Applied Mathematics & Information Sciences
An International Journal

Priority Area-based Quad-Tree Packet Classification Algorithm and Its Mathematical Framework

Hyesook Lim[1], *Changhoon Yim*[2], *and Earl E. Swartzlander, Jr.*[3]

[1] Department of Electronics Engineering, Ewha W. University, Seoul, Korea
[2] Department of Internet and Multimedia Engineering, Konkuk University, Seoul, Korea
[3] Department of Electrical and Computer Engineering, The University of Texas at Austin, TX 78712, USA

Received: 27 Jul. 2012; Accepted 15 Sept. 2012
Published online: 1 January 2013

Abstract: Packet classification is an essential function for next-generation Internet routers to provide high quality of service. Packet classification using multiple header fields is a challenging problem that should be performed at wire speed for all incoming packets. This paper proposes a novel mathematical framework for packet classification problem. Then the priority area-based quad-tree (PAQT) packet classification algorithm combining priority search and recursive space decomposition is formally described using the framework. The validity of the PAQT algorithm is mathematically proved for theoretical justification. The proposed mathematical framework can be applied to other packet classification algorithms for formal description and theoretical justification. Extensive simulation results demonstrate that the PAQT algorithm has very good performance compared to other packet classification algorithms in terms of search speed, memory size, and scalability.

Keywords: Internet, router, packet classification, priority search, quad-tree, space decomposition.

1. Introduction

Traditional routers provide the best-effort service to all incoming packets, and hence they treat every incoming packet equally. However, emerging multimedia applications request multiple levels of quality of service (QoS) [1]-[4]. Advanced routers can use packet classification to support higher level functions such as QoS routing and access control [5]. For the Internet routers to provide the requested QoS, they should provide various features such as admission control, resource reservation, per-flow queuing, and output scheduling [6].

As an essential pre-requisite for these features, packets need to be classified into multiple flows. The rules for packet classification consists of a set of fields and an associated action [7]. Each rule in a classification table has a priority which is defined by multiple fields. Header fields included in a given input packet are compared with corresponding rule fields, and multiple rules can match to an input packet. The highest priority rule is selected among the matching rules. Most of the previous packet classification algorithms have a trade-off between the required memory size and the search speed. The search speed is often measured by the number of memory accesses since memory access is the most time consuming operation in the search procedure.

Ternary content addressable memory (TCAM) has been widely used in commercial routers since TCAM provides very good search performance [11], [27], [28], [29], [30]. However, TCAM consumes a lot of power and about six times larger than ordinary memory. Studies have been conducted to reduce the required space [28], to reduce the power consumption [29], and to improve the throughput of TCAM [30]. The large size and power consumption make large classifiers implemented with TCAM impractical.

As a basic algorithmic approach, trie-based algorithms use a source IP prefix and a destination IP prefix to build tries [9], [15], [20], [24]. A hierarchical trie (H-trie) [13] constructs the first trie using the source prefix field, and each node of the trie connects to the second trie constructed with the destination field rules with the same source field so that both fields are searched simultaneously. In H-trie, searching for matching rules has to be continued until a

* Corresponding author: e-mail: cyim@konkuk.ac.kr

leaf of the first trie is visited even if a match is already found in a higher level because of the possibility that other nodes with higher priorities will match. Set-pruning tries [13] improve the search time of the H-trie by copying all the possible matching rules in the search path into the leaves. It improves the search time, but requires huge amount of memory. The approach used in the grid-of-tries [24] and the extended grid-of-tries [9] overcomes the disadvantage of rule duplication by pre-computing the best matching rule of each node and storing switch pointers from one node to another. The intrinsic problem of trie-based algorithms is that there are often a large number of empty nodes which do not store any rule, resulting in a large memory requirement and a slow search speed.

The HiCuts (hierarchical intelligent cuttings) [14] and HyperCuts [22] algorithms partition a multi-dimensional search space based on heuristics that exploit the structure of classifiers. The decision tree characteristics such as its depth, the degree of the node, and the local search decision to be made at each node are determined in a pre processing step based on the structure of the classifier. Each leaf node in the tree includes a pre-determined small number of rules that can be searched linearly. The cutting algorithms have some issues that the search speed is highly dependent on the characteristics of the classifiers and excessive pre processing time may be required.

The tuple-space search algorithm [23] decomposes a classification query into a number of exact match queries. Since the number of tuples consisting of the source and destination prefixes is usually small, the algorithm decomposes a classification table into multiple tuples. Rules mapped to the same tuple have fixed lengths, so they can be stored in a hash table. Each tuple is searched sequentially by hashing, thus the classification speed depends on the number of tuples.

In the cross-product algorithm [24], the distinct values in each field of the classification table are listed and a cross-product table combining them is constructed. The best matching rule is pre-computed and stored in the cross-product table. In search, the results of each field search are combined to produce a pointer to the cross-product table. This algorithm provides fast classification since packet classification is done with multiple one-dimensional field searches and a lookup to the cross-product table, but it has the disadvantage that it requires huge memory for tables in each field and the cross-product table. Moreover it is difficult to update because of the pre-computation required to build the cross-product table.

The geometric interpretation of the packet classification is based on the fact that a k-dimensional rule represents a hyper-rectangle in a k-dimensional space. A packet header represents a point (a smallest hyper-rectangle) in k-dimensional space. Classifying a packet is finding the highest-priority hyper-rectangle that contains the point using space decomposition [10],[12],[18]. In the area-based quad-tree (AQT) algorithm [10], a 2-dimensional (2D) search space using source and destination prefix fields is recursively partitioned into four equal-sized spaces. If the partitions are repeated L times, 4^L equal-sized spaces are obtained, and each space is represented by L-bit prefix pairs. Each rectangular search space is mapped into a node in a quad-tree. Thus the entire space is mapped into the root node of the quad-tree, and four equal-sized quadrants which partition the space are mapped into four children of the root node, etc. The AQT defines a crossing filter as a rule which spans at least one dimension of the rectangular space. Rules that belong to a crossing filter set (CFS) are stored into the corresponding quad-tree node. The AQT is an efficient data structure, but it has some issues which affect its practicality. The search speed is directly related to the depth of the quad-tree, and the depth of the quad-tree in AQT is usually the maximum prefix length. Moreover, the quad-tree may be very sparse with many empty internal nodes that waste memory space and increase the number of memory accesses.

The basic idea of a priority-based quad-tree algorithm was presented in [18], in which crossing filters in AQT are not necessarily stored into the corresponding quad-tree node. If inputs are compared with prefix values, rules can be stored in a node at a higher level. However the priority-based quad-tree algorithm was not theoretically justified and only preliminary simulation results were presented in [18].

In this paper, we first propose a new mathematical framework for packet classification problem. Priority search and recursive space decomposition concepts are described in this mathematical framework. Then this paper presents the priority area-based quad-tree (PAQT) algorithm using the novel mathematical framework. The data structure, build process, search process, and update process of the PAQT algorithm are presented using the mathematical framework. Especially the validity of the build and search process is proved to provide the theoretical justification of PAQT algorithm. Extensive simulation results are presented to compare the PAQT algorithm with various packet classification algorithms.

This paper is organized as follows. In Section 2, packet classification problem is formulated in a new mathematical framework. Section 3 presents the priority search and the recursive space decomposition using the framework, which are the basis of the PAQT algorithm. In Section 4, we present the PAQT algorithm in terms of data structure, build process, search process, update process, and scalability, with an implementation example. The validity of PAQT algorithm is proved mathematically in this section. Simulation results to compare with various packet classification algorithms are presented in Section 5. Finally, Section 6 concludes the paper.

