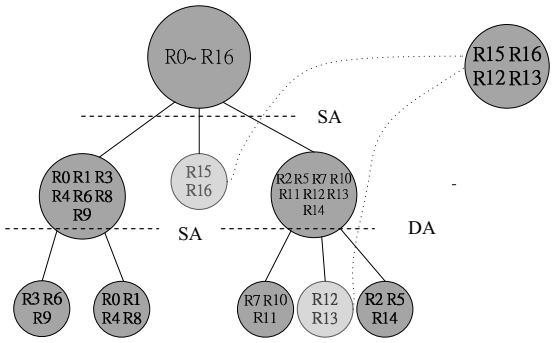

Figure 2 The decision trees for the example in Table 2.

After partitioning a rule-database into several subsets, the rules of a subset is stored in an RFC data structure and use an index rule to represent the space of a subset. Each range of the index rule starts from the smallest starting point to the largest end point of the corresponding field of all rules in the subset. Therefore, if we partition a rule-database into k subsets, we create k index rules. Table III lists the index rules and their corresponding ranges in each field for the previous example in Table II. After creating the index rules for all subsets, we use an RFC data structure to store these index rules, called index RFC.

For each RFC data structure, five filter fields are split into seven chunks, including six 16-bit chunks and one 8-bit chunk in the first phase. For each chunk, a 2^W-entry index array is constructed for accessing the equivalence class ID (eqID) corresponding to the value of a packet header field, where w denotes the chunk size. Each eqID is associated with a class bitmap to indicate the rules matching the chunk equivalence set. Each class bitmap of an eqID is different. Two or three chunks are combined to generate a chunk in the next phase by cross-producting their eqIDs. The class bitmap of a new chunk is equal to the intersection of the class bitmaps of the merged eqIDs. Each distinct class bitmap represents an equivalence set in the new phase. Each equivalence set is then assigned an eqID. The new eqIDs are stored in an index array whose size is equal to the multiplication of the number of merged eqIDs. The procedure proceeds until all chunks are merged. For an incoming packet, the search procedure in an RFC data structure starts by splitting the packet header into seven chunks. The value of each chunk is used to access the eqID in the index array. If there is any subsequent phase, then the search procedure uses the combination of the fetched eqID to generate the index of the next phase. As the procedure traverses to the last phase and fetches the eqID, the class bitmap corresponding to the eqID is accessed to determine the matching rules.

For an incoming packet, the complete search procedure starts by traversing the index RFC data structure to find the matching index rules. Then, the search procedure proceeds to search the subsets of the matching index rules by accessing the corresponding RFC data structures. The framework of our algorithm consists of *six* RFC data structures, *five* for the resulted subsets and *one* for the index rules. Table IV shows the cross-producting table entries in each phase for the original RFC and our algorithm. In this example, we reduce 63% entries of the original RFC.

TABLE III. THE INDEX RULES FOR THE RULES IN TABLE II.

	Source address	Destination address	Source port	Destination port	Protocol
Index Rule 0	[5.64.0.0: 64.63.255.255]	[32.0.0.0: 200.16.14.255]	[0:65535]	[0:65535]	[0:255]
Index Rule 1	[69.250.70.0: 100.100.255.255]	[79.0.0.0: 255.255.99.191]	[36:136]	[0:1024]	[6:6]
Index Rule 2	[128.128.0.0: 225.16.255.255]	[0.0.0.8: 64.64.153.255]	[80:6887]	[0:1024]	[3:7]
Index Rule 3	[128.23.128.0: 254.80.255.255]	[128.17.88.0: 180.37.255.255]	[17:2436]	[150:55444]	[3:8]
Index Rule 4	[0.0.0.0: 255.255.255.255]	[0.0.0.0: 255.255.255.255]	[0:65535]	[0:65535]	[0:255]

TABLE IV. THE CROSS-PRODUCTING TABLE ENTRIES IN EACH PHASE FOR THE ORIGINAL RFC AND OUR ALGORITHM.

RFC				Our Scheme				Total entries	
	Phase 1	Phase 2	Phase 3		Phase 1	Phase 2	Phase 3	RFC	Our Scheme
Entries	1905	224	420	Entries	665	133	155	2549	953

III. REFINEMENTS

In this section, we present three techniques to further improve both speed and storage performance.

