High Performance and High Scalable Packet Classification Algorithm for Network Security Systems

Wooguil Pak, *Member, IEEE* and Young-June Choi, *Member, IEEE*

Abstract—Packet classification is a core function in network and security systems; hence, hardware-based solutions, such as packet classification accelerator chips or Ternary Content Addressable Memory (T-CAM), have been widely adopted for high-performance systems. With the rapid improvement of general hardware architectures and growing popularity of multi-core multi-threaded processors, software-based packet classification algorithms are attracting considerable attention, owing to their high flexibility in satisfying various industrial requirements for security and network systems. For high classification speed, these algorithms internally use large tables, whose size increases exponentially with the ruleset size; consequently, they cannot be used with a large rulesets. To overcome this problem, we propose a new software-based packet classification algorithm that simultaneously supports high scalability and fast classification performance by merging partition decision trees in a search table. While most partitioning-based packet classification algorithms show good scalability at the cost of low classification speed, our algorithm shows very high classification speed, irrespective of the number of rules, with small tables and short table building time. Our test results confirm that the proposed algorithm enables network and security systems to support heavy traffic in the most effective manner.

Index Terms—Packet classification, partitioning, cache-aware table structure, integrated inter- and intra-table search

1 Introduction

Multi-field based packet classification algorithms are widely used not only for policy-based routing, NAT (network address translation), and load balancing in network systems, but also for firewall policy, SPD (security policy database) of VPN (virtual private network), and access control in network security systems [1]. The objective of multi-field-based packet classification is to select the highest priority policy that matches all fields with a given packet header among a ruleset whose rules consist of multiple fields; the packet is then processed according to the policy. The complexity of algorithms increases with the number of fields and the size of rulesets; thus, it is challenging to maintain high classification speed regardless of the size of rulesets [3], [5], [11], [16]. In recent times, the size of networks has significantly increased and there has been a corresponding exponential increase in network traffic, so packet classification becomes bottlenecks in the performance of network systems. This problem is further aggravated by more stringent requirements for security and high-level services. Such services require multiple packet classifications to process one packet, which drastically degrades the network performance [16].

Hardware-based solutions such as packet classification accelerator chips or Ternary Content Addressable Memory (T-CAM) have been widely adopted since they support wire-speed classification performance [10] but they have a difficulty to satisfy various industrial requirements [1]. For example, most hardware-based classifications adopt FPGA/ASIC based algorithm. Developing such a dedicated hardware chip requires a high cost as well as a long time, so it is impossible to modify and extend the chips to support new functions whenever customers request them. Therefore, we require a new solution to achieve high classification performance and high flexibility, simultaneously.

To overcome this problem, we need to develop a software-based packet classification algorithm that supports fast classification as well as a large ruleset size. For application to various environments, the algorithm should also support small table sizes and high-speed table updates. Generally, packet classification algorithms use large and complex internal tables to maximize classification performance, and the size of the tables increases exponentially with the size of the rulesets [9], [11], [19]. When a table is created for a ruleset with tens of thousands of rules, the size of the classification tables is of the order of several gigabytes, making it infeasible for most network and security platforms. One of the best solutions is to divide the entire ruleset into small sub-rulesets by partitioning, and to apply a packet classification algorithm to each sub-ruleset [3], [12], [20]. Small sub-ruleset size is advantageous for achieving high classification speed as well as small table size. However, it does not always guarantee the highest classification speed because the overhead of inter-partition search[1] increases with the ruleset size [3]. Thus, non-partitioning algorithms, such as the cross-producting algorithm, facilitate high-speed

- *W. Pak is with the Computer Engineering Department, Keimyung University, Daegu, Korea. E-mail: wooguilpak@kmu.ac.kr.*
- *Y.-J. Choi is with the School of Information and Computer Engineering, Ajou University, Suwon, Korea. E-mail: choiyj@ajou.ac.kr.*

Manuscript received 7 Oct. 2013; revised 3 Apr. 2015; accepted 4 Apr. 2015. Date of publication 10 June 2015; date of current version 18 Jan. 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2015.2443773

1. Finding the partition which can contain the matching rule.

classification unless large table sizes and long building times are considered [11], [15], [16].

Finally, there is no packet classification algorithm for simultaneously supporting a large ruleset and fast classification. To solve this problem, we propose a new partitioning-based algorithm. The features of our algorithm are summarized below.

- It maintains constant high performance of packet classification regardless of ruleset size.
- It supports large ruleset size that is almost impossible for existing fast packet classification algorithms.
- It eliminates the inter-partition search overhead, which is a critical weakness of partitioning-based algorithms.
- It adopts a new partitioning technique that minimizes redundant rules and supports fast partitioning.

The proposed algorithm involves a new approach in supporting large rulesets while maintaining packet classification performance by integrating *partition search tables* and *packet classification tables*. To realize this concept, we require a new partitioning algorithm that generates integrated tables and partitions efficiently, i.e., the number of duplicated rules included in multiple partitions should be minimized. For achieving higher efficiency, partitioning must be performed using as many fields as possible; however, a large number of fields results in high inter-partition search overhead since the overhead is usually proportional to the total field number used for partitioning [3]. In the proposed algorithm, the overhead is completely eliminated, and therefore, all fields are utilized for partitioning to achieve the best performance.

For the inter-partition search, the proposed algorithm adopts a non-partitioning approach, i.e., a cross-producting algorithm. Our concept of integrating partition search tables and packet classification tables can be easily implemented using cross-producting algorithms. Cross-producting algorithms show the highest classification speed, but they are not widely used because of their poor scalability in terms of the ruleset size, table size, and table building time [4], [11], [19]. In contrast, our algorithm provides very high scalability while maintaining the classification speed. Moreover, the performance can be easily improved by using caching and multi-threads.

The remainder of this paper is organized as follows. In Section 2, we review related studies, and in Section 3, we describe our algorithm; in particular; we explain how the integrated table is created by combining inter- and intra-partition search tables. Next, we evaluate our algorithm by comparing its performance with that of existing algorithms, and we present the results in Section 4. Finally, we conclude the paper in Section 5.

2 Related Work

Packet classification algorithms are classified according to their characteristics or implementation types. In this paper, we classify algorithms into non-partitioning and partitioning types according to the adopted partitioning[2] techniques.

2. *Cut* and *partition* are interchangeable; in this paper, we use only *partition* for consistency.

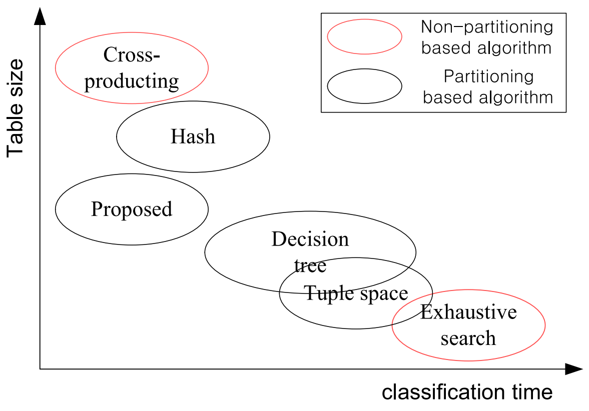


Fig. 1. Performance comparison with respect to classification time and table size of each category of packet classification algorithms.

Fig. 1 shows the performance comparison of well-known non-partitioning and partitioning algorithms [1], [4]. As shown in the figure, partitioning-based algorithms simultaneously fulfill two requirements: reasonable table size and high-speed packet classification. However, partitioning-based algorithms are limited in the maximum packet classification speed they can achieve, while the cross-producting algorithm (one of the most widely used non-partitioning based algorithms) gives a very fast performance but requires a very large amount of memory and very long table construction time.

Now, we will briefly introduce some algorithms belonging to each category.

2.1 Non-Partitioning-Based Approach

This approach finds matching rules for the given keys by searching the entire ruleset. To achieve a high classification speed, it uses very large and complicated tables; hence, it suffers from large table size as the size of the ruleset increases. Therefore, it is not suitable for large rulesets. Exhaustive search [5], cross-producting [9], [11], [19], and caching-based algorithms [21] are well-known algorithms belonging to this category.

2.1.1 *Exhaustive Searching*

A well-known exhaustive search algorithm is the linear search algorithm, which linearly searches all rules that are arranged in the decreasing order of priority [5]. Therefore, it shows poor classification speed, i.e., O(N), where N is the total number of rules; on the other hand, the rule update speed and memory requirement are O(1) and O(N), which are fairly good compared to other algorithms. Owing to these characteristics, this algorithm is preferred for small rulesets. For large rulesets, it should be used together with partitioning-based algorithms, such as decision tree or hashing-based algorithms, for intra-partition search [3], [12].

2.1.2 *Cross-Producting*

Cross-producting algorithms find a matching rule by merging search results field by field using pre-built tables [9], [19]. In general, it achieves the highest performance for packet classification, at the cost of very large tables. For example, RFC, one of the most well-known cross-producting algorithms, shows a fixed classification speed regardless of the ruleset size [11]. However, it requires very large table size and a very long building time. The table size increases exponentially with the ruleset size; thus, this algorithm cannot be applied for large rulesets. Moreover, it is impossible to support partial table update; hence, it has a very long table update time.

To overcome these problems, tables can be built with partitioning techniques. For example, FRFC generates tables in a partitioning manner but searches in a non-partitioning manner [16]. It divides the ruleset into small partitions and builds RFC tables for each partition; these tables are then merged. Consequently, FRFC improves the table building speed while maintaining high classification speed. For each partition, the table building time decreases exponentially, and the total time also decreases significantly, as compared to RFC. However, FRFC is not a fundamental solution because it cannot improve the low scalability caused by large table size.