2. Packet Classification Problem

Let $\mathcal{F} = \{F_1, F_2, \cdots, F_N\}$ represent a classifier where F_i $(i = 1, \cdots, N)$ is a rule, and N is the number of rules in the classifier. Each rule consists of multiple fields. For each rule F_i, let $F_i = (F_i[1], \cdots, F_i[k])$, where $F_i[j]$ $(j =$

$1, \cdots, k)$ is the jth field. A field can be expressed as a range $F_i[j] = [b, e]$, where b and e are the beginning and end points of the field, respectively [25].

When a packet arrives, header values $H[j](j = 1, \cdots, k)$ from the relevant k fields in the packet header are extracted, and the headers are expressed as $H = (H[1], \cdots, H[k])$, where each field is a bit string.

Let $r_1 = [b_1, e_1]$ and $r_2 = [b_2, e_2]$ be two ranges in the same field (or dimension), where range inclusion is defined as $r_1 \in r_2$ if $b_1 \geq b_2$ and $e_1 \leq e_2$. A packet with header fields $H = (H[1], \cdots, H[k])$ matches rule $F_i = (F_i[1], \cdots, F_i[k])$, if $H[j] \in F_i[j]$ for all $j = 1, \cdots, k$ [25].

Rules in a classifier are usually sorted in priority order and each rule has a priority index. This priority index is necessary because a packet could match more than one rule. In this case, the rule with the highest priority is selected. In this paper, it is assumed that rules are sorted in order of decreasing priority (or increasing priority index). In a classifier $\mathcal{F}$, F_1 has the highest priority and F_N has the lowest priority. When a packet matches multiple rules, the rule with the smallest priority index (the highest priority) is defined as the best matching rule (BMR). If a packet H matches multiple rules, the rule with the smallest priority index is selected as the BMR.

Let $\mathcal{M}(H)$ represent the set of rules that a packet H matches $\mathcal{M}(H) = \{F_i \in \mathcal{F} : H[j] \in F_i[j], \forall j = 1, \cdots, k\}$. Let $P(\mathcal{C})$ represent the smallest priority index of a set of rules, $\mathcal{C}$. The packet classification problem is to find the smallest priority index in the set of matched rules, $P(\mathcal{M}(H))$, in the set of rules in the classifier, $\mathcal{F} = \{F_1, F_2, \cdots, F_N\}$, for a given packet $H = (H[1], \cdots, H[k])$.

3. Priority Search and Space Decomposition

3.1. Priority Search

In most of the previous packet classification algorithms except the simple linear search, the set of matching candidates is first searched for a given packet H, and then the highest priority rule is selected. Thus all the rules belonging to $\mathcal{M}(H)$ are first searched in $\mathcal{F}$ before $P(\mathcal{M}(H))$ is searched.

However, in the linear search, not all of the elements (rules) of $\mathcal{M}(H)$ need to be searched to find $P(\mathcal{M}(H))$. The packet matching condition, $H[j] \in F_i[j], \forall j = 1, \cdots, k$, is checked in the order of $F_1, F_2, \cdots, F_N$, i.e., in decreasing priority order. The search is completed when either a match is found or the last rule is examined. If the match is found at F_m, it is concluded that $m = P(\mathcal{M}(H))$. In this case, H has not matched any of $F_1, \cdots, F_{m-1}$, and F_m is the BMR with the smallest priority index. Although the linear search can find the BMR without searching the entire set of rules belonging to $\mathcal{M}(H)$, it has a problem that the search space is reduced only by $1/N$ with each memory access. Hence it is not scalable to large classifiers.

The PAQT packet classification algorithm combines priority search and recursive space decomposition. The priority search is used to find the BMR without searching entire rules in $\mathcal{M}(H)$, and the recursive space decomposition is used to reduce the search space exponentially with each memory access.

3.2. Recursive Space Decomposition

The recursive space decomposition can be described either in 1-dimensional (1D) space or in multi-dimensional space. In 1D space, among fields comprising rules, either the source prefix or destination prefix is generally used in recursively partitioning the 1D line. In 2-dimensional (2D) space, both the source and the destination prefixes are considered at the same time in recursively partitioning the 2D area. While a range in 1D line corresponds to a node in a binary trie, a rectangle in 2D area corresponds to a node in a quad-tree (or quad-trie) in the recursive decomposition. In this paper, the PAQT algorithm is presented for the 2D case, but it can be extended easily to higher dimensions.

Rules are also called *filters*. In this paper, the term filter is used for a rule that relates to a 2D space of the source and destination prefixes. If W is the maximum prefix length, the size of a 2D search space is $2^W \times 2^W$. If the length of the source and the destination prefixes are s and d, respectively, a filter is a rectangle with the size of $2^{W-s} \times 2^{W-d}$. A node in a quad-tree corresponds to a 2D rectangle. The root node of the quad-tree corresponds to the entire 2D space, and four children of the root node correspond to four equal-sized quadrants, etc.

For a node v, let $A(v)$ represent the area which corresponds to the node v. An area $A(v)$ can be represented as ranges, i.e., $A(v) = (a_v[1], a_v[2]) = ([b_1, e_1], [b_2, e_2])$.

Definition 1 *Area $A(v)$ and $A(w)$ are disjoint, if $A(v) \cap A(w) = \emptyset$.*

Theorem 1. *Area $A(v) = (a_v[1], a_v[2])$ and $A(w) = (a_w[1], a_w[2])$ are disjoint, if $a_v[1] \cap a_w[1] = \emptyset$ or $a_v[2] \cap a_w[2] = \emptyset$.*

The relation between a filter $F_i = (F_i[1], \cdots, F_i[k])$ and an area $A = (a[1], a[2])$ is defined by the first two fields, $F_i[1]$ and $F_i[2]$, which are the source and destination prefixes, respectively.

Definition 2 *A filter $F_i = (F_i[1], \cdots, F_i[k])$ is included in an area $A = (a[1], a[2])$, i.e., $F_i \in A$, if $F_i[1] \in a[1]$ and $F_i[2] \in a[2]$.*

Let $\mathcal{I}(A)$ represent the set of filters that are included in an area A. When n filters are included in A, $\mathcal{I}(A)$ can be represented as $\mathcal{I}(A) = \{F_{i_1}, \cdots, F_{i_n}\}$, if $F_{i_p} \in A$ for all $p = 1, \cdots, n$. If the set $\mathcal{I}(A) = \{F_{i_1}, \cdots, F_{i_n}\}$ is sorted in the order of the increasing priority index, then $P(\mathcal{I}(A)) = i_1$. Crossing filters are defined using the source and the destination prefixes in the 2D space [10]. Let $\mathcal{C}(A)$ represent the set of filters that cross the area A. The crossing filter is formally defined as follows.

Definition 3 *A filter $F_i = (F_i[1], \cdots, F_i[k])$ is a crossing filter of area $A = (a[1], a[2])$ if $(F_i[1] = a[1]$ and $F_i[2] \in a[2])$ or $(F_i[1] \in a[1]$ and $F_i[2] = a[2])$.*

Theorem 2. *For area A, $\mathcal{C}(A) \subset \mathcal{I}(A)$.*

Theorem 2 can be proved from Definition 2 and Definition 3.