A. Merging Small Subsets

While partitioning a rule database, small subsets could be generated. These small subsets would result in less efficient RFC data structure. Extra memory accesses to these data structures are also incurred. To avoid generating small rule subsets, we merge the subsets whose numbers of rules are smaller than a threshold. These rules of the merged groups are thus stored in the same RFC data structure.

B. Merging the First Phases of Different RFCs

As mentioned above, we need k+1 RFC data structures for a database partitioned into k groups. Since each of RFC data structure is traversed independently, we need 7*(k+1) memory accesses to retrieve eqIDs in the index arrays of the first phase. To reduce the number of memory accesses, we merge the index arrays of the same chunk from different RFC data structures, where each entry in the new array has k+1 fields. Each field maps to one eqID of different RFCs. Accordingly, we can fetch the eqIDs of the same chunk in different RFC data structures with one memory access. The number of memory accesses is thus reduced from 7*(k+1) to seven for the first phase of all RFCs.

The first phase of each RFC data structure stores eqIDs for six 16-bit chunks and one 8-bit chunk. The lookup table of each 16-bit chunk is a 2^{16}-entry index array and that for the 8-bit chunk is a 2^8-entry index array. If we partition a database into k subsets,

we will need $6\times2^{16}\times(k+1)+2^{8}\times(k+1)$ array entries in the first phase. In order to reduce the memory consumption, we replace the index array with a binary-search array for the first-phase eqIDs. For each index array, the eqIDs stored in contiguous entries could be the same. We can merge them into an interval, which starts from the first entry to the last entry with the same eqID. In this way, we can transform a 2^{w}-entry index array into an n-interval array, which can be binary searched. This approach can reduce memory consumption in the first phase. We note that we have merged the first phase of k+1 RFC data structures; therefore, the prerequisite for merging contiguous entries into one interval is modified from "with the same eqID" to "with the same (k+1) eqIDs".

IV. Experimental Results

In this section, we use both real and synthetic filter databases to evaluate the performance of the proposed scheme. We use three types rule sets in our experiment: access control list (ACL), firewall (FW) and IP chains (IPC). All the databases are publicly available in [14]. We also compare the proposed scheme with several existing schemes.

The experimental result consist of three parts, the first part shows the tradeoff between speed and storage performance with different subset size. The second part demonstrates the performance improvement based on various threshold values for subset merging. The last part is a performance study that compares our scheme with the existing schemes.

A. Different Subset Size

For the first part, the number of rules in a subset is determined by using a divisor. With the defined divisor d_1, the subset size is equal to the total number of rules divided by d_1. A rule set is partitioned until the number of rules in each subset is less than the threshold value. We use three divisors, 4, 8, and 16, in the following evaluation and choose the one with the best performance.

Fig. 3 shows the memory requirement and the numbers of memory accesses in the worst case for three different types of databases with three different subset sizes. As shown in Fig. 3(a), the memory requirements degrade gradually along with a smaller subset size and a small group size usually leads to low memory requirement for cross-producting tables. However, Fig. 3(b) also shows that with more subsets generated, more memory accesses are needed to accomplish a classification. This is because an incoming packet may match multiple index rules in the index RFC and the corresponding RFC data structures must be accessed. After comparing these group sizes, we set the group divisor d_1 to 8 since it can better leverage the storage and speed performance. The subset size is thus equal to the number of rules divided by 8.

After partitioning a database into several subsets, the subsets with few rules may decrease the overall search performance since their RFC data structures store relatively few rules. In addition, extra memory accesses might be incurred if an incoming packet matches these subsets.

B. Different Threshold Values for Subset Merging

In the second part, we set a threshold to improve the performance by merging the small subsets. The merge threshold is also determined by using a divisor d_2, where the merge threshold is equal to the subset size divided by d_2. The subsets are merged if their sizes are smaller than the merge threshold. We use three divisor values, 2, 3, and 4, in the following evaluation and choose the one with the best performance.