2.1.3 Caching-Based Algorithms

Caching-based algorithms save pre-searched results associated with keys into the cache [21]. When searching with a key, the result is quickly found via an exact matching based on a hash function. This approach is simple and very effective, especially when the same key is frequently used for searches. However, it is known that the locality of keys decreases drastically in a large-scale network. For such a network, the cache hit ratio is low, the cache update overhead is large, and the cache size increases rapidly; hence, the overall packet classification performance is degraded drastically. The caching-based approach is suitable only for small networks [23].

2.2 Partitioning-Based Approach

Partitioning-based approaches perform packet classification efficiently by reducing the search space for the given keys through partitioning. It is very hard to develop an optimal partitioning algorithm. For example, the well-known two-means clustering problem which is equal to the bipartition problem turned out to be *NP-hard* [26], [27]. Thus, most of current partitioning algorithms are based on heuristic approaches, so they cannot guarantee optimal results. Well-known partitioning-based algorithms include decision-tree, tuple-space and hash-based algorithms [6], [12], [13].

2.2.1 Decision Tree

Decision trees reduce the ruleset size to be searched by using tree-based data structures [2], [3], [13]. A large ruleset is partitioned into multiple sub-rulesets, each adopting a linear search algorithm for intra-partition search. Decision-tree algorithms have many variants such as basic radix trees, hierarchical trees, multi-field search trees, and modified trees with smaller table size [7], [14], [17], [18]. Most algorithms show moderate performance in terms of the classification speed and table size. In particular, HyperCuts shows very high search speed but the speed decreases as the ruleset size increases. Moreover, the table size increases exponentially, and hence, it may be impossible to support large rulesets.

2.2.2 Tuple-Space-Based Algorithms

Tuple-space-based algorithms partition rulesets according to tuples, where a tuple consists of bit indices for multiple fields of a rule [6], [22]. To find a matching rule, the algorithm finds corresponding tuples for the given keys. Previous studies have shown that a tuple space is smaller than a ruleset size; hence, searching tuples is faster than searching the ruleset. Each rule belonging to a tuple has the same bit-mask length for any field of the tuple; therefore, adopting a hash algorithm facilitates faster classification for a ruleset in the tuple.

2.2.3 Hash-Based Algorithms

Multiple memory accesses are required to find a partition including the matching rule using decision-tree or tuple-space-based algorithms; hence, these algorithms provide limited support for fast inter-partition search. This problem is resolved by a hash-based algorithm, which creates a hash key from all or some selected keys for the corresponding fields; thus, a partition is found with one or two memory accesses [12]. Although this algorithm almost completely eliminates the inter-partition search overhead, the total number of partitions is significantly increased, resulting in a very large table size. Therefore, hash-based algorithms are preferable when high packet classification performance must be achieved regardless of memory size.

3 Algorithm Description

3.1 Motivation

As the requirements of various functions in security and network systems increase, packet classification algorithms need to support larger rulesets than ever before, while maintaining high classification speed. Non-partitioning algorithms provide limited support for large rulesets and fast updates, whereas partitioning algorithms provide limited support for fast classifications. To satisfy all the requirements, we need to design a new packet classification algorithm that supports fast classification, fast table update, large ruleset size, and a small table size, simultaneously. In this paper, we propose a new partitioning-based algorithm for packet classification. To find a matching rule, we require inter-partition search followed by intra-partition search, which is nearly similar to previous partition-based algorithms. However, data for intra-partition search are cached during inter-partition search. Accordingly, the total number of slow DRAM (dynamic random access memory) accesses for intra-partition search decreases, and our algorithm achieves almost the same performance as fast non-partitioning algorithms such as cross-producting algorithms. Before further explanation about our idea, let us define some words as following:

- *Field*. Layer 3 and 4 headers of a packet consists of many fields such as IP addresses, ports and length. Packet classification uses some of them to decide how to process the packet. In this paper, we limitedly use the term 'field' to refer to header fields which are used by packet classification. For example, source IP address is one of such fields. Generally, we assume that packet classifiers are based on five

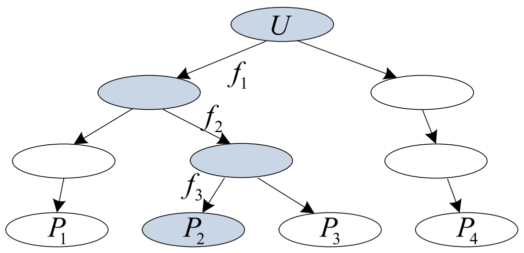


Fig. 2. Example of partition decision tree with three fields (F_1, F_2, F_3), when partitioning is performed in the order of F_1, F_2 and F_3 and ruleset U is divided into four partitions, P_1, P_2, P_3, P_4.

fields: source IP, destination IP, source port, destination port, protocol.

- *Key*. An input value corresponding to each field is called 'key'. For example, 10.1.1.1 is a key for source IP address.
- *Rule*. The packet classifier is a collection of pairs of a condition and corresponding action. Each of the pair is called 'rule'. The condition consists of sub-conditions for each field. For example, a rule can be defined as (SIP: 10.0.0.0 - 20.0.0.0, DIP: ANY, SPORT: ANY DPORT: 80, Protocol: TCP, Action: allow)
- *Entry*. Packet classification requires table structures to search a matching rule efficiently. Each table consists of multiple elements which are called 'entry'.

To explain our key concept, we assume that there is a ruleset U where each rule consists of three fields (F_1, F_2, F_3). Fig. 2 shows a partition decision tree that divides the ruleset U into four sub-partitions, P_1, P_2, P_3 and P_4, by partitioning in the order of fields F_1, F_2 and F_3. Each leaf node corresponds to a sub-partition, and the other nodes correspond to intermediate partitions that are divided into partitions for child nodes. For example, when traversing the tree according to keys (f_1, f_2, f_3), we visit the colored nodes in the tree. Thus, the final matching partition for the given keys is P_2, as shown in Fig. 2. When we find a matching rule for the keys (f_1, f_2, f_3), we look up only rules belonging to partition P_2.

Suppose RFC is adopted for intra-partition search, for simplicity. We briefly explain how RFC works. It maps a k_i-bit key, f_i onto T_i^1-bit action identifier where $T_i^1 \ll k_i$. This process requires only one memory access to the pre-processed table called a chunk table. Some T_i^1 values are combined into one value which is mapped to T_j^2 by using a next phase chunk table. This procedure is repeated until only one chunk table which addresses to the matched entry remains.

Although it cannot support large rulesets due to large sizes of chunk tables, it only requires a fixed and small number of memory accesses to find a matching rule regardless of the key value and ruleset size, so it can achieve a very high classification performance.

Fig. 3 shows the RFC tables and its search process for partition P_2 obtained from Fig. 2. Here, c_k^{p,P_h} denotes the kth chunk table in phase p for the hth partition. In Fig. 3, the matching rule index is shown to be 5 for keys (f_1, f_2, f_3).

The search mechanisms in the partition decision tree in Fig. 2 and in the RFC table in Fig. 3 are different for the following reasons. First, inter-partition search in Fig. 2 is

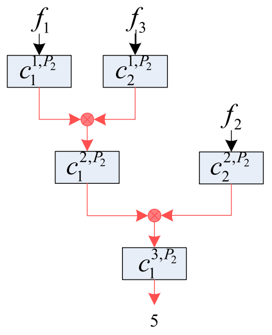


Fig. 3. Example of searching a rule in RFC tables for partition P_2 from Fig. 2. Each box represents an RFC chunk table, and the final rule index found is 5 for keys (f_1, f_2, f_3) .

performed in the order of F_1, F_2, F_3, whereas intra-partition search in Fig. 3 is performed in the order of F_1, F_3, F_2. When we construct a partition decision tree as shown in Fig. 2, we first select a partitioning field having a smaller number of duplicated rules. In contrast, to build RFC tables as shown in Fig. 3, we combine the field of a large CBM (class bitmap) table size with that of a small CBM size, because it is helpful to avoid a large table size in the next phase. Therefore, the orders of searching fields in the tables are different for inter- and intra-partition search. Another difference is that only RFC tables need additional accesses to tables $(c_1^{2,P_2}, c_1^{3,P_2})$.

However, both search mechanisms described above are similar in that every entry is read for the keys (f_1, f_2, f_3) for all fields F_1, F_2, and F_3. Assume that, for partition P_2, each entry of key f_i in both the tables is located in the same cache line. Then, when we search RFC tables after traversing a decision tree, we obtain all the entries of RFC tables for keys (f_1, f_2, f_3) without accessing the tables in DRAM, because each entry has already been loaded into the cache during the access of the decision tree. Therefore, the total number of memory accesses to find a matching rule becomes the same as the number for searching RFC tables only.

In other words, the cost of accessing slow DRAM for intra-partition search is already included in the cost for intra-partition search. In addition, the table size, table building time, and table updating time are reduced significantly, owing to the characteristics of the partitioning-based algorithm. Thus, we can simultaneously improve the performance and scalability of the packet classification algorithm.

Against cache oblivious algorithms, our approach requires prior information such as the cache-line size [25]. However, it is not a serious restriction since most of platforms have a cache-line size, 64 bytes. It can also support any cache line size if the size is known before table building. Since this approach also does not consider any specific cache hierarchy, it can be adopted by most of existing platforms.

To realize this idea, we need to develop a partitioning algorithm for achieving high partitioning efficiency and short partitioning time through rule search tables integrated with a partition decision tree. First, we will explain a partitioning algorithm, and then, discuss the structure of integrated tables. Finally, we will explain how to further improve the performance of the algorithm.