4. The PAQT Algorithm

This section presents the priority area-based quad-tree (PAQT) algorithm using the proposed mathematical framework including data structure, build process, search process, update process, and scalability. Especially the validity of build and search process is mathematically proved to provide the theoretical justification. An implementation example is also presented for easier adaptation of mathematical formulation.

4.1. Data Structure

Let $\mathcal{S}(v)$ represent the set of filters that are stored in node v in the build process. In the AQT [10], only the crossing filters of $A(v)$ are stored in node v, i.e., $\mathcal{S}(v) = \mathcal{C}(A(v))$. Hence if there is no crossing filter of $A(v)$, then $\mathcal{S}(v)$ would be empty in the AQT. However, filters are not necessarily stored into the corresponding node, only when one of their fields crosses an area. If a filter is stored in a lower level node than its crossing level node, the filter should be replicated into multiple filters, but there is no reason not to store a filter in a higher level node.

In the proposed data structure, not only the crossing filters, but also the highest priority filter that are included in $A(v)$ are stored. In the proposed data structure, the set $\mathcal{S}(v)$ for a node at level l is formally defined as

$$\mathcal{S}(v) = \mathcal{C}(A(v)) \cup \{F_m\},\ m = P(\mathcal{I}(A(v)) \cap \mathcal{F}_l) \quad (1)$$

where $\mathcal{F}_l$ is the set of filters that can be stored at level l and it is updated from the initial classifier $\mathcal{F}$ in the build process. Thus $\mathcal{F}_l$ is composed of the filters that are not stored in nodes at levels 0 through $l-1$, that are described in the next subsection.

Theorem 3. $\mathcal{S}(v) \subset \mathcal{I}(A(v))$.

Proof In Eq. (1), the filter F_m has the smallest priority index among the filters in $\mathcal{I}(A(v)) \cap \mathcal{F}_l$. Hence $\{F_m\} \subset (\mathcal{I}(A(v)) \cap \mathcal{F}_l) \subset \mathcal{I}(A(v))$. By Theorem 2, $\mathcal{C}(A(v)) \subset \mathcal{I}(A(v))$. Thus $\mathcal{S}(v) = \mathcal{C}(A(v)) \cup \{F_m\} \subset \mathcal{I}(A(v))$.

Theorem 4. *If $\mathcal{I}(A(v)) \cap \mathcal{F}_l$ is not empty for a node at level l, then $\mathcal{S}(v)$ is not empty.*

Proof If $\mathcal{I}(A(v)) \cap \mathcal{F}_l$ is not empty, there exist F_{i_p}'s in $\mathcal{F}_l$ for $p = 1, \cdots, n$ where n is the number of elements in $\mathcal{I}(A(v)) \cap \mathcal{F}_l$. Then there exists the filter with the smallest priority index (the highest priority) F_{i_m} such that $i_m < i_p$ for all $p = 1, \cdots, n, p \neq m$. Hence $F_{i_m} \in \mathcal{S}(v)$ and there exists at least one filter in $\mathcal{S}(v)$.

Theorem 5. *Let $A(v)$ and $A(w)$ be two disjoint areas, $A(v) \cap A(w) = \emptyset$. If an input packet H matches a filter F_m in $\mathcal{I}(A(v))$, then H cannot match any filter in $\mathcal{I}(A(w))$.*

Proof Let $A(v) = (a_v[1], a_v[2])$ and $A(w) = (a_w[1], a_w[2])$. Since $A(v)$ and $A(w)$ are disjoint, $a_v[1] \cap a_w[1] = \emptyset$ or $a_v[2] \cap a_w[2] = \emptyset$. Since $H = (H[1], H[2])$ matches F_m, $H[1] \in F_m[1]$ and $H[2] \in F_m[2]$. Since $F_m \in \mathcal{I}(A(v))$, $F_m[1] \in a_v[1]$ and $F_m[2] \in a_v[2]$. Hence $H[1] \in a_v[1]$ and $H[2] \in a_v[2]$. If $a_v[1] \cap a_w[1] = \emptyset$, $H[1] \notin a_w[1]$. If $a_v[2] \cap a_w[2] = \emptyset$, $H[2] \notin a_w[2]$. Hence $(H[1], H[2]) \notin A(w)$, and H cannot match any filter in $\mathcal{I}(A(w))$.

Theorem 6. *Let $A(v)$ and $A(w)$ be two disjoint areas, $A(v) \cap A(w) = \emptyset$. If input packet H matches a filter F_m in $\mathcal{S}(v)$, then H cannot match any filter in $\mathcal{S}(w)$.*

Proof By Theorem 3, $\mathcal{S}(v) \subset \mathcal{I}(A(v))$ and $\mathcal{S}(w) \subset \mathcal{I}(A(w))$. If input packet H matches F_m in $\mathcal{S}(v)$, then H matches F_m in $\mathcal{I}(A(v))$. By Theorem 5, H cannot match any filter in $\mathcal{I}(A(w))$. Hence H cannot match any filter in $\mathcal{S}(w)$.

4.2. Build Process

The build process begins at a root node. Let v_r be the root node at level 0 and let $A(v_r)$ represent the 2D space which corresponds to the root node. If a classifier $\mathcal{F}$ is sorted in the order of increasing priority index, F_1 has the smallest priority index. Since $A(v_r) = (a[1], a[2])$ is the entire 2D space, $a[1] = [0, 2^W - 1]$ and $a[2] = [0, 2^W - 1]$, where W is 32 for IPv4. If the source prefix of F_i is a wildcard (with prefix length 0), $F_i[1] = [0, 2^W - 1] = a[1]$. Similarly if the destination prefix of F_i is a wildcard, $F_i[2] = [0, 2^W - 1] = a[2]$. The crossing filter set of the root node v_r is composed as

$$\mathcal{C}(A(v_r)) = \{F_i \in \mathcal{F} \ :\ F_i[1] = [0, 2^W - 1] \text{ or } F_i[2] = [0, 2^W - 1]\}. \quad (2)$$

The set of filters that are stored in the root node v_r in the PAQT packet classification algorithm is

$$\mathcal{S}(v_r) = \{F_1\} \cup \mathcal{C}(A(v_r)). \quad (3)$$

The filters in $\mathcal{S}(v_r)$ are stored in a node v_r as a linked list in order of increasing priority index. The set of filters that can be stored in the nodes of level 1 is now updated as

$$\mathcal{F}_1 = \mathcal{F} - \mathcal{S}(v_r). \quad (4)$$

Equation (4) means that once a filter is stored in a node, this filter is removed from the current set of rules and is not stored in any other node at lower levels.

After the set $\mathcal{S}(v_r)$ for the root node v_r is constructed, the search space $A(v_r)$ is decomposed into four equal-sized rectangles, $A(v_0)$, $A(v_1)$, $A(v_2)$, and $A(v_3)$, and they correspond to source and destination prefix pair (0*, 0*), (0*, 1*), (1*, 0*), and (1*, 1*), respectively.