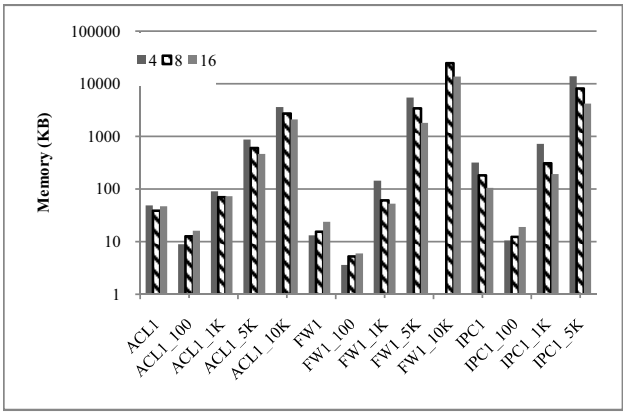

Figure 3(a). Storage performance for three databases in three kinds of thresholds.

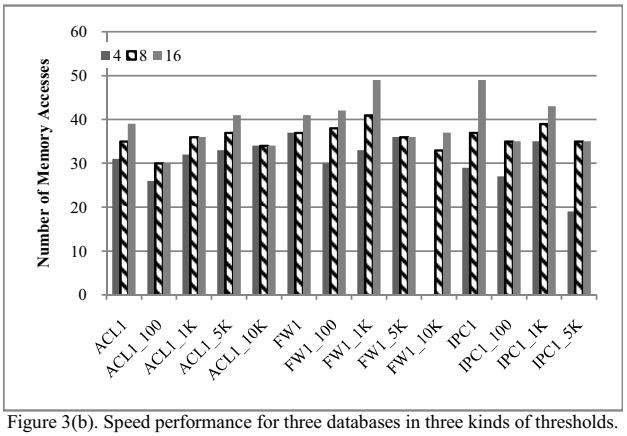

Figure 3(b). Speed performance for three databases in three kinds of thresholds.

Fig. 4 shows the memory requirement and the number of memory accesses in the worst case for three types of databases with three different thresholds for subsets merging. Fig. 4(a) shows that a large threshold may incur more memory requirements since more subsets are merged to result in larger cross-producting tables in RFC. Fig. 4(b) shows that a small merge threshold may incur more memory accesses for several databases because the number of RFC data structures cannot be effectively reduced. As a result, an incoming packet may match more subsets to lead to more memory accesses in the search procedure.

We further compare the performance of our scheme with and without subset merging by setting the divisor d_2 to 3. Fig.5(a) shows that the memory requirements increase slightly with subset merging since an RFC data structure storing more rules usually results in more cross-producting entries. Fig. 5(b) shows that subset merging can reduce the number of memory accesses. Although the tradeoff between storage and speed performance is present, the speed improvement is preferable since our algorithm has significantly reduced memory requirement of RFC.

C. Comparative analysis

In the last part, we compare the performance of our optimized scheme with four existing schemes, including RFC [2], HSM [3], Hypercuts [8] and ISET [13]. We also use three types of rule sets in the comparisons, as shown in Fig. 6~8. Some results cannot be generated because the programs for building data structures ran

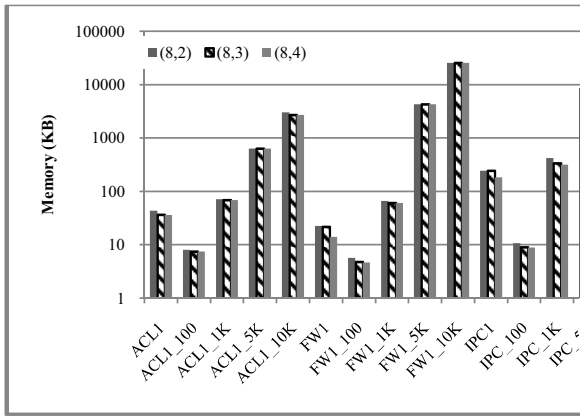

Figure 4(a). Storage performance for three databases in three merge thresholds.