3.2 Recursive Binary Ruleset Partitioning Algorithm

In packet classification algorithms, partitioning is an effective technique to reduce the number of rules to be searched, such that searching time and table size significantly decrease. However, most of partitioning algorithms do not guarantee the optimal performance although they have high computational complexity. For example, DIANA, the well-known hierarchical partitioning algorithm internally uses a dissimilarity matrix which consists of the distance of every pair of data elements; therefore, it requires at least $O(n^2)$ to build the matrix where n is the total data size but fails to guarantee the optimal partitioning result [24].

Our partitioning algorithm is similar to DIANA but it does not need a dissimilarity matrix and takes only $O(n)$ to divide a ruleset into two sub-rulesets while guaranting optimality. The algorithm fully utilizes the characteristics of the ruleset for packet classification to achieve high performance. At first, we introduce the optimal binary ruleset partitioning algorithm (BRP) and then show a general partitioning algorithm developed based on the BRP.

3.2.1 Optimal Binary Ruleset Partitioning

For making explanation simple, we define some notations as follows.

- $U = \{R_1, R_2, \ldots, R_n\}$: a firewall ruleset with n rules.
- $|\cdot|$: the cardinality of the set.
- d : the total number of fields for each rule.
- $R_m = \{R_1^m, R_2^m, \ldots, R_d^m\}$: the mth rule of U, $\forall m \in \{1, 2, \ldots, n\}$.
- K_j : a set of all possible key values for the jth field, $\forall j \in \{1, 2, \ldots, d\}$.
- $R_j^m = \{r | R_j^{m,\min} \leqslant r \leqslant R_j^{m,\max}\}$: the matching condition for the jth field of R_m, where $R_j^m \subseteq K_j$. Each field is defined by a range of matching key values.
- $M_j^i = \{m | \forall m, \text{ s.t. } i \in R_j^m\}$: the set of matching rules with key $i (\in K_j)$ for the jth field.
- $M_j = \{M_j^i | \forall i \in K_j\}$.
- $M_j(h)$: the $h(> 0)$th element of M_j.
- $\eta(M_j(h)) = \min\{i | M_j^i = M_j(h), \forall i\}$: the smallest value among keys corresponding to M_j^i.
- $M_j^S = \bigcup_{\forall h \in S} M_j(h)$.
- $M_j^{a\sim b} = \bigcup_{h=a}^{b} M_j(h)$, where $M_j^{a\sim a} = M_j(a)$.
- $A_j^k = M_j(k+1) - M_j(k), \quad \forall k \in \{1, 2, \ldots, |M_j| - 1\}$, where $A_j^0 = M_j(1)$.
- $B_j^k = M_j(k) - M_j(k+1), \quad \forall k \in \{1, 2, \ldots, |M_j| - 1\}$, where $B_j^0 = \varnothing$.

We assume that the elements of M_j are ordered such that $\eta(M_j(h)) < \eta(M_j(h+1))$ for any $h > 0$.

Proposition 1. *$\{A_j^k | \forall k\}$ is pairwise disjoint.*

Proof. Let $\Gamma(M_j(k))$ be the set of all matching key values to $M_j(k)$. According to the definition of $M_j(k)$, each key value corresponds to only one $M_j(k)$. Therefore, $\{\Gamma(M_j(k)) | \forall k\}$ is pairwise disjoint. If we assume that $\{A_j^k | \forall k\}$ is not pairwise disjoint for contradiction, there is at least one element, $i \in A_j^h \cap A_j^k (\neq \varnothing)$. We conclude that $r_j^{i,\min} \in \Gamma(M_j(h+1))$ and $r_j^{i,\min} \in \Gamma(M_j(k+1))$ since $i \in A_j^k \Rightarrow r_j^{i,\min} \in \Gamma(M_j(k+1))$. However, it is contradiction of the fact that $\{\Gamma(M_j(k)) | \forall k\}$ is pairwise disjoint. Thus, $\{A_j^k | \forall k\}$ is pairwise disjoint. □

Now, we consider the partitioning problem which divides $\Omega = \{k | k \in \mathrm{Z}^+, 1 \leqslant a \leqslant k \leqslant b \leqslant |M_j|\}$ into two sub-sets $P (\subsetneq \Omega)$ and $\Omega - P$ while minimizing the number of rules belonging to both of M_j^P and $M_j^{\Omega - P}$ as follows:

P1 :

$$\underset{P \subsetneq \Omega}{\text{minimize}} \, |M_j^P \cap M_j^{\Omega - P}|$$
$$\text{where } \Omega = \{k | k \in \mathrm{Z}^+, 1 \leqslant a \leqslant k \leqslant b \leqslant |M_j|\}.$$

Assuming that $P^* (\subsetneq \Omega)$ is the solution to **P1**, we let $i (a \leqslant i < b)$ be $\max\{k | k \in P^*\}$. Then we can establish Proposition 2.

Proposition 2. *$A_j^t \cap M_j^R = \varnothing, \forall t \geqslant i$, when $\forall R \subsetneq P^*$ and $M_j(i) \notin R (\neq \varnothing)$.*

Proof. According to Proposition 1, $A_j^t \cap M_j(k) = \varnothing, \forall t > k$ $\because M_j(k) = \bigcup_{u=0}^{k-1} (A_j^u - B_j^u)$. Since $M_j(k) \in R \Rightarrow k < i - 1$, we see that $A_j^t \cap M_j^R = \varnothing, \forall t \geqslant i > k - 1$. $\therefore A_j^t \cap M_j^R = \varnothing, \forall t \geqslant i$. □

Then we obtain Lemma 1 as follows.

Lemma 1. *The solution for **P1** is defined as $P^* = \{k | a \leqslant k \leqslant i < b\}$, where $i = \max\{k | k \in P^*\}$.*

Proof. Let us assume that the solution **P1** is not P^* but $P^* - R$, where $R \subsetneq P^*$ and $M_j(i) \notin R (\neq \varnothing)$. Then,

$$\begin{aligned}
&|M_j^{P^* - R} \cap M_j^{(\Omega - P^*) \cup R}| \\
&= |M_j^{P^* - R} \cap (M_j^{\Omega - P^*} \cup M_j^R)| \\
&= |(M_j^{P^* - R} \cap M_j^{\Omega - P^*}) \cup (M_j^{P^* - R} \cap M_j^R)|,
\end{aligned}$$

and

$$\begin{aligned}
&|M_j^{P^*} \cap M_j^{\Omega - P^*}| \\
&= |M_j^{(P^* - R) \cup R} \cap M_j^{\Omega - P^*}| \\
&= |(M_j^{P^* - R} \cap M_j^{\Omega - P^*}) \cup (M_j^{\Omega - P^*} \cap M_j^R)|.
\end{aligned}$$

We know that

$$|M_j^{P^* - R} \cap M_j^R| \geqslant |M_j^{\Omega - P^*} \cap M_j^R|,$$

since

$$\begin{aligned}
&|M_j^{P^* - R} \cap M_j^R| \geqslant |M_j(i) \cap M_j^R| \because M_j(i) \subset M_j^{P^* - R} \\
&\geqslant |(M_j(i) - B_j^i) \cap M_j^R| \\
&= |(M_j(i) \cup A_j^i - B_j^i) \cap M_j^R| \\
&\because A_j^i \cap M_j^R = \varnothing \text{ from Proposition 2} \\
&= |M_j(i+1) \cap M_j^R| \\
&= |(M_j(i+1) \cup A_j^{i+1} \cup A_j^{i+2} \cup \cdots \cup A_j^{b-1} \cup A_j^b) \cap M_j^R| \\
&\because A_j^k \cap M_j^R = \varnothing, i < k \leqslant b \text{ from Proposition 2} \\
&= |M_j^{\Omega - P^*} \cap M_j^R|.
\end{aligned}$$

Therefore, we conclude that

$$\left|M_j^{P^*-R} \cap M_j^{(\Omega-P^*)\cup R}\right| \geqslant \left|M_j^{P^*} \cap M_j^{\Omega-P^*}\right|.$$

It is contradiction to the assumption that $P^* - R$ is the solution, so P^* is the solution of **P1**. □

Finally, the solution P^* of **P1** is defined as

$$P^* = \{k | a \leqslant k \leqslant i^* < b\}, \tag{1}$$

where $i^* = \arg\min_i |M_j^{a\sim i} \cap M_j^{i+1\sim b}|$.

The total partition size is $N = |M_j|$ when $\Omega = \{k | 1 \leqslant k \leqslant |M_j|\}$. According to (1), the computational complexity to obtain P^* is $O(N)$ since $P^* = \{k | 1 \leqslant k \leqslant i^* < N\}$. This complexity is much lower than the complexity only to build a dissimilarity matrix, $O(N^2)$ for existing partitioning algorithms such as DIANA.

If M_j^i s are independent of each other like the general assumption for other partitioning problems, we cannot also solve the problem with low time complexity. However, M_j^i s of the packet classification rulesets are dependent, so we could exploit this characteristic to develop an optimal algorithm with $O(n)$.

The building or searching time complexity for packet classification tables may increase after partitioning a ruleset. Thus, partitioning is performed only if the complexity decreases as follows:

$$O_T\left(M_j^{P^*}\right) + O_T\left(M_j^{\Omega-P^*}\right) < O_T\left(M_j^{\Omega}\right), \tag{2}$$

where $O_T(\cdot)$ is defined as the building or searching time complexity of packet classification tables for the given ruleset.

$O_T(\cdot)$ depends on the algorithm for intra partitioning search. Since it requires a long time to build entire tables every time for each partitioning, we should use the approximated complexity. One of simple solutions is to measure building complexity for only some selected sub-tables instead of the entire tables.