Note that these four areas are disjoint. In each node v_i $(i = 0, 1, 2, 3)$, the smallest priority index in the filter set $\mathcal{F}_1$, which is included in each $A(v_i)$, i.e., $m_i = P(\mathcal{I}(A(v_i)) \cap \mathcal{F}_1)$ is identified, and the crossing filter set $\mathcal{C}(A(v_i))$ is composed for each area $A(v_i)$. For example, $\mathcal{C}(A(v_0))$ comprises

$$\mathcal{C}(A(v_0)) = \{F_i \in \mathcal{F}_1 \ : \ F_i[1] = [0, 2^{W-1} - 1] \text{ or } F_i[2] = [0, 2^{W-1} - 1]\}. \quad (5)$$

Other crossing filter sets, $\mathcal{C}(A(v_1))$, $\mathcal{C}(A(v_2))$, and $\mathcal{C}(A(v_3))$ can be constructed similarly. Now $\mathcal{S}(v_i) = \{F_{m_i}\} \cup \mathcal{C}(A(v_i))$ for $i = 0, 1, 2, 3$ from filter set $\mathcal{F}_1$ at level 1. If $\mathcal{I}(A(v_i)) \cap \mathcal{F}_1$ is empty for any i, then node v_i becomes invalid. This invalid node is not generated in the build process. Hence there is no empty node in the PAQT data structure in contrast to the AQT [10] data structure.

The set of filters that can be stored in the nodes of level 2 is similarly updated as

$$\mathcal{F}_2 = \mathcal{F}_1 - \bigcup_{i=0}^{3} \mathcal{S}(v_i). \quad (6)$$

The search space for each $A(v_i)$ is decomposed into four equal-size rectangles, etc. The build process continues until the set of filters that should be stored is empty. Let $L(v)$ represent the level of a node v. For example, $L(v_r) = 0$ and $L(v_0) = 1$. Equation (6) can be generalized to the set of filters that can be stored at level $l + 1$ as

$$\mathcal{F}_{l+1} = \mathcal{F}_l - \bigcup_{\{v:L(v)=l\}} \mathcal{S}(v). \quad (7)$$

Note that the set of filters at a lower level is a subset of the set of filters at a higher level since

$$\mathcal{F}_{l+1} \subset \mathcal{F}_l \subset \cdots \subset \mathcal{F}_1 \subset \mathcal{F}. \quad (8)$$

Theorem 7. *If $L(v) = l$ and $\mathcal{F}_l$ is the set of filters at level l, $P(\mathcal{S}(v)) = P(\mathcal{I}(A(v)) \cap \mathcal{F}_l)$.*

Proof From Eq. (1), $\mathcal{S}(v) = \mathcal{C}(A(v)) \cup \{F_m\}$, $m = P(\mathcal{I}(A(v)) \cap \mathcal{F}_l)$. Since $\mathcal{C}(A(v)) \subset (\mathcal{I}(A(v)) \cap \mathcal{F}_l)$, $P(\mathcal{C}(A(v))) \geq P(\mathcal{I}(A(v)) \cap \mathcal{F}_l)$ and $P(\mathcal{C}(A(v)) \cup \{F_m\})$ $= \min(P(\mathcal{C}(A(v)), m) = m$. Hence $P(\mathcal{S}(v)) = P(\mathcal{I}(A(v))$ $\cap \mathcal{F}_l)$.

Theorem 8. *If w is a valid child node of v, then $P(\mathcal{S}(v)) < P(\mathcal{S}(w))$.*

Proof Let $m = P(\mathcal{S}(v))$, $n = P(\mathcal{S}(w))$, $L(v) = l$, and $\mathcal{F}_l$ be the set of remaining filters at level l. Since w is a child node of v, $L(w) = l + 1$. From Eq. (1), $\mathcal{S}(v) = \mathcal{C}(A(v)) \cup \{F_m\}$. From Eq. (7), $\mathcal{F}_{l+1} = \mathcal{F}_l - \bigcup_{\{v:L(v)=l\}} \mathcal{S}(v)$. Since w is a child node of v, $A(w) \subset A(v)$ and $\mathcal{I}(A(w)) \subset \mathcal{I}(A(v))$. By Theorem 7, $m = P(\mathcal{I}(A(v)) \cap \mathcal{F}_l)$ and $n = P(\mathcal{I}(A(w)) \cap \mathcal{F}_{l+1})$. Since $n = P(\mathcal{S}(w))$, $F_n \subset \mathcal{I}(A(w)) \subset \mathcal{I}(A(v))$ and $F_n \subset \mathcal{F}_{l+1} \subset \mathcal{F}_l$. Hence $F_n \subset \mathcal{I}(A(v)) \cap \mathcal{F}_l$. By the definition in Eq. (1), m is the smallest priority index in $\mathcal{I}(A(v)) \cap \mathcal{F}_l$. Thus m is less than any other priority index in $\mathcal{I}(A(v)) \cap \mathcal{F}_l$. Hence $m < n$, i.e., $P(\mathcal{S}(v)) < P(\mathcal{S}(w))$.

Corollary 1. *If w is a valid descendent node of node v, then $P(\mathcal{S}(v)) < P(\mathcal{S}(w))$.*

Corollary 1 can be proved by induction from Theorem 8.

4.3. Search Process

For classifier $\mathcal{F}$, assume that filters are sorted in decreasing priority (or increasing index) order.

During initialization, the matching rule (MR) is first set as $N + 1$ (default class) and v is set as the root node. Then $\mathcal{I}(A(v)) = \mathcal{F}$ and $P(\mathcal{I}(A(v)) \cap \mathcal{F}) = 1$. Pseudo-code for the search process can be represented as follows:

```
Get Packet H = (H[1], ··· , H[k])
MR = N + 1;  // Initial value of MR (default class)
m = P(S(v));
do
    if (H[j] ∈ F_m[j] for all j = 1, ··· , k) // Input packet matches filter F_m
        if (m < MR)  // The BMR is found (Theorem 9)
            MR ← m;  break;
    else if (C(A(v)) is not empty)  // Crossing filters exist
        C(A(v)) = {F_{i_1}, ··· , F_{i_n}}
        //Crossing filter set constructed in build process
        for p ← 1 to n do
            if (H[j] ∈ F_{i_p}[j] for all j = 1, ··· , k)
                if (i_p < MR)
                    MR ← i_p; break;
    v ← w;
    // w is the child node of v identified by the next one bit of H[1] and H[2]
    m = P(S(v));
while (v is a valid node and m < MR)
// Search ends at a leaf or if m > MR (Corollary 2)
BMR ← MR;
return BMR;
```

Theorem 9. *Let v be a node with $L(v) = l$ and $m = P(\mathcal{S}(v))$. If an input packet H matches F_m and m is less than the smallest priority index among current matched rules ($m <$ MR), then $m = P(\mathcal{M}(H))$, i.e., F_m is the rule with the best match in the given classifier $\mathcal{F}$.*

Proof We prove by induction. If $L(v) = 0$, v is the root node and $m = P(\mathcal{S}(v)) = 1$. The MR is initially set as $N + 1$. H matches F_1 and $m <$ MR. Since 1 is the smallest possible priority index, F_1 is the BMR and $m = P(\mathcal{M}(H)) = 1$. Assume that MR is the smallest priority index among current matched rules in nodes at levels 0 through $l - 1$. Actually this MR might be the default class $(N+1)$ or a crossing filter of a node at levels $0, \cdots, l-1$. For a node v with $L(v) = l$, assume that $m = P(\mathcal{S}(v))$ and $m <$ MR. There can be four cases that the BMR in $\mathcal{F} = \{F_1, F_2, \cdots, F_N\}$ can exist. Case 1: The BMR exists in a node at levels $0, \cdots, l-1$. Case 2: The BMR exists in a node u $(u \neq v)$ at level l. Case 3: The BMR exists in a node v at level l. Case 4: The BMR exists in a descendent node w of v at a lower level than l $(L(w) \geq l + 1)$. Since $m = P(\mathcal{S}(v))$ for a node v with $L(v) = l$ and $m <$ MR, the MR cannot be the BMR and the BMR does not exist at a node at levels $0, \cdots, l - 1$, and hence Case 1 is eliminated. Assume that u is a node at level l and $u \neq v$. Then $A(u)$ and $A(v)$ are disjoint. By the Theorem 6, the input packet H cannot match any filter in $\mathcal{S}(u)$. Hence the BMR cannot exist in a node u $(u \neq v)$ at level l, and hence Case 2 is eliminated. If there is no matching rule in the descendent nodes of v, then F_m in the node v is the BMR. Assume that a matching rule exists in a descendent node w of v. By Theorem 1, $P(\mathcal{S}(v)) < P(\mathcal{S}(w))$, i.e., the smallest priority index in $\mathcal{S}(w)$ is greater than the smallest priority index in $\mathcal{S}(v)$. Hence the BMR cannot exist in a descendent node of v at lower levels, and hence Case 4 is eliminated. Thus descendent nodes of v at lower levels do not need be searched. Thus the matching rule F_m in the node v at level l (Case 3) is the BMR and $m = P(\mathcal{M}(H))$.