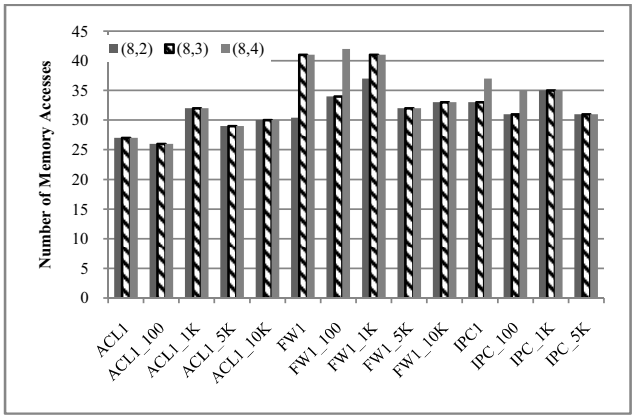

Figure 4(b). Speed performance for three databases in three merge thresholds.

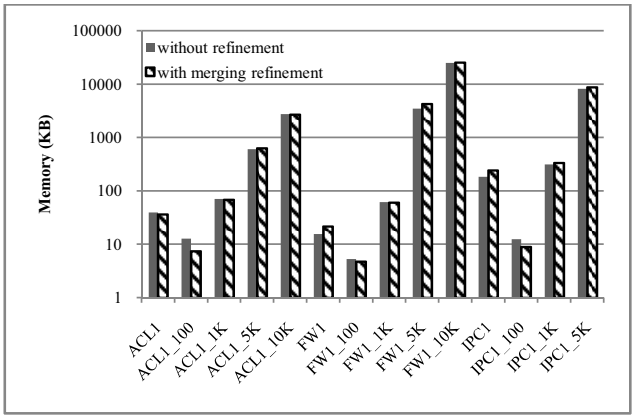

Figure 5(a). Storage performance of our scheme with and without group merging.

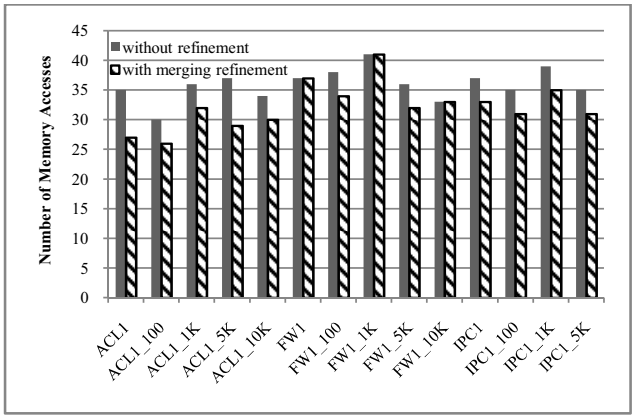

Figure 5(b). Speed performance of our scheme with and without group merging.

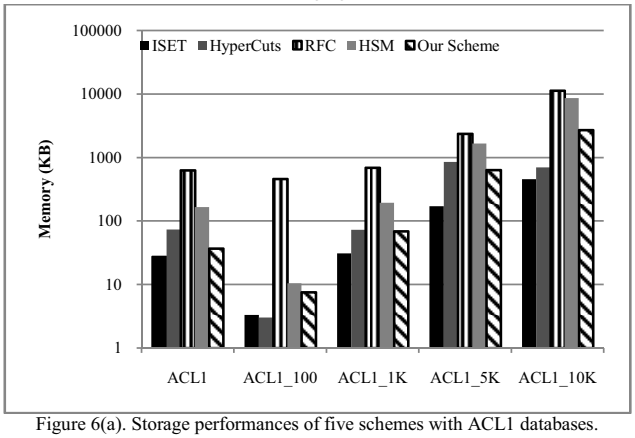

Figure 6(a). Storage performances of five schemes with ACL1 databases.