For cross-producting algorithms like RFC, it is approximated as follows. Let $b_{j'}^P$ and $b_{j''}^P$ be arbitrary two CBMs for the ruleset P to build the next phase CBM $b_{j'}^P \otimes b_{j''}^P$ where $j \neq j'$ and $j \neq j''$. Also, let $|b_j^P|$ be the total bitmap number of b_j^P. Then, (2) is approximated by $|b_{j'}^{M_j^{P^*}} \otimes b_{j''}^{M_j^{P^*}}| + |b_{j'}^{M_j^{\Omega-P^*}} \otimes b_{j''}^{M_j^{\Omega-P^*}}| < |b_{j'}^{M_j^{\Omega}} \otimes b_{j''}^{M_j^{\Omega}}|$.

In addition to (2), there can be many good criteria to choose partitions for further partitioning such as the minimum partition size. However, we just rely on (2) to make the algorithm simple.

The binary ruleset partitioning algorithm, BRP is repeated for each partition, yielding multiple partitions from the original ruleset U. BRP is outlined as Algorithm 1 below.

3.2.2 Recursive Binary Ruleset Partitioning

To perform partitioning based on multiple fields, we apply BRP according to a predefined order of fields, i.e., $F_1, F_2, F_3, \ldots, F_d$, where d is the total number of fields. Using all fields helps maximize the partitioning performance compared to using some selecting fields, thereby minimizing the table building overhead; moreover, the partitioning procedure is simplified by removing the field selection problem [3]. All fields are used for partitioning without performance degradation because the increased cost of inter-partition search is offset by the decreased cost of intra-partition search, owing to the unique feature of the proposed algorithm. Because BRP is performed recursively in the order of fields, we call this partitioning Recursive BRP (RBRP), which is outlined in Algorithm 2 below.

Algorithm 1. BRP(g, j)

```
1: $Z \leftarrow \varnothing$
2: Build $M_j$ for $g$.
3: while do
4:     Find the largest partition $M_j^{\Omega} (\in M_j)$ where $\Omega = \{k | a \leqslant k \leqslant b\}$.
5:     if not found then
6:         Return $Z$
7:     else
8:         $i^* = \arg\min_i |M_j^{a\sim i} \cap M_j^{i+1\sim b}|, a \leqslant i < b$.
9:         $P^* = \{k | a \leqslant k \leqslant i^*\}$
10:        if $O_T(M_j^{a\sim i^*}) + O_T(M_j^{i^*+1\sim b}) < O_T(M_j^{a\sim b})$ then
11:            $M_j \leftarrow M_j - M_j^{\Omega}$
12:            $M_j \leftarrow M_j \cup M_j^{P^*} \cup M_j^{\Omega-P^*}$
13:        else
14:            $Z \leftarrow Z \cup \{M_j^{\Omega}\}$
15:        end if
16:    end if
17: end while
```

Algorithm 2. RBRP(G, k)

```
1: if $k > d$ then
2:     Return $G$
3: end if
4: $H \leftarrow \varnothing$, $Z \leftarrow \varnothing$
5: for each $g \in G$ do
6:     $H \leftarrow H \cup$ BRP$(g, k)$
7: end for
8: for each $h \in H$ do
9:     $Z \leftarrow Z \cup$ RBRP$(h, k+1)$
10: end for
11: Return $Z$
```

Now, we describe the procedure of the RBRP algorithm with an example in Table 1, where a small ruleset consists of 10 rules. We assume that each rule consists of three fields, and partitioning is performed in the order of DIP_U16, DIP_L16, and DPORT (destination port). We also assume that RFC is used for intra partitioning search. From Table 1, the total ruleset is $U = \{1, 2, 3, \ldots, 10\}$, where each number denotes a rule ID. Thus, we obtain M_1, M_2, M_3 as follows:

$M_1 = \{\{1, 2, 3, 10\}, \{1, 2, 3, 4, 10\}, \{1, 2, 3, 4, 6, 9, 10\}, \{1, 2, 3, 4, 5, 6, 9, 10\}, \{7, 8, 10\}\}$

$M_2 = \{\{1, 2, 3, 10\}, \{4, 6, 9, 10\}, \{4, 5, 6, 9, 10\}, \{5, 6, 7, 9, 10\}, \{7, 8, 10\}\}$

$M_3 = \{\{1, 2, 3, 4, 7, 10\}, \{1, 2, 3, 5, 6, 7, 8, 9, 10\}, \{5, 6, 7, 8, 9, 10\}, \{5, 6, 7, 8, 10\}\}$

TABLE 1
Example of Ruleset U

Rule ID	Field name		
	DIP_U16(F_1)	DIP_L16(F_2)	DPORT(F_3)
1	0.0-100.2	0.0-3.255	0-4095
2	0.0-100.2	0.0-3.255	0-4095
3	0.0-100.2	0.0-3.255	0-4095
4	30.0-100.2	4.0-100.255	0-1023
5	100.1	10.0-199.255	1024-65535
6	50.0-100.2	4.0-199.255	1024-65535
7	100.3-255.255	101.0-255.255	ANY
8	100.3-255.255	200.0-255.255	1024-65535
9	50.0-100.2	4.0-199.255	1024-16383
10	ANY	ANY	ANY

A smaller ID means a higher priority. X.Y is an integer value represented by IP address notation; therefore, it is equal to $256 \cdot X + Y$.

If we apply Algorithm 1 to M_1, i^* is equal to 4 and the table building overhead decreases by 1 since

$$\left|b_2^{M_1^{1\sim4}} \otimes b_3^{M_1^{1\sim4}}\right| + \left|b_2^{M_1^5} \otimes b_3^{M_1^5}\right| = 10$$
$$< \left|b_2^{M_1^{1\sim5}} \otimes b_3^{M_1^{1\sim5}}\right| = 11.$$

Therefore, we have two partitions as follows:

$$\{M_1^{1\sim4}, M_1^5\} \leftarrow \mathrm{BRP}(U, 1).$$

Here, $\leftarrow$ shows the result obtained by applying BRP to a given ruleset. Owing to the increased building time, it is impossible to partition sub-partitions based on $F1$ anymore; hence, we select $F2$ for the next partitioning set, and the partitions are given as follows:

$\{M_1^{1\sim4} \cap M_2^1, M_1^{1\sim4} \cap M_2^{2\sim5}\} \leftarrow \mathrm{BRP}(M_1^{1\sim4}, 2)$,
$\{M_1^5\} \leftarrow \mathrm{BRP}(M_1^5, 2)$.

Finally, we perform partitioning F_3 recursively, and then, we obtain the following:

$\{M_1^{1\sim4} \cap M_2^1\} \leftarrow \mathrm{BRP}(M_1^{1\sim4} \cap M_2^1, 3)$,
$\{M_1^{1\sim4} \cap M_2^{2\sim5} \cap M_3^1, M_1^{1\sim4} \cap M_2^{2\sim5} \cap M_3^{2\sim4}\} \leftarrow \mathrm{BRP}(M_1^{1\sim4} \cap M_2^{2\sim5}, 3)$,
$\{M_1^5, 3\} \leftarrow \mathrm{BRP}(M_1^5, 3)$.

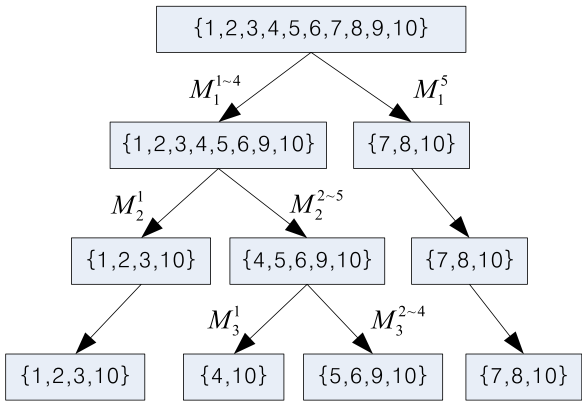


Fig. 4. RBRP partitioning tree for example ruleset of Table 1.

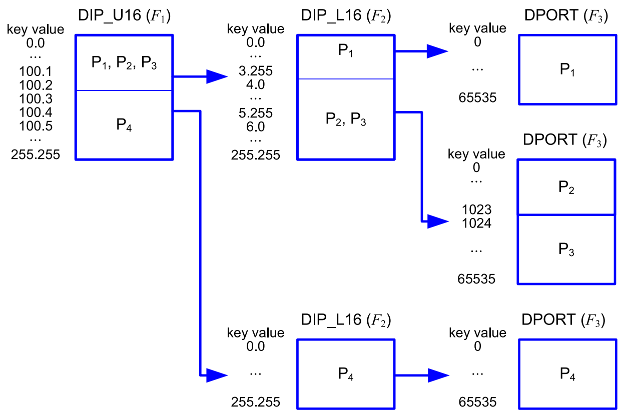


Fig. 5. Partition decision tree built based on BRP tables in Fig. 4.

From partitioning using BRP, we construct an RBRP partitioning tree described in Fig. 4. As shown in Fig. 4, we obtain four partitions as follows:

$$P_1 = M_1^{1\sim4} \cap M_2^1 = \{1, 2, 3, 10\},$$
$$P_2 = M_1^{1\sim4} \cap M_2^{2\sim5} \cap M_3^1 = \{4, 10\},$$
$$P_3 = M_1^{1\sim4} \cap M_2^{2\sim5} \cap M_3^{2\sim4} = \{5, 6, 9, 10\},$$
$$P_4 = M_1^5 = \{7, 8, 10\}.$$

Using Fig. 4, we can also build a partition decision tree that is traversed by key values, as shown in Fig. 5. Each entry in the partitioning decision tree contains one or multiple partition IDs and a pointer to the next table for the corresponding key value. To search the decision tree, we read the entry matched with a key for a field, and find the next entry for the next field repeatedly until we reach a leaf node that contains the final partitioning ID.