Theorem 9 indicates that the search process can be finished in a node at any level without searching lower level nodes, if a given input matches a priority rule. Thus, the BMR for a packet H can be found without searching all the rules in $\mathcal{M}(H)$, which is a very important characteristic of the PAQT algorithm in improving the search speed.

Corollary 2. *Let v be a node with $L(v) = l$ and $m = P(\mathcal{S}(v))$. If m is greater than the smallest priority index among current matched rules ($m >$ MR), then MR $= P(\mathcal{M}(H))$, i.e., F_{MR} is the BMR in the given classifier $\mathcal{F} = \{F_1, F_2, \cdots, F_N\}$.*

Corollary 2 indicates that the search process finishes in a node at any level without searching lower levels, if the priority index of a node is greater than the priority index of the current match. Thus if an input packet matches a filter stored by a crossing filter, the index of the crossing filter becomes the current MR. If the MR is less than $P(\mathcal{S}(v))$ in a node v, then the search finishes at node v. The rules in node v do not need to be compared and the lower level nodes of v do not need to be searched. This also means that the BMR for a packet H can be found without searching all the rules in $\mathcal{M}(H)$, which is another important characteristic in improving the search performance.

4.4. Update Process

Easy incremental update is also important for a good packet classification algorithm [19]. The PAQT data structure provides simple incremental update of classification tables. First, consider inserting a new rule, $F_n = (F_n[1], \cdots, F_n[k])$. Based on the first two fields $(F_n[1], F_n[2])$ and the priority of the rule, a node v_n to store F_n is located.

Assume that $\mathcal{C}(A(v_n)) = \{F_{i_1}, \cdots, F_{i_n}\}$ and $\mathcal{S}(v_n) = \{F_m\} \cup \mathcal{C}(A(v_n))$. F_m is the rule stored by priority and $\mathcal{C}(A(v_n))$ is not empty if crossing filters exist. If the new rule F_n has a higher priority than F_m, then F_n should be stored in v_n and $\mathcal{S}(v_n)$ should be updated as $\mathcal{S}(A(v_n)) = \{F_n, F_m, F_{i_1}, \cdots, F_{i_n}\}$. In this case, two priority rules are stored in the node. The linked list of the node v_n is updated in this order. If the new rule is to be inserted as a crossing filter, it is inserted in the appropriate location in the linked list of v_n so that the priority order is maintained.

Now consider deleting a rule, F_d. From the first two fields and the priority of F_d, the node v_d is located. Assume that

$$\mathcal{S}(v_d) = \{F_{i_1}, \cdots, F_{i_{q-1}}, F_{i_q}, F_{i_{q+1}}, \cdots, F_{i_d}\}$$

where $F_{i_q} = F_d$. The rule F_d is deleted and $\mathcal{S}(v_d)$ is updated as $\mathcal{S}(v_d) = \{F_{i_q}, \cdots, F_{i_{q-1}}, F_{i_{q+1}}, \cdots, F_{i_d}\}$. The linked list of v_d is updated so that the priority order is maintained. If the node v_d (before the deletion of F_d) contains only one rule, then this node is empty after the deletion. Since this node still contains child pointers, it is maintained. The empty node v_d can be used for an insertion of a new rule later. For example, if there is a new rule to be inserted, and if the proper location of the new rule is one of the descendent nodes of the node v_d, then the new rule can be stored in node v_d since rules can be stored into a higher level node than their crossing levels. In this case, the new rule cannot be a priority rule since there might be other rules with a higher priority in lower level nodes, and hence it is marked as a crossing filter rule even though it is not. The new rule will be compared earlier than it should be, but that causes no problems. Since the insertion or the deletion of a rule affects only a single node in the proposed data structure, simple incremental update is achieved.

4.5. Scalability

There are two aspects of scalability: more fields and larger classifiers.

First, consider the scalability to more fields. The following definition can be used for field scalability (similar to [25]).

Definition 4 *Let $\mathcal{F}$ be a k-dimensional classifier where $F_i = (F_i[1], \cdots, F_i[k])$ $(i = 1, \cdots, N)$ is a rule consisting of k fields. One more field can be added to each rule in $\mathcal{F}$ such that $F_i^e = (F_i[1], \cdots, F_i[k], F_i[k+1])$ and $\mathcal{F}^e = \{F_1^e, \cdots, F_N^e\}$. The (k+1)-dimensional rule F_i^e is a (k+1)-dimensional extension of F_i and (k+1)-dimensional classifier $\mathcal{F}^e$ is a (k+1)-dimensional extension of $\mathcal{F}$.*

In the 2D PAQT algorithm, if the memory entry structure including the new field is extended, the build process and the search process can be performed in the same way. The build process is the same since the node location in a quad-tree is determined by the first two fields of the rule. The search process is also the same except there is an additional comparison for the new field in each visited memory address. Hence the algorithm is easily scaled to packet classification with more fields.

Second, consider the scalability to larger classifiers. For scalability to larger classifiers in terms of required memory size, the PAQT algorithm requires a memory size of $O(N)$ for N rules. Only the linear search, TCAM, and the tuple space search (with perfect hash function) schemes require a memory size of $O(N)$, which is the minimal memory size. Hence the algorithm is scalable to larger classifiers in terms of the memory size.

For scalability to larger classifiers in terms of search speed, even though the worst-case bound in the number of memory accesses of the PAQT algorithm is theoretically $O(N)$, the number of memory accesses of the algorithm increases only moderately for larger classifiers as will be shown in the simulation results in Section 5. Hence the algorithm is scalable to larger classifiers in terms of the search speed.

4.6. Implementation Example

Table 1 shows an example rule set which is composed of 8 rules (or filters), and each rule is composed of 5 fields; a source prefix, a destination prefix, a source port number, a destination port number, and a protocol type. Rules with a smaller priority index are assumed to have higher priorities. Fig. 1 shows the quad-tree built by AQT, where white nodes are empty and gray nodes contain rules. In Fig. 1, since rules are stored only by crossing filter condition, there are many empty nodes in the paths to the low level rule nodes. There are 10 empty nodes for 8 rules and the depth is 5. For real packet classification tables, the maximum level would be typically 32 since there could exist rules with both source and destination prefix lengths of 32.

Fig. 2 shows the quad-tree built by the PAQT algorithm, where v_r is the root node and v_1, v_2, v_3, and v_4 are four valid children of v_r. In Fig. 2, the first rule in each node is the rule stored by the priority in that node.