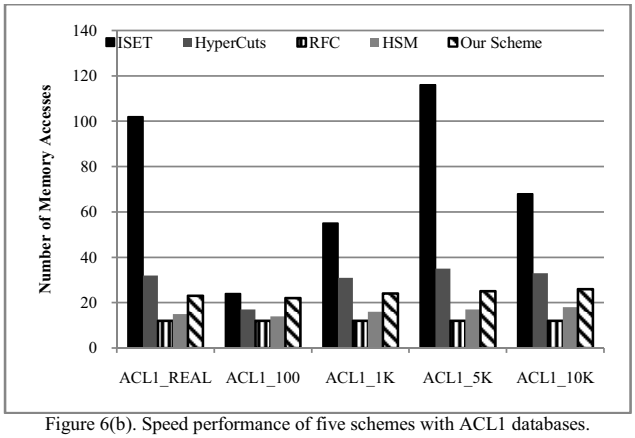

Figure 6(b). Speed performance of five schemes with ACL1 databases.

out of memory. Fig. 6(a), 7(a) and 8(a) show the comparisons of memory requirements with five schemes. Our scheme greatly improves the storage performance of both RFC and HSM since they use similar data structures. The memory requirement of our proposed scheme is larger than that of ISET since ISET heavily relies on linear search in their data significantly worse than our scheme. However, the speed performance of ISET is also significantly worse than our scheme. The results of Hypercuts vary for different databases. While Hypercuts performs well for ACL, its performance degrades severely for FW and IPC databases. These databases result in heavy filter replication structures. As a result, both storage and speed performance is worsened simultaneously. For FW and IPC databases, our scheme outperforms Hypercuts for both speed and storage performance. In summary, our scheme shows the best feasibility among these schemes. Although it may not be the best scheme for a single performance metric, it always provides consistent throughput and avoids the worst case performance.

V. CONCLUSIONS

Packet classification is an important technique for the future Internet. In this paper, we proposed an effective algorithm based on RFC. RFC can classifies packets within few memory accesses. However, the main drawback of RFC is that it may

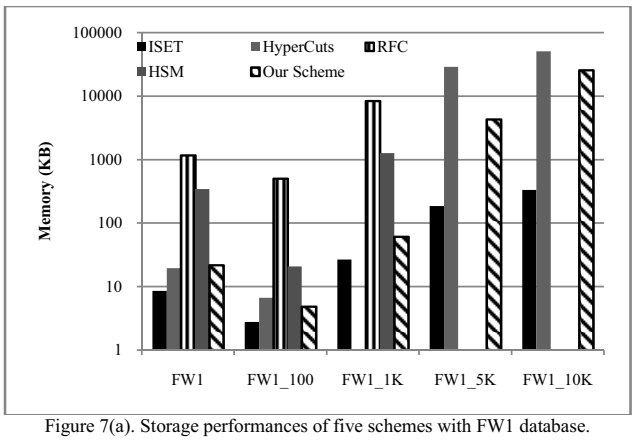

Figure 7(a). Storage performances of five schemes with FW1 database.

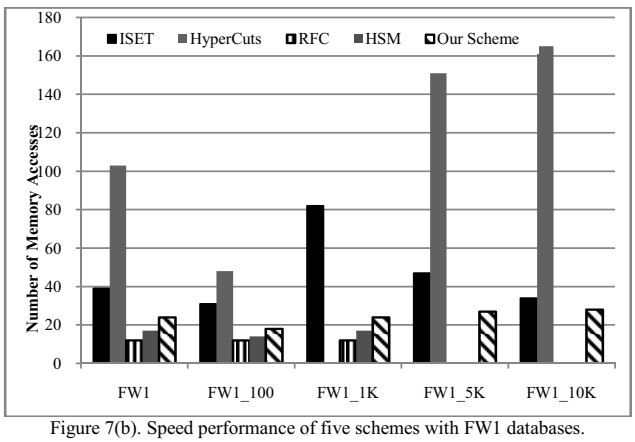

Figure 7(b). Speed performance of five schemes with FW1 databases.

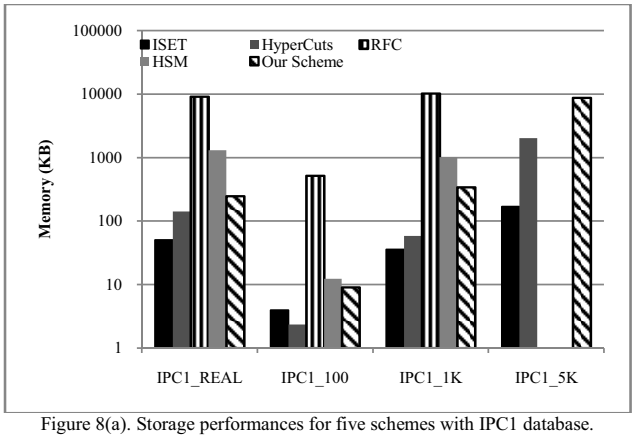

Figure 8(a). Storage performances for five schemes with IPC1 database.