3.3 Cache-Oriented Reduction Tree Integrated with Partition Decision Tree

After obtaining the partitions, we build a table of intra-partition search for each partition. During table building, the information of the decision tree is placed in the table according to the cache-line size and memory alignment in order to reduce the cost of memory access. For some selected tables, all entries for the same key values can be merged in order to make one large entry. Now, we call the generated table an integrated table.

Although our algorithm utilizes a cache to improve classification performance, it differs from existing cache based algorithms in that it uses the cache to reduce the number of memory accesses by rearranging data fields required for classifying one packet closer to each other for every single packet classification. This approach needs only $64 \times 13 = 832$ byte cache memory, so it is insensitive to the size of cache or cache hierarchy.

In general, the concept of integrated tables can be applied to other non-partitioning based packet classification algorithms; however, we use the cross-producting algorithm for the following reasons.

- It is one of the fastest algorithms for small and medium size rulesets for the software-based approach but it suffers from poor scalability in terms of the ruleset size and the number of fields. Our partitioning algorithm divides the total ruleset into multiple small sub rulesets. Since the size of the classification table for cross-producting algorithms increases exponentially as the total ruleset size increases, the total table size for all sub rulesets is greatly reduced compared to the table size for the total ruleset. Therefore, the advantage of coupling cross-producting and the proposed algorithm is fully taken.

- Owing to the long table building and updating time, the applications of the cross-producting algorithm are limited. Our proposal can improve the scalability of the cross-producting algorithm significantly in terms of the table size, table building time, updating time and ruleset size; hence, the integrated algorithm is applicable under various environments.

Now, by using the RFC, one of the most well-known cross-producting algorithms, we will explain the procedure for constructing an integrated table for intra-partition search. For simplicity, we define the following notations for $1 \leqslant p \leqslant p_{\max}$.

- c_k^{p,P_h} kth chunk of partition P_h for phase p (refer to Fig. 3 for definition of k.).
- $|c_k^{p,P_h}|$ Total number of entries in c_k^{p,P_h}.
- $c_k^{p,P_h}[i]$ Ruleset corresponding to ith entry of c_k^{p,P_h}.
- b_k^{p,P_h} kth CBM of partition P_h for phase p.
- $b_k^{p,P_h}[i]$ Ruleset corresponding to ith entry of b_k^{p,P_h}.
- $|b_k^{p,P_h}|$ Total number of entries in b_k^{p,P_h}.
- $\phi_k^{p,P_h}(i)$ j s.t. $b_k^{p,h}[j] = c_k^{p,P_h}[i]$ or -1 if $c_k^{p,P_h}[i]$ does not exist.

From the definitions, we define a chunk table, which is used for intra-partition search, by $\{\phi_k^{p,P_h}(0), \phi_k^{p,P_h}(1), \ldots\}$. To construct an integrated table using RFC tables, we require two processes: *packing*, which combines some chunk tables entry by entry into one large table for each field, and *merging*, which merges information of multiple partitions, thus reducing the entry size to maintain it less than or equal to the cache line size.

3.3.1 Packing

Packing is a procedure for storing all entries of RFC tables corresponding to each key into an integrated table entry, thus enabling all the entries to be cached together with one memory access. We define the following variables for describing the integrated table.

- $C_k^{p,l}$ Integrated table built on the basis of c_k^{p,P_h} for $P_h = l, l+1, \ldots$ where l is the smallest ID of corresponding partitions to be combined.
- $|C_k^{p,l}|$ Total number of entries in $C_k^{p,l}$.
- $C_k^{p,l}[i]$ CBM indices of partitions corresponding to ith entry of $C_k^{p,l}$.

If the corresponding partitions for $C_k^{p,a}[i]$ are P_a, P_{a+1}, $\ldots$, P_b, $C_k^{p,a}[i]$ is defined as

$$C_k^{p,a}[i] = \langle \phi_k^{p,P_a}(i), \phi_k^{p,P_{a+1}}(i), \ldots, \phi_k^{p,P_b}(i) \rangle, \quad (3)$$

where $\langle\cdot\rangle$ is an array that consists of the given elements.

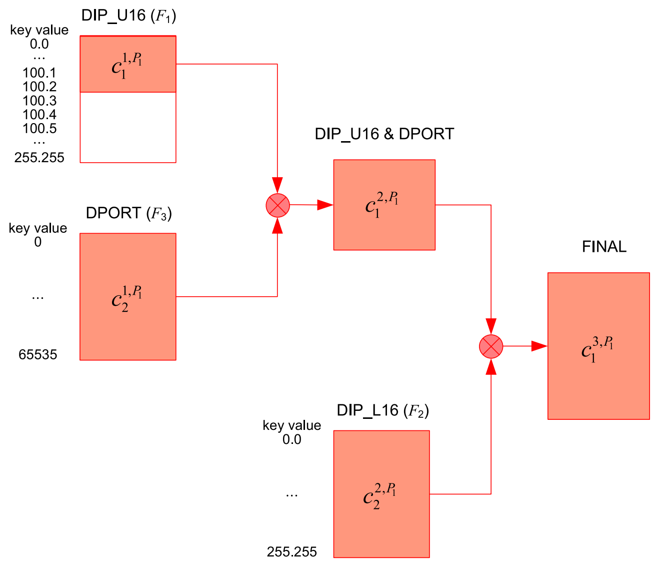


Fig. 6. Example of RFC tables for intra-partition search of partition P_1. The blank box represents unavailable entries for P_1.

In RFC tables, we assume that c_k^{p,P_h} is generated by c_l^{p-1,P_h} and c_{l+1}^{p-1,P_h}, and each entry of $C_l^{p-1,a'}[m]$ and c_{l+1}^{p-1,P_h} is given as follows:

$$C_l^{p-1,a'}[m] = \langle \phi_l^{p-1,P_{a'}}(m), \phi_l^{p-1,P_{a'+1}}(m), \ldots, \phi_l^{p-1,P_{b'}}(m) \rangle,$$
$$C_{l+1}^{p-1,a''}[m] = \langle \phi_{l+1}^{p-1,P_{a''}}(m), \phi_{l+1}^{p-1,P_{a''+1}}(m), \ldots, \phi_{l+1}^{p-1,P_{b''}}(m) \rangle.$$

Then, $a = \max\{a', a''\}, b = \min\{b', b''\}$ in (3). When the structure of RFC tables for **P1** is given as shown in Fig. 6, the entire integrated tables are shown in Fig. 7.

3.3.2 Merging

As shown in Fig. 7, the entry size of the sub-tables varies according to the number of partitions included in the entry. A node that is closer to the root node of a partition decision tree, i.e., smaller k, contains a larger number of partitions. Therefore, each node obtained from F_1, the first field for partitioning, has the largest number of partitions. To maximize the classification performance of the proposed algorithm, each entry of the integrated table should be stored in one cache line, as mentioned above. However, the cache line size is fixed according to platform types, and hence, it may fail to satisfy this constraint for some rulesets; therefore, the number of memory accesses is increased.

To solve this problem, we use *partition merging approach*. After table building for each partition, the partition number in each entry, if required, is readjusted according to the cache line size by merging multiple partitions into one. Although this approach is valid only for some cross-producting algorithms, it is very effective in terms of table size and building time because it handles searching tables generated from each partition instead of a ruleset data. For better performance, two partitions with the lowest cost are selected among partitions in the table entry, i.e., $\langle P_a^*, P_b^* \rangle = \arg\min_{\langle P_a, P_b \rangle}\{|b_k^{p,P_a}| \cdot |b_k^{p,P_b}|\}$ in each merging procedure, and we repeat this procedure n times, thereby decreasing the total number of partitions by n.

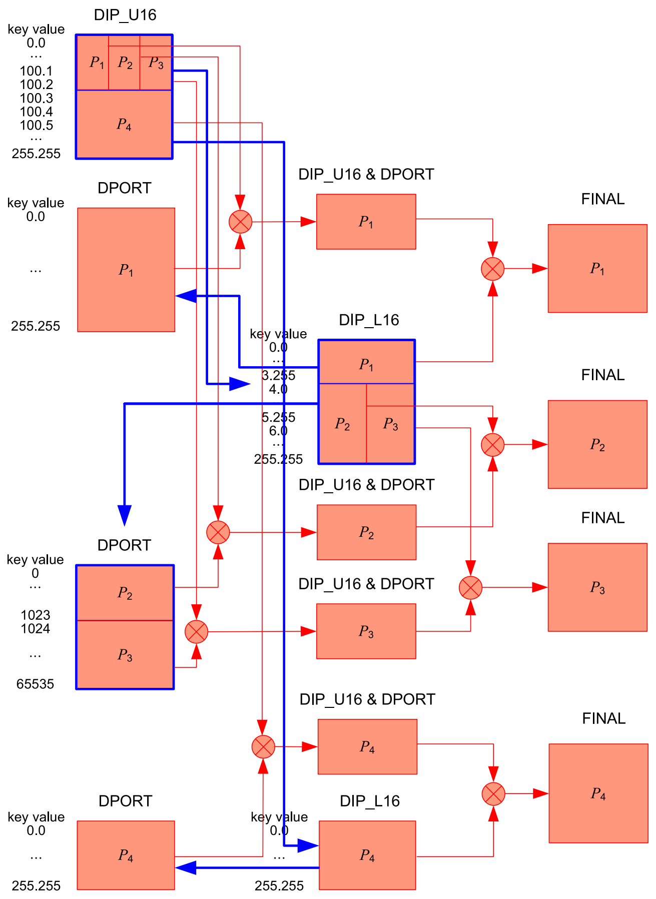


Fig. 7. Integrated table structure for the ruleset in Table 1. The dotted lines denote the order of searching for the decision tree according to the keys. The solid lines denote the inter-connection of RFC tables.