In this example, the initial classifier in the build process is

$$\mathcal{F} = \{F_1, F_2, \cdots, F_8\}.$$

For the root node v_r, the smallest priority index, the set of crossing filters, and the set of stored filters are $P(A(v_r) \cap \mathcal{F}) = 1, \mathcal{C}(A(v_r)) = \{F_2, F_7\}, \mathcal{S}(v_r) = \{F_1, F_2, F_7\}$, respectively. The set of filters for level 1 is updated as $\mathcal{F}_1 = \mathcal{F} - \mathcal{S}(v_r) = \{F_3, F_4, F_5, F_6, F_8\}$. For nodes v_0, v_1, v_2, v_3 at level 1, the smallest priority index, the set of crossing filters, and the set of stored filters are as follows. $P(A(v_0) \cap \mathcal{F}_1) = 4, \mathcal{C}(A(v_0)) = \emptyset, \mathcal{S}(v_0) = \{F_4\}, P(A(v_1) \cap \mathcal{F}_1) = 6, \mathcal{C}(A(v_1)) = \{F_6\}, \mathcal{S}(v_1) = \{F_6\}, P(A(v_2) \cap \mathcal{F}_1) = 8, \mathcal{C}(A(v_2)) = \emptyset, \mathcal{S}(v_2) = \{F_8\}, P(A(v_3) \cap \mathcal{F}_1) = 3, \mathcal{C}(A(v_3)) = \{F_5\}, \mathcal{S}(v_3) = \{F_3, F_5\}$. The set of filters for level 2 is updated as

$$\mathcal{F}_2 = \mathcal{F}_1 - \bigcup_{i=0}^{3} \mathcal{S}(v_i) = \emptyset.$$

Hence the build process is completed at level 1 in this case. From this example, it is clear that the quad-tree of the PAQT algorithm has no empty nodes and the tree depth is much smaller than that of the AQT.

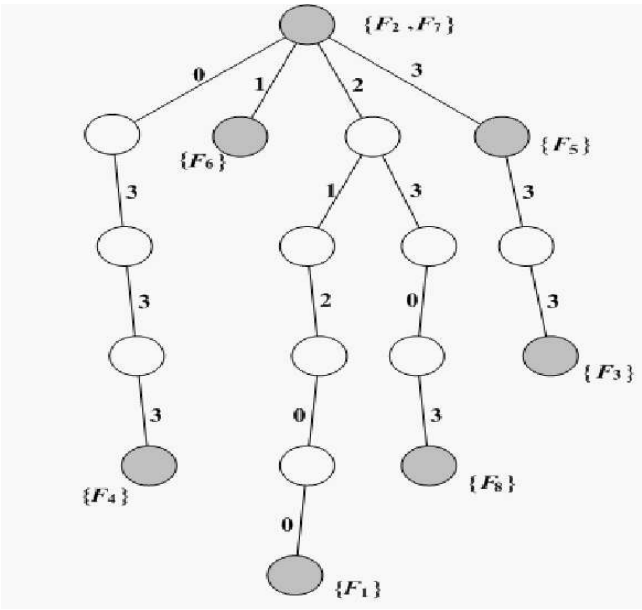

Figure 1 Quad-tree by the AQT.

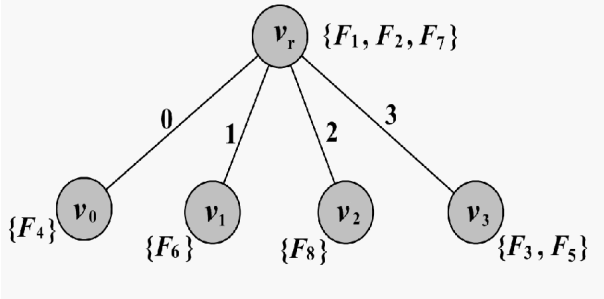

Figure 2 Quad-tree by the PAQT algorithm.

Table 1 An example classification table

Rule (Filter)	Src Prefix	Dst Prefix	Src Port	Dst Port	Protocol Type	Action
F_1	10100*	01000*	> 1024	> 1024	*	Permit
F_2	*	10101*	80	80	*	Permit
F_3	111*	11110*	*	> 1024	UDP	Deny
F_4	01110*	0111*	80	80	TCP	Permit
F_5	1*	10011*	21	21	TCP	Permit
F_6	0*	11100*	80	80	TCP	Permit
F_7	1100*	*	88	88	TCP	Permit
F_8	1101*	0101*	*	*	*	Deny

5. Simulation Results and Performance Evaluation

Class-Bench [26] provides rule sets and input traces similar to real classifiers used by Internet routers. Simulations have been performed using three different types of rule sets generated by class-bench: access control list (ACL), firewall (FW), and IP chain (IPC). Rule sets in the sizes of approximately 1000 and 5000 rules are used for the simulations.

Each type of classifier has different characteristics. When each type of classifier is built using AQT algorithm, the ACL type classifier has a relatively large number of empty nodes, and it has relatively few crossing filters associated with each node. On the other hand, the FW type classifier has fewer empty nodes and more crossing filters in a node. The IPC type classifier is in-between. In case of 5000-rule sets, the number of the empty nodes is 3214, 27, and 1055, and the maximum number of crossing filters in a node is 27, 231, and 146 for the ACL, FW, and IPC types, respectively. The different characteristics of classifiers affect the performance of packet classification algorithms in different ways.

Extensive simulations are performed for various packet classification algorithms including the linear search, hierarchical trie (H-trie) [13], set-pruning trie [13], grid-of-tries (GOT) [24], extended grid-of-tries (EGT) [9], bit-vector algorithm [16], and AQT [10] to compare with the PAQT algorithm. The performance of cutting algorithms such as HiCuts [14] and HyperCuts [22] depends on the number of rules for linear search in a leaf node of the decision tree and the number of cuts made in each dimension. Determining the number of cuts and the number of rules in a leaf highly depends on the heuristics related to the characteristics of rule distribution, and hence the simulation results could not be included in this paper.

Table 2 shows the required memory sizes for storing classifiers. The number of table entries is equal to the number of rules in the linear search, and hence the required memory size scales linearly. The linear search algorithm requires the smallest memory as expected. For the bit-vector algorithm [16], three separate memories are required, one for storing the trie structure, another for storing the bit-vectors, and a third for storing the rules. Depending on the number of rules in a classifier, entries for storing bit-vectors have $\lceil N/8 \rceil$ byte width. The width could be greater for large classifiers so that it is not possible to be read and written through one single memory access, and hence it may affect the search performance. This issue has been researched in [8].

For the PAQT algorithm, the entry width in the routing table can be implemented with 32 bytes. The AQT algorithm is implemented using two tables: a quad-tree table and a rule table. The width of the quad-tree table is 10 bytes and the rule table is the same as that in the linear search. The AQT can be implemented with a single table (like the PAQT algorithm), but it wastes memory space because of empty nodes included in the AQT quad-tree.

The PAQT algorithm is close to the linear search algorithm in terms of the required memory size. The number of entries of the PAQT algorithm is equal to the number of rules as the linear search, but the memory width of the PAQT algorithm is 11 bytes wider than that of the linear search algorithm because of additional pointers such as the child pointers and the linked list pointer. The AQT algorithm requires more memory than the PAQT algorithm because of the additional quad-tree table. The bit-vector algorithm requires the largest amount of memory. Since the constructed trie structure is highly dependent on the classifier characteristics, the required memory size varies in the hierarchical approach such as H-trie, set-pruning, GOT, and EGT. For example, the FW2 classifier requires a lot smaller memory space than ACL2 or IPC2 since prefix lengths are generally short in FW type classifiers and hence the number of empty nodes are small.