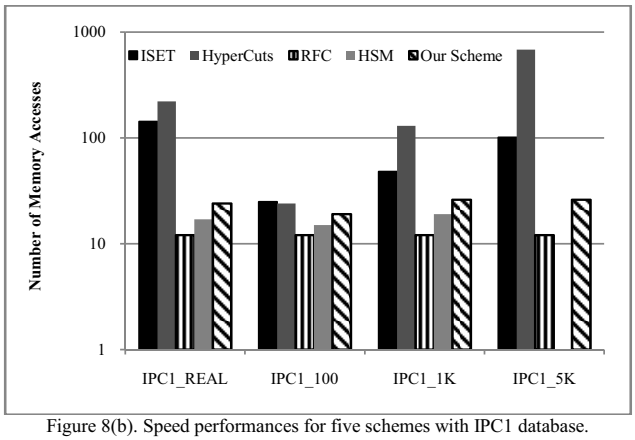

Figure 8(b). Speed performances for five schemes with IPC1 database.

incur high memory consumption in generating the cross-product tables. Owing to this drawback, RFC is not feasible for large databases. To improve the storage performance, we design a algorithm to partition the rule database into several subsets geometrically. Each rule in a subset is stored in an RFC data structure and each subset is represented by an index rule. All index rules are stored in an index RFC for pointing to the corresponding RFC data structure. By traversing these RFC data structures recursively, the highest priority matching rule for an incoming packet can be yielded. We further merge the index arrays of the same chunk in the first phase to reduce memory accesses. We also transform the index arrays for binary searches to improve the storage performance. We use three types of rule sets to evaluate the performance, including access control list, firewall and IP chains. The results show that our scheme can significantly reduce the memory requirement as compared with RFC. It also leverages the performance of storage and speed to avoid extreme cases of the existing schemes.

REFERENCES

[1] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," SIGCOMM Comput. Commun. Rev., vol. 29, no. 4, pp. 135-146, 1999.

[2] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," SIGCOMM Comput. Commun. Rev., vol. 29, no. 4, pp. 147-160, 1999.

[3] X. Bo, J. Dongyi, and L. Jun, "HSM: A Fast Packet Classification Algorithm," in Proceedings of AINA 2005, pp. 987-992, 2005.

[4] M. Zhang and G. Li, " Research on Packet Classification based on improved cross-product method," Procedia Engineering, vol. 24, no. 1, pp. 232-236, 2011.

[5] L. Choi, H. Kim, S. Kim, and M. H. Kim, "Scalable Packet Classification Through Rulebase Partitioning Using the Maximum Entropy Hashing," IEEE/ACM Trans. Netw., vol. 17, no. 6, pp. 1926-1935, 2009.

[6] P. Fong and T. Nian-Feng, "HaRP: Rapid Packet Classification via Hashing Round-Down Prefixes," IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 7, pp. 1105-1119, 2011.

[7] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in Proceedings of Hot Interconnects VII, pp. 34-41, 1999.

[8] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in Proceedings of ACM SIGCOMM, pp. 213-224, 2003.

[9] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: Optimizing Packet Classification for Memory and Throughput," SIGCOMM Comput. Commun. Rev., vol. 41, no. 4, pp. 207-218, 2010.

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

[11] Y.-K. Chang, I. L. Chun, and S. Cheng-Chien, "Multi-field Range Encoding for Packet Classification in TCAM," in Proceedings of IEEE INFOCOM, pp. 196-200, 2011.

[12] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-Based Classification Using Gray Coding," IEEE Transactions on Computers, vol. 61, no. 1, pp. 18-30, 2012.

[13] X. Sun, S. K. Sahni, and Y. Q. Zhao, "Packet Classification Consuming Small Amount of Memory," IEEE/ACM Trans. Netw., vol. 13, no. 5, pp. 1135-1145, 2005.

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