Suppose two partitions P_a and P_b are merged into a single partition represented by $P_a + P_b$ when RFC is used for intra-partition search. Then, we define $c_k^{p,P_a+P_b}[i]$ as

$$c_k^{p,P_a+P_b}[i] = \langle \phi_k^{p,P_a}[i], \phi_k^{p,P_b}[i] \rangle$$

where $\langle A, B\rangle = \langle C, D\rangle \Leftrightarrow A = C$ and $B = D$.

We also define $\phi_k^{p,P_a+P_b}(i)$ as

$$\phi_k^{p,P_a+P_b}(i) = j, \text{ s.t. } b_k^{p,P_a+P_b}[j] = c_k^{p,P_a+P_b}[i].$$

Then, we obtain the following proposition.

Proposition 3. *$\phi_k^{p,P_a+P_b}(i) = \phi_k^{p,P_a\cup P_b}(i)$, where $b_k^{p,P_a+P_b}[j]$ is defined by the jth element of the set $\{c_k^{p,P_a+P_b}[i] \mid \forall i\}$.*

Proof. First, let us show

$$c_k^{p,P_a+P_b}[i] = c_k^{p,P_a+P_b}[j] \Rightarrow c_k^{p,P_a\cup P_b}[i] = c_k^{p,P_a\cup P_b}[j].$$

$$\begin{aligned}
&c_k^{p,P_a+P_b}[i] = c_k^{p,P_a+P_b}[j]\\
&\Rightarrow \langle \phi_k^{p,P_a}(i), \phi_k^{p,P_b}(i)\rangle = \langle \phi_k^{p,P_a}(j), \phi_k^{p,P_b}(j)\rangle\\
&\Rightarrow \phi_k^{p,P_a}(i) = \phi_k^{p,P_a}(j) \text{ and } \phi_k^{p,P_b}(i) = \phi_k^{p,P_b}(j)\\
&\Rightarrow c_k^{p,P_a}[i] = c_k^{p,P_a}[j] \text{ and } c_k^{p,P_b}[i] = c_k^{p,P_b}[j]\\
&\Rightarrow c_k^{p,P_a}[i] \cup c_k^{p,P_b}[i] = c_k^{p,P_a}[j] \cup c_k^{p,P_b}[j]\\
&\Rightarrow c_k^{p,P_a\cup P_b}[i] = c_k^{p,P_a\cup P_b}[j].
\end{aligned}$$

Now, we will show

$$c_k^{p,P_a\cup P_b}[i] = c_k^{p,P_a\cup P_b}[j] \Leftarrow c_k^{p,P_a\cup P_b}[i] = c_k^{p,P_a\cup P_b}[j].$$

We prove that

$$\begin{aligned}
&c_k^{p,P_a}[i] \cup c_k^{p,P_b}[i] = c_k^{p,P_a}[j] \cup c_k^{p,P_b}[j]\\
&\Rightarrow c_k^{p,P_a}[i] = c_k^{p,P_a}[j] \text{ and } c_k^{p,P_b}[i] = c_k^{p,P_b}[j]\\
&\text{if } P_a \cap P_b = \varnothing.
\end{aligned}$$

By definition,

$$c_k^{p,P_a}[\cdot] \subset P_a, \text{ and } c_k^{p,P_b}[\cdot] \subset P_b$$

$$\therefore\ c_k^{p,P_a}[m] \cap c_k^{p,P_b}[n] = \emptyset, \forall m, n \because P_a \cap P_b = \emptyset$$

Thus,

$$\begin{aligned}
&c_k^{p,P_a}[i] \cup c_k^{p,P_b}[i] = c_k^{p,P_a}[j] \cup c_k^{p,P_b}[j]\\
&\Leftrightarrow c_k^{p,P_a}[i] = c_k^{p,P_a}[j] \text{ and } c_k^{p,P_b}[i] = c_k^{p,P_b}[j]
\end{aligned}$$

If $\overline{P_a} = P_a - P_a \cap P_b$ and $\overline{P_b} = P_b - P_a \cap P_b$, then, $\overline{P_a}$, $\overline{P_b}$, and $P_a \cap P_b$ are mutually orthogonal. Therefore, we apply the above results and obtain

$$\begin{aligned}
&c_k^{p,P_a\cup P_b}[i] = c_k^{p,P_a\cup P_b}[j]\\
&\Rightarrow c_k^{p,\overline{P_a}\cup\overline{P_b}\cup(P_a\cap P_b)}[i] = c_k^{p,\overline{P_a}\cup\overline{P_b}\cup(P_a\cap P_b)}[j]\\
&\Rightarrow c_k^{p,\overline{P_a}}[i] \cup c_k^{p,\overline{P_b}}[i] \cup c_k^{p,P_a\cap P_b}[i]\\
&= c_k^{p,\overline{P_a}}[j] \cup c_k^{p,\overline{P_b}}[j] \cup c_k^{p,P_a\cap P_b}[j]\\
&\Rightarrow c_k^{p,\overline{P_a}}[i] = c_k^{p,\overline{P_a}}[i] \text{ and } c_k^{p,\overline{P_b}}[i] = c_k^{p,\overline{P_b}}[i]\\
&\quad \text{and } c_k^{p,P_a\cap P_b}[i] = c_k^{p,P_a\cap P_b}[i]\\
&\Rightarrow c_k^{p,P_a}[i] = c_k^{p,P_a}[j] \text{ and } c_k^{p,P_b}[i] = c_k^{p,P_b}[j]\\
&\Rightarrow \phi_k^{p,P_a}(i) = \phi_k^{p,P_a}(j) \text{ and } \phi_k^{p,P_b}(i) = \phi_k^{p,P_b}(j)\\
&\Rightarrow \langle \phi_k^{p,P_a}(i), \phi_k^{p,P_b}(i)\rangle = \langle \phi_k^{p,P_a}(j), \phi_k^{p,P_b}(j)\rangle\\
&\Rightarrow C_k^{p,P_a+P_b}[i] = C_k^{p,P_a+P_b}[j].
\end{aligned}$$

Finally, we determine that

$$C_k^{p,P_a+P_b}[i] = C_k^{p,P_a+P_b}[j] \Leftrightarrow c_k^{p,P_a\cup P_b}[i] = c_k^{p,P_a\cup P_b}[j]$$

and by definition,

$$\Phi_k^{p,P_a+P_b}(i) = \phi_k^{p,P_a\cup P_b}(i)$$ □

From Proposition 3, RFC tables for $P_a + P_b$ are made using RFC tables for P_a and P_b. To obtain $\phi_k^{p,P_a\cup P_b}$, we require bitmaps of size $|P_a \cup P_b|$, but we need only a 16-bit value for each partition; hence, two 16-bit values are needed to calculate $\phi_k^{p,P_a+P_b}(i)$. Therefore, the total calculation and memory costs are reduced by $\frac{|P_a\cup P_b|}{32}$ in this case. In general, $\Omega < |P_a \cup P_b| < 2\Omega$ and $\Omega \gg 16$; hence, we conclude that the merging technique is very efficient for most cases. For example, if $\Omega = 1{,}000$, we can decrease the costs by $\frac{32}{|P_a\cup P_b|}(< \frac{1}{30})$.

When we merge chunk tables in phase k, we should also regenerate chunk tables in the next phase. Assume that the

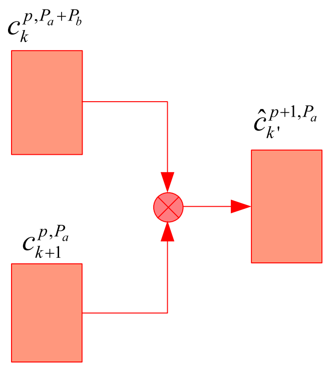


Fig. 8. Chunk table for phase $p+1$ created after merging P_a and P_b for phase p.

k'th chunk for phase $p+1$ is created from kth and $(k+1)$th CBMs for phase p. Then, $c_{k'}^{p+1,P_a}[i] = b_k^{p,P_a}[u] \cap b_{k+1}^{p,P_a}[v]$, for all non-negative integers i, u, v, where $i = u\left|b_{k+1}^{p,P_a}\right| + v$. Now, we will explain how each entry of the k'th chunk of phase $p+1$, $\hat{c}_{k'}^{p+1,P_a}[\cdot]$ is created after merging. Assuming that $b_{k'}^{p,P_a+P_b}[u_{a+b}] = \left\langle b_{k'}^{p,P_a}[u], b_{k'}^{p,P_b}[u_b]\right\rangle$, we obtain

$$\begin{aligned}\hat{c}_{k'}^{p+1,P_a}[j] &= b_k^{p,P_a+P_b}[u_{a+b}] \cap b_{k+1}^{p,P_a}[v]\\ &= \langle b_k^{p,P_a}[u], b_k^{p,P_b}[u_b]\rangle \cap b_{k+1}^{p,P_a}[v]\\ &= b_k^{p,P_a}[u] \cap b_{k+1}^{p,P_a}[v]\\ &= c_{k'}^{p+1,P_a}[i]\end{aligned}$$

where $j = u_{a+b}|b_{k+1}^{p,P_a}| + v$, $\forall j, u_{a+b}, v \in \mathbb{Z}^+$ as shown in Fig. 8. Now, from (3.3.2), we can build k'th chunk tables of phase $p+1$ from CBM tables of phase 0. Similarly, we can also calculate $\hat{c}_k^{p+1,P_b}[\cdot]$.