Table 3 shows the worst-case number of memory accesses in searching the BMR in each type of classifiers. The worst-case number of memory accesses for linear search is equal to the number of rules, and hence it is much larger than the other algorithms. The number of memory accesses depends on the number of nodes including empty nodes and the number of crossing filters in each node in the AQT algorithm and the PAQT algorithm. Since the ACL classifier has a relatively small number of crossing filters compared with other two types, it has better worst-case memory access performance. The FW type classifier has many

Table 2 Memory size comparison for ACL, FW, and IPC type classifiers

Type	ACL		FW		IPC	
Name	ACL1	ACL2	FW1	FW2	IPC1	IPC2
No. of Rules	958	4659	870	4343	988	4467
Linear search	20.1	97.9	17.8	91.2	20.5	101.2
H-trie [13]	192.8	950.9	65.3	127.7	296.1	416.1
Set-pruning [13]	258.6	1746.7	291.9	286.8	1165.4	896.8
GOT [24]	192.8	950.9	65.3	127.7	296.1	416.1
EGT [9]	210.5	1036.9	70.9	136.0	323.5	453.6
Bit-vector [16]	153.3	2793.2	111.9	2394.6	154.3	2531.5
AQT [10]	56.4	200.2	35.3	479.8	71.2	234.3
PAQT	29.9	145.6	27.2	136.0	30.9	139.9

Table 3 Worst-case number of memory accesses for ACL, FW, and IPC type classifiers

Type	ACL		FW		IPC	
Name	ACL1	ACL2	FW1	FW2	IPC1	IPC2
No. of Rules	958	4659	870	4343	988	4467
Linear search	958	4659	870	4343	988	4467
H-trie [13]	104	158	104	134	122	181
Set-pruning [13]	90	102	77	104	69	87
GOT [24]	90	102	75	104	69	82
EGT [9]	94	109	104	134	104	176
Bit-vector [16]	68	76	318	1044	80	230
AQT [10]	64	94	444	1193	119	415
PAQT	75	113	293	999	106	295

Table 4 Average number of memory accesses for ACL, FW, and IPC type classifiers

Type	ACL		FW		IPC	
Name	ACL1	ACL2	FW1	FW2	IPC1	IPC2
No. of Rules	958	4659	870	4343	988	4467
Linear search	407.8	2399.0	447.1	2292.3	569.1	1957.2
H-trie [13]	74.3	80.1	46.7	63.1	70.4	80.2
Set-pruning [13]	64.8	64.6	36.5	50.2	49.5	56.5
GOT [24]	64.8	64.3	35.2	48.9	49.5	53.0
EGT [9]	68.9	70.4	45.9	62.8	59.7	65.9
Bit-vector [16]	66.0	64.1	196.6	738.8	63.6	151.9
AQT [10]	38.6	50.1	369.3	660.5	94.5	344.8
PAQT	35.6	59.6	197.9	571.1	73.6	202.1

crossing filter rules associated with a node, and hence the worst-case search performance is the worst among the three types of classifiers. For the ACL classifiers, the AQT algorithm shows better performance than the PAQT algorithm. If a given input matches only a crossing filter in a low level node, the AQT algorithm has better worst-case search performance than the PAQT algorithm. The PAQT algorithm has better worst-case search performance for FW and IPC types. The worst-case number of memory accesses of the PAQT algorithm can be reduced by adopting a 1-dimensional version [17] instead of the simple linear search for nodes with many crossing filters.

It should be noted that the performance of the bit-vector algorithm could be worse than the result that is shown here for large classifiers since the width of memory is too large to be read and written through a single memory access. If the maximum memory width is 32 bytes, the number of memory accesses for the bit-vector algorithm should be multiplied by a factor of 4 or 20 for 1000-rule and 5000-rule classifiers, respectively. Algorithms in hierarchical approach show similar worst-case search performance.

Table 4 shows the average number of memory accesses in searching the BMR. The linear search algorithm has the worst performance as the size of classifier grows. The PAQT algorithm shows better performance than AQT al-

gorithm for all types of classifiers except the ACL2 in terms of the average number of memory accesses. For ACL type classifiers, the PAQT algorithm and the AQT algorithm show the best average search performance among all algorithms. The hierarchical approach algorithms consistently show very good average search performance for all types of classifiers.

From Table 3 and Table 4, the scalability issue in worst-case and average search performance can be discussed. For the ACL type classifiers, since rules are evenly distributed to nodes and not many crossing filters are associated with each node, the number of memory accesses is almost flat even though the size of classifier grows. On the other hand, for the FW type classifier, since many crossing filters are associated with each node, the number of memory accesses rapidly increases as the size of the classifiers grows, but it is still a lot better than the worst-case bound of $O(N)$. The IPC type classifier shows scalability in the middle among the three types of classifiers.

6. Conclusion

A novel mathematical framework for packet classification is proposed in this paper. We present the priority area-based quad-tree (PAQT) algorithm including data structure, build process, search process, and update process using the mathematical framework. The validity of build and search process is proved mathematically to provide the theoretical justification of the PAQT algorithm. We belive that the new mathematical framework can also be applied for the justification of other packet classification algorithms.

Compared with the previous recursive space decomposition algorithm, the highest priority rule that is included in the search space is compared first, even if it does not satisfy the crossing filter condition. In this way, empty nodes are completely removed, and hence the required memory size scales linearly with the number of rules and the search speed is improved. The PAQT algorithm also provides incremental update by limiting the change to a single node when inserting or deleting a rule.

Extensive simulation results show that the PAQT algorithm is attractive in terms of search speed, memory size, and scalability. The PAQT algorithm is formally justified using the mathematical framework, while the simulation results show the competitiveness compared with previous packet classification algorithms. Hence we conclude that the PAQT packet classification algorithm can be used effectively for providing high quality of service in next-generation networks.

Acknowledgement

The research of the first author (H. Lim) was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (2012-005945). This research was also supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC support program supervised by the NIPA (NIPA-2012-H0301-12-4004).

The research of the second author (C. Yim) was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0009426).

References

[1] P. Wang, C. Chan, S. Hu, C. Lee, and W. Tseng, High-speed packet classification for differentiated services in next generation networks, *IEEE Trans. Multimedia*, **6(6)**: 925–935 (2004).

[2] Y. Zhang, Y.-F. Pu, J.-R. Hu, and J.-L. Zhou, A class of fractional-order variational image inpainting models, *Appl. Math. Inf. Sci.* **6(2)**: 299–306 (2012).

[3] Q. Lai, Numerical calculation of rate-distortion function of information source, *Appl. Math. Inf. Sci.* **6(1)**: 113–116 (2012).

[4] C.S. Withers and S. Nadarajah, Channel capacity for MIMO systems with multiple frequencies and delay spread, *Appl. Math. Inf. Sci.* **5(3)**: 460–483 (2011).

[5] F. Baboescu, P. Warkhede, S. Suri and G. Varghese, Fast packet classification for two-dimensional conflict-free filters, *Computer Networks*, **50(11)**: 1831–1842 (2006).

[6] H. J. Chao, Next generation routers, *Proceedings of the IEEE*, **90(9)**: 1518–1558 (2002).

[7] P.-C. Wang, Scalable packet classification with controlled cross-producing, *Computer Networks*, **53(6)**: 821–834 (2009).

[8] F. Baboescu and G. Varghese, Scalable packet classification, *IEEE/ACM Trans. Networking*, **13(1)**: 2–14 (2005).

[9] F. Baboescu, S. Singh, and G. Varghese, Packet classification for core router: is there an alternative to CAMs?, *Proc. IEEE INFOCOM*, **1**: 53–63 (2003).

[10] M.M. Buddhikot, S. Suri, and M. Waldvogel, Space decomposition techniques for fast layer-4 switching, *Proc. of Protocols for Speed Networks*, 25–41 (1999).

[11] Y. Chang, A 2-level TCAM architecture for ranges, *IEEE Trans. Computers*, **55(12)**: 1614–1629 (2006).

[12] F. Geraci, M. Pellegrini, P. Pisata, and L. Rizzo, Packet classification via improved space decomposition technique, *Proc. IEEE INFOCOM*, **1**: 1304–312 (2005).

[13] P. Gupta and N. Mckeown, Algorithms for packet classification, *IEEE Network*, **15(2)**: 24–32 (2001).

[14] P. Gupta and N. Mckeown, Classification using hierarchical intelligent cuttings, *IEEE Micro*, **20(1)**: 34–41 (2000).

[15] Y. Jung and H. Lim, A two-dimensional binary prefix tree for packet classification, *Proc. IEEE HPSR*, (2005).

[16] T.V. Lakshman and D. Stildialis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, *Proc. ACM SIGCOMM*, (1998).

[17] H. Lim and J. Mun, An efficient IP address lookup algorithm using a priority-trie, *Proc. IEEE Globecom-2006*, 1–5 (2006).

[18] H. Lim, M.Y. Kang, and C. Yim, Two-dimensional packet classification algorithm using a quad-tree, *Computer Communications*, **30**: 1396–1405 (2007).
[19] C. Macian and R. Finthammer, An evaluation of the key design criteria to achieve high update rates in packet classifiers, *IEEE Network*, 24–29 (2001).
[20] W. Lu and S. Sahni, Packet classification using space-efficient pipelined multibit tries, *IEEE Trans. Computers*, **57(5)**: 591–605 (2008).
[21] J. Lunteren and T. Engberson, Fast and scalable packet classification, *IEEE Journal of Selected Areas in Communications*, **21(4)**: 560–571 (2003).
[22] S. Singh, F. Baboescu, G. Varghese, and J. Wang, Packet classification using multidimensional cutting, *Proc. SIGCOMM*, 213–224 (2003).
[23] V. Srinivasan, S. Suri, and G. Varghese, Packet classification using tuple space search, *Proc. ACM SIGCOMM*, 135–146 (1999).
[24] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, Fast and scalable layer four switching, *Proc. ACM SIGCOMM*, 191–202 (1998).
[25] X. Sun, S.K. Sahni, and Y.Q. Zhao, Packet classification consuming small amount of memory, *IEEE/ACM Trans. Networking*, **13(5)**: 1135-1145 (2005).
[26] D.E. Taylor and J.S. Turner, ClassBench: a packet classification benchmark, *Proc. IEEE INFOCOM*, **3**: 2068–2079 (2005).
[27] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent policy table update algorithm for TCAM without locking, *IEEE Trans. Computers*, **53(12)**: 1602–1614 (2004).
[28] F. Yu, R.H. Katz, and T.V. Lakshman, Efficient multimatch packet classification and lookup with TCAM, *IEEE Micro*, **1**: 50-59 (2005).
[29] F. Yu, T.V. Lakshman, M.A. Motoyama, and R.H. Katz, Efficient multimatch packet classification for network security applications, *IEEE Journal of Selected Areas in Communications*, **24(10)**: 1805–1816 (2006).
[30] K. Zheng, H. Che, Z. Wang, B. Liu, and X. Zhang, DPPC-RE: TCAM-based distributed parallel packet classification with range encoding, *IEEE Trans. Computers*, **55(8)**: 947–961 (2006).

Hyesook Lim received the B.S. and the M.S. degrees at the Department of Control and Instrumentation Engineering in Seoul National University, Seoul, Korea, in 1986 and 1991, respectively. She got the Ph.D. degree at the Electrical and Computer Engineering from the University of Texas at Austin, Texas, in 1996. From 1996 to 2000, she had been employed as a member of technical staff at Bell Labs in Lucent Technologies, Murray Hill, NJ. From 2000 to 2002, she had worked as a hardware engineer for Cisco Systems, San Jose, CA. She is currently a professor in the Department of Electronics Engineering, Ewha Womans University, Seoul, Korea, where she does research on router design issues such as IP address lookup, packet classification, and deep packet inspection and on hardware implementation of various network protocols such as TCP/IP and Mobile IPv6.

Changhoon Yim received the B.Eng. degree from the Department of Control and Instrumentation Engineering, Seoul National University, Korea, in 1986, the M.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology, Korea, in 1988, and the Ph.D. degree in Electrical and Computer Engineering from the University of Texas at Austin, USA, in 1996. He was a research engineer working on HDTV at Korean Broadcasting System, Korea, from 1988 to 1991. From 1996 to 1999, he was a member of technical staff in HDTV and Multimedia Division, Sarnoff Corporation, NJ, USA. From 1999 to 2000, he worked at Bell Labs, Lucent Technologies, NJ, USA. From 2000 to 2002, he was a software engineer in KLA-Tencor Corporation, CA, USA. From 2002 to 2003, he was a principal engineer in Samsung Electronics, Suwon, Korea. He is currently a professor in the Department of Internet and Multimedia Engineering, Konkuk University, Seoul, Korea. His research interests include digital image processing, video compression, multimedia network, and multimedia communication.

Earl E. Swartzlander, Jr. received the B.S. degree from Purdue University in 1967, the M.S. degree from the University of Colorado in 1969, and the Ph.D. degree from the University of Southern California in 1972, all in electrical engineering. He is a professor of electrical and computer engineering at the University of Texas at Austin. In his current position, he and his students conduct research in computer engineering with emphasis on application-specific processor design, including high-speed computer arithmetic, embedded processor architecture, VLSI technology, and nanotechnology. As of May 2012, he has supervised 35 Ph.D. students. From 1975 to 1990, he held a variety of positions at TRW including the director of Independent Research and Development in the TRW Defense Systems Group, the manager of the Digital Processing Laboratory in the Electronics and Technology Division, and the manager of the Advanced Development Office in the System Development Division. He was the editor-in-chief of the *IEEE Transactions on Computers* from 1990 to 1994 and was the founding editor-in-chief of the Journal of VLSI Signal Processing. In addition, he has served as an associate editor for the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, and the *IEEE Journal of Solid-State Circuits*. He has been a member of the Board of Governors of the IEEE Computer Society (1987-1991), the IEEE Signal Processing Society (1992-1994), and the IEEE Solid-State Circuits Council/Society (1986-1991). He has been a member of the IEEE History Committee (1996-2004), the IEEE Fellows Committee (2000-2003), and the IEEE James H. Mulligan, Jr., Education Medal Committee (2006-2011). He has chaired a number of conferences. He is the author of one book, editor of 8 books and the author or coauthor of 70 refereed journal papers, 33 book chapters, and 279 conference papers. He is a fellow of the IEEE. He has been honored with the IEEE Third Millennium Medal, the Distinguished Engineering Alumnus Award from the University of Colorado, the Outstanding Electrical Engineer and Distinguished Engineering Alumnus Awards from Purdue University, and the IEEE Computer Society Golden Core Award.