3.3.3 Searching in an Integrated Table

The procedure of searching in an integrated table consists of inter- and intra-partition search. For inter-partition search, we need to find a partition that includes a rule matching with the given keys from the partition decision tree embedded in the table. For example, in Fig. 7, if the given keys are *100.1.4.1* and *80* for DIP and DPORT, respectively, we can determine that the partition is one of P_1, P_2, and P_3, from the entry of key *100.1* in a table for DIP_U16. We also obtain the address of a table for DIP_L16, which is indicated by a blue dotted line. Then, we can see that the partition is one of P_2, P_3 from the entry corresponding to *4.1* in a table for DIP_L16. Finally, we can find the partition P_2 from the entry for *80* in a table for DPORT.

Now, we finish inter-partition search and start intra-partition search for P_2 to find a matching rule. First, we read a CBM ID value from the entry for *100.1* in a table for DIP_U16. The value is already loaded into the cache memory during inter-partition search; therefore, we can obtain it without slow DRAM access. Similarly, we can obtain each entry from the cache for DIP_U16 and DPORT. After that, we can find the matching rule by reading entries, which are not cached, from two DIP_U16 & DPORT and FINAL tables in turn.

We require only two memory accesses to perform intra-partition search; hence, the total number of DRAM accesses including inter- and intra-partition search is exactly the same as that of the original RFC. Through this feature, we can achieve very high packet classification performance.

3.4 Implementation Issues

Various implementation techniques can be used to improve the performance of the algorithm. One of the most effective techniques is to use a hard disk as a cache to save table images for each partition. When a ruleset is updated, all partitions are not always updated. Therefore, the probability that tables in the cache are reused to build the integrated table is very high, when the ruleset is updated. Thus, we can improve the performance in most cases.

In addition to caching, the use of multi-threads to build tables is highly effective. After partitioning, the process for building tables for each partition is independent of each other before the merging procedure. By building tables through multi-threaded algorithms, we can significantly reduce the table building and updating time.

4 Performance Evaluation

We evaluated the performance of the proposed algorithm by comparing it with existing packet classification algorithms. Our algorithm targets on enterprise-level security equipments. Since Intel Xeon platforms with multi-cores and large memory are very common for such equipments, we used a Xeon server equipped with Intel Xeon X5680 (3.33 GHz, hexa-core) and 12 GB physical memory for the evaluation environment. We installed Fedora Linux 18 64-bit on our platform.

For testing rulesets, we obtained a large ruleset from a firewall operating on Samsung's networks. The ruleset consists of 272,808 (273 K in short) rules. It is very important to determine how fast the performance is degraded as the ruleset size increases; therefore, we synthesized large rulesets by using a ruleset generation tool, Classbench to make 546 K, 1 M and 2 M rulesets[3] [8].

For comparison, we selected RFC and FRFC as non-partitioning algorithms [11], [16]. We also selected packet classification using maximum entropy hash (MEH, shortly) and HyperCuts as partitioning algorithms [3], [12].

For HyperCuts, we configured the parameters for the factor, bucket size and filter push level to 4, 32 and 2 which were found by trial and error to maximize the classification performance.

We implemented our testbed by using C++. It is very difficult to measure the accurate number of cache line accesses directly. This problem is resolved by an assumption that packet classification does not take any advantage of the prior cached data by the previous classifications since the cache hit ratio for such cases is very low for most large networks [23]. From this assumption, we can easily calculate the number of cache line accesses only for each packet classification in the testbed.

To measure the performance of packet classification, we generated keys by using each rule data from the ruleset, counted the total number of cache line accesses with the key data, and calculated the average performance.

3. Since some competitors could not work for larger rulesets than 2 M, we used only 273 K, 546 K, 1 M and 2 M rulesets.

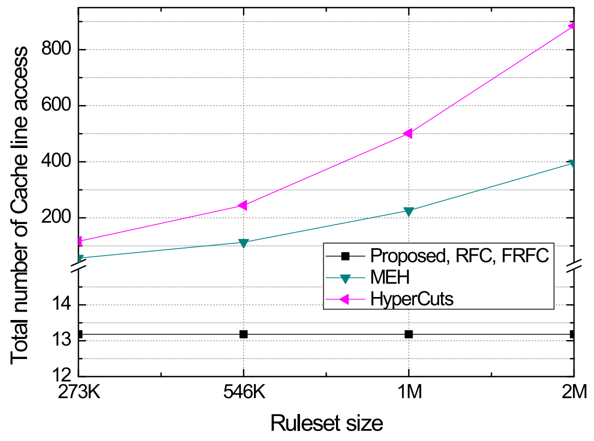


Fig. 9. Average number of cache line accesses for packet classification as a function of ruleset size.

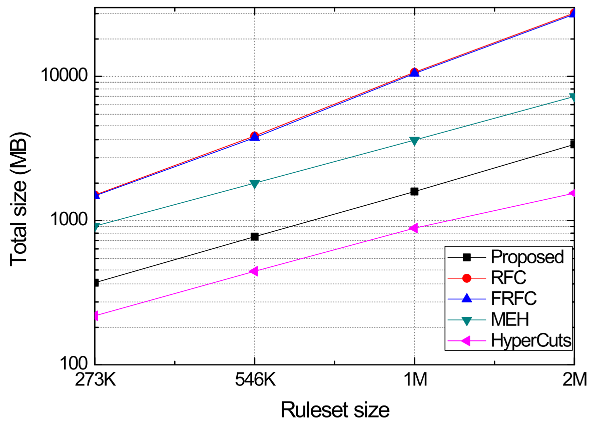


Fig. 10. Total size of tables for packet classification as a function of ruleset size.

4.1 Performance of Packet Classification

Since the performance is mainly determined by the total number of cache line accesses, we measured the average number for various total ruleset sizes from 273 K to 2 M. The result is shown in Fig. 9. RFC and FRFC require only 13 table accesses in all cases. However, MEH and HyperCuts require a larger number of accesses with larger rulesets because the partition size increases with the ruleset size. Moreover, they search linearly for intra-partition search, resulting in poor performance for large partitions.

The performance can be improved if we use a better algorithm for intra-partition search. However, we cannot expect that they will outperform cross-producting algorithms like RFC and FRFC. For example, the height of the tree of HyperCuts is already 13 that is the number of table accesses for the worst case, when the ruleset size is 273 K; thus, HyperCuts is definitely slower than the other algorithms if the intra-partitioning overhead is counted. Although our algorithm is based on partitioning, it shows the same performance as RFC and FRFC regardless of the ruleset size.

4.2 Table Size

The table size is critical to the scalability of the ruleset size. We measured the table size generated by each algorithm for various ruleset sizes from 273 K to 2 M.

As expected, in Fig. 10, cross-producting algorithms (RFC and FRFC) show the largest table sizes, and as the ruleset size increases, the table size increases faster than that of partitioning-based algorithms. In contrast, HyperCuts has the smallest table size, and it can be a good solution for network systems with a small memory size. However, the classification performance is poor, as shown in Fig. 9; therefore, our algorithm is a unique solution for achieving a small table size and high classification speed simultaneously. MEH shows poor performance in terms of the table size because it selects the number of fields and bits in a fixed manner, thus failing in adaptive partitioning.

4.3 Table Updating Time

A short table updating time is an essential requirement for system performance as well as efficient management of rulesets. For example, if a packet classification algorithm is used for firewalls, the updating time should be as short as possible for effectively blocking hostile traffic coming from external networks. The updating time will be longer when a large ruleset is used; hence, it is an important performance metric in large-scale high-speed networks.

We measured the rule updating time of each algorithm for ruleset sizes of 273 K, 576 K, 1 M and 2 M. The proposed algorithm rebuilds only tables of partitions that are affected by the updated ruleset. For unchanged partitions, tables cached in an external storage are reused to build the integrated tables. Thus, we can reduce the table building time efficiently. We also measured the updating time when the table cache is adopted. For precision, we built entire tables, changed a randomly selected rule, and measured the table updating time.

As shown in Fig. 11, the proposed algorithm achieves the shortest table updating time and it is nearly four times faster than MEH that shows the fastest performance among competitors, when the ruleset size is 2 M although no cache is used. When a cache is used, the performance is improved sixfold, compared to the case of using no cache. The tables are updated within 10 s, when the ruleset size is 2 M.

4.3.1 Table Updating Time versus Cache State

We also investigated the effect of cache states on the performance. First, we measured the updating time when the cache is empty. As shown in Fig. 12, we can see that when the cache is empty, the table updating time is increased

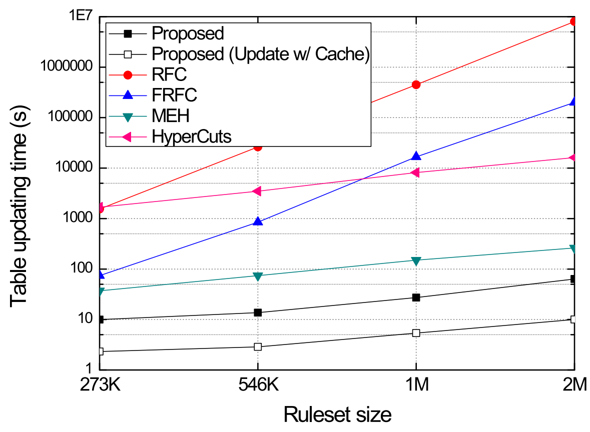


Fig. 11. Table updating time (estimated for RFC when the ruleset size is 2 M).

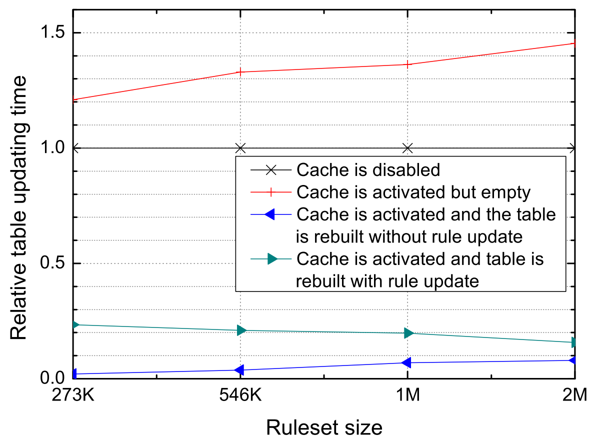


Fig. 12. Comparison of relative table updating time as a function of cache status.

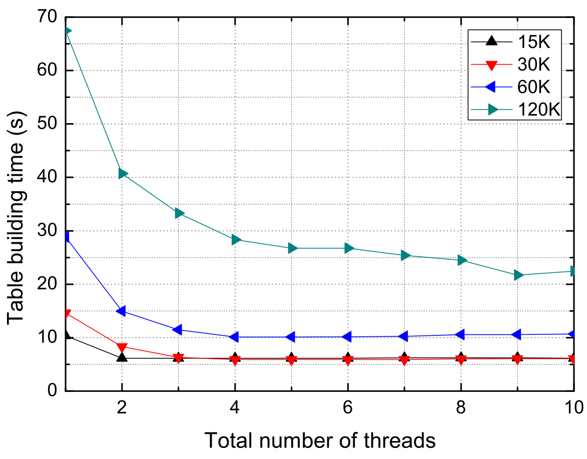


Fig. 13. Comparison of table updating time as a function of the number of threads.

by 20-40 percent, as compared to the case of using no cache, owing to the cache overhead. Second, we measured the table rebuilding time when the ruleset is not changed and all the tables in the cache are reused.

We also measured the table updating time by changing a randomly selected rule. When the cache is filled, the table updating time owing to the rule update is reduced by 77-84 percent, as compared to the case of using no cache. The performance gap increases with the ruleset size, which can be explained as follows. The total number of partitions and the partitioning efficiency also increase. Therefore, when a rule is updated, the ratio of the number of partitions to be updated to the total number of partitions decreases. Fig. 12 confirms that caching tables is very effective for performance improvement.

4.3.2 Table Updating Time versus Number of Threads

Because the proposed algorithm generates independent tables for each partition and builds integrated tables, it can exploit parallel processing when multiple threads are used to build the tables. For the total thread number from 1 to 10, we measured the table updating time, as shown in Fig. 13. The updating time decreases as the size of the ruleset and the number of threads increase. When the ruleset size is 273 K, 546 K, 1 M, and 2 M, the updating time of 10 threads is decreased by 31, 59, 63, and 67 percent, respectively, compared to a single thread case. This is because the number of partitions increases with the ruleset size. The table building time for each partition can vary, irrespective of the partition size, owing to the characteristics of the ruleset of the partition. Therefore, the number of partitions increases with the size of the ruleset, which is advantageous for assigning jobs more evenly for each thread.

5 Conclusion

In this paper, we proposed a new packet classification algorithm that employs partitioning and cache-aware integrated table structure. As packet classification has been used for various purposes, there are many requirements such as fast packet classification, fast ruleset update, small table sizes, and large ruleset sizes. Existing algorithms cannot satisfy all these requirements simultaneously. However, our algorithm can simultaneously support very large rulesets and fast packet classification, unlike existing algorithms. Therefore, the proposed algorithm can potentially enhance the poor scalability of existing algorithms. We expect our algorithm to play a critical role in high-performing network and network security systems.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF-2014R1A1A1038306). Y. -J. Choi is the corresponding author.

References

[1] D. E. Taylor, “Survey and taxonomy of packet classification techniques,” *ACM Comput. Surveys*, vol. 37, no. 3, pp. 238–275, Sep. 2005.
[2] D. Rovniagin and A. Wool, “The geometric efficient matching algorithm for firewalls,” *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 1, pp. 147–159, Jan./Feb. 2011.
[3] S. Singh, F. Baboescu, G. Varghese, and , J. Wang, “Packet classification using multidimensional cutting,” in *Proc. ACM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, Aug. 2003, pp. 213–224.
[4] M. Dixit, B. V. Barbadekar, and A. B. Barbadekar, “Packet classification algorithms,” in *Proc. IEEE Symp. Indus. Electron.*, Jul. 2009, pp. 1407–1412.
[5] S. Sahni, K. S. Kim, and H. Lu, “Data structures for one-dimensional packet classification using most specific-rule matching,” in *Proc. Int. Parallel Archit. Algorithm Netw.*, May 2002, pp. 1–12.
[6] V. Srinivasan, S. Suri, and G. Varghese, “Packet classification using tuple space search,” in *Proc. ACM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, Aug. 1999, pp. 135–146.
[7] P. Gupta, “Algorithms for Packet classification,” *IEEE Netw.*, vol. 15, no. 2, pp. 24–32, Mar./Apr. 2001.
[8] D. E. Taylor and J. S. Turner, “ClassBench: A packet classification benchmark,” in *Proc. IEEE Conf. Comput. Commun. Soc.*, Mar. 2005, pp 2068–2079.
[9] D. E. Taylor and J. S. Turner, “Scalable packet classification using distributed crossproducting of field labels,” in *Proc. IEEE Conf. Comput. Commun. Soc.*, Mar. 2005, pp 269–280.
[10] J. van Lunteren and T. Engbersen, “Fast and scalable packet classification,” *IEEE J. Sel. Areas Commun.*, vol. 21, no. 4, pp. 560–571, May 2003.
[11] P. Gupta and N. McKeown, “Packet classification on multiple fields,” in *Proc. ACM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, Aug. 1999, pp. 147–160.

[12] 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, Dec. 2009.

[13] B. Florin and V. George, "Scalable packet classification," in *Proc. ACM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, Aug. 2001, pp. 199–210.

[14] T. V. Lakshman and D. Stidialis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, Sep. 1998, pp. 203–214.

[15] P.-C. Wang, "Scalable packet classification with controlled cross-producting," *Comput. Netw.*, vol. 53, no. 6, pp. 821–834, Apr. 2009.

[16] W. Pak and S. Bahk, "FRFC: Fast table building algorithm for recursive flow classification," *IEEE Commun. Lett.*, vol. 14, pp. 1182–1184, Dec. 2010.

[17] F. Geraci, M. Pellegrini, P. Pisati, and L. Rizzo, "Packet classification via improved space decomposition techniques," in *Proc. IEEE Conf. Comput. Commun. Soc.*, Mar. 2005, pp. 304–312.

[18] A. Feldmann and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proc. IEEE Conf. Comput. Commun. Soc.*, Mar. 2000, pp. 1193–1202.

[19] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proc. ACM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, Sep. 1999, pp. 191–202.

[20] X. Sun, S. K. Sahni, and Y. Q. ZhaoX, "Packet classification consuming small amount of memory," *Trans. IEEE/ACM Netw.*, vol. 13, no. 5, pp. 1135–1145, Oct. 2005.

[21] J. Xu, M. Singhal, and J. Degroat, "A novel cache architecture to support Layer-four packet classification at memory access speeds," in *Proc. IEEE Conf. Comput. Commun. Soc.*, Mar. 2000, pp. 1445–1454.

[22] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *Proc. ACM Conf. Appl., Technol., Archit. Protocols Comput. Commun.*, Oct. 1997, pp. 25–36.

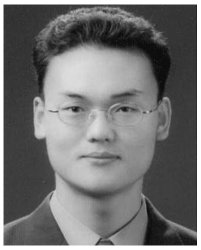
[23] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 lookups using distributed and load balanced bloom filters for 100 Gbps core router line cards," in *Proc. IEEE Conf. Comput. Commun. Soc.*, Apr. 2009, pp. 2518–2526.

[24] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. New York, NY, USA: Wiley, May 2008.

[25] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. IEEE Conf. Found. Comput. Sci.* Oct. 1999, pp. 285–297.

[26] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, "NP-hardness of Euclidean Sum-of-squares clustering," *Mach. Learn.*, vol. 75, pp. 245–249, 2009.

[27] S. Dasgupta and Y. Freund, "Random projection trees for vector quantization," *Trans. IEEE Inf. Theory*, vol. 55, no. 7, pp. 3229–3242, Jul, 2009.



Wooguil Pak received the BS and MS degrees in electrical engineering from Seoul National University in 1999 and 2001, respectively, and the PhD degree in electrical engineering and computer science from Seoul National University in 2009. In 2010, he joined the Jangwee Research Institute for National Defence as a research professor. In 2013, he is currently an assistant professor at Keimyung University, Daegu, Korea. His current research interests include MAC and routing protocol design for flying ad-hoc network, wireless sensor network and wireless mesh network, and network security for high-speed networks. He is a member of the IEEE.

Young-June Choi received the BS, MS, and PhD degrees from the Department of Electrical Engineering & Computer Science, Seoul National University, in 2000, 2002, and 2006, respectively. From September 2006 through July 2007, he was a postdoctoral researcher at the University of Michigan, Ann Arbor, MI. From 2007 through 2009, he was with the NEC Laboratories America, Princeton, NJ, as a research staff member. He is currently an assistant professor at Ajou University, Suwon, Korea. His research interests include radio resource management, mobile security, and cognitive radio networks. He received the Gold Prize at the Samsung Humantech Thesis Contest in 2006. He is a member of the IEEE.