Efficient Multidimensional Packet Classification with Fast Updates

Yeim-Kuan Chang, *Member*, *IEEE Computer Society*

Abstract—Packet classification has continued to be an important research topic for high-speed routers in recent years. In this paper, we propose a new packet classification scheme based on the binary range and prefix searches. The basic data structure of the proposed packet classification scheme for multidimensional rule tables is a hierarchical list of sorted ranges and prefixes that allows the binary search to be performed on the list at each level to find the best matched rule. We also propose a set of heuristics to further improve the performance of the proposed algorithm. We test our schemes by using rule tables of various sizes generated by ClassBench and compare them with the existing schemes, EGT, EGT-PC, and HyperCuts. The performance results show that in a test using a 2D segmentation table, the proposed scheme not only performs better than the EGT, EGT-PC, and HyperCuts in classification speed and memory usage but also achieves faster table update operations that are not supported in the existing schemes.

Index Terms—Packet classification, rule table partitioning, binary search.

1 Introduction

Traditionally, Internet routers only provide the best effort service by processing each incoming packet in the same manner (i.e., forwarding packets based only on their destination addresses). Today, however, packet classifiers in routers have to compare multiple header fields of each incoming packet against a set of filters or rules in order to classify the packets into different flows. The classified flows can then be used by many emerging layer-4 switching technologies to provide quality-of-service guarantees to packets belonging to the specified flow. The most common header processing performed in a packet classifier is the examination of the standard 5-tuple fields: source and destination IP addresses, source and destination ports, and transport protocol number. A typical packet classifier is configured with a rule set consisting of a set of predefined 5-tuple filters. The packet classifier searches for the highest priority filter or set of filters matching the headers of the incoming packets in the rule table. To meet the demands for a fast packet classification in current speed-growing Internet, routers have to maintain an efficient data structure for a set of predefined rules.

In this paper, we propose a set of packet classification algorithms that are a generalization of 1D binary range and prefix searches [3]. The basic data structure of our multidimensional packet classification scheme is a hierarchical structure of expansion lists used by the binary range and prefix searches. Although this basic hierarchical structure is fast, its two drawbacks are large memory consumption and slow update process. We develop a set of optimizations to overcome these drawbacks. The most important optimization is the use of a multidimensional segmentation table to partition the set of original rules into smaller subsets, each of which is then organized by the basic hierarchical structure of the expansion lists. The proposed algorithms augmented with these optimizations are space and time efficient and can provide faster rule table updates. Our experiments show that the proposed scheme using a 2D segmentation table is better than the existing schemes, EGT, EGT-PC [1], and HyperCuts [19], in terms of search speed, memory usage, and update speed. The proposed scheme using a d-dimensional segmentation table is similar to HyperCuts and Common-Branches trees [5] in that multiple dimensions are used at the same time to partition the set of rules into smaller subsets. The mechanism to reduce the rule duplications in the proposed scheme is richer than that of HyperCuts and Common-Branches trees. Their differences are summarized as follows:

1. The rules in each subset partitioned by the d-dimensional segmentation table is organized as a hierarchical structure of expansion lists used by the binary range and prefix searches, while the same cutting technique based on some dynamic criteria in HyperCuts is applied recursively at each node to form the multilevel decision tree. Although our partitioning scheme is based on a predetermined criterion, a fast update speed can be obtained.
2. If four levels of expansion lists that used the binary and prefix searches are built for each subset partitioned by a d-dimensional segmentation table, the linear search at the leaf node of the hierarchical expansion list is only operated on the final field (i.e., protocol number). However, after reaching the leaf node of the decision tree in HyperCuts, the linear search must be operated on all the five fields, which is slow.
3. We provide a mechanism to control the degree of rule duplication which is richer than that of the Common-Branches trees [5]. If we choose the scheme that uses the least degree of rule duplication, no rule will be replicated among the rule subsets partitioned by the segmentation table. In HyperCuts, the rule

• *The author is with the Department of Computer Science and Information Engineering (CSIE), National Cheng Kung University, No. 1, Ta-Hsueh Road, Tainan 701, Taiwan, R.O.C. E-mail: ykchang@mail.ncku.edu.tw.*

Manuscript received 17 July 2007; revised 3 Jan. 2008; accepted 26 Feb. 2008; published online 17 Sept. 2008.
Recommended for acceptance by X. Zhang.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-07-0322.
Digital Object Identifier no. 10.1109/TC.2008.181.

duplication is its inherent property that will result in larger memory consumption.

This paper is organized as follows: Section 2 briefly summarizes the related works. The problem statement and notations are given in Section 3. In Section 4, we describe the 1D binary prefix and range searches which are used as our basic search structures. Many optimizations are also proposed. Section 5 presents the proposed multidimensional packet classification and its optimizations. Section 6 presents the performance results in terms of search speed, memory requirement, and update speed. Finally, conclusions are made.

2 Related Works

The process that classifies packets into flows in Internet routers is called packet classification. Specifically, packet classification algorithm searches a set of predefined rules against one or more fields in the packet headers to find all the matched rules or the best one among them.

The simplest classification algorithm is a linear search that performs the header matching against the rules in a one-by-one fashion. For a large number of rules, this approach implies a long query time, but it is very efficient in terms of memory and rule updates. To improve the performance of query times, a 2D algorithm called *Grid of Tries* (GoT) was proposed [21]. Since GoT cannot be easily extended to more than two fields, they also proposed a better-generalized scheme called *Cross-Producting* [21]. Unfortunately, the size of this table grows astronomically with the number of rules. Baboescu et al. [1] proposed an extended version of Grid of Tries (EGT). They also proposed an improved version of EGT called EGT-PC (Path Compression) which is a standard compression scheme for tries that function by removing single branching paths. In addition to trie-based hierarchical schemes such as GoT, many hierarchical decision tree based schemes have been developed based on different data structures. In HiCuts [11], a precomputed decision tree is built as follows: Suppose a node T in the decision tree is associated with a set of rules. The set of rules in T is cut into many smaller subsets that in turn are associated with the child nodes of T. The cutting process is repeated at each child node until the number of rules associated with the node is not more than a threshold. The search process is done by traversing the decision tree to identify the matching rule that is always located in a leaf node. Each leaf node stores a small number of rules that are linearly searched. A modular scheme similar to HiCuts is developed in [23], where an index jump table is used to create multiple decision trees to save memory space, and the range tests in HiCuts are replaced by the bit tests. In [19], an improved version of HiCuts called HyperCuts is proposed. Instead of selecting one dimension at a time, HyperCuts selects multiple dimensions at the same time to cut the rules associated with a node into many smaller rule subsets. As a result, the depth of the decision tree will be smaller than that of HiCuts. In [5], an enhanced decision tree based on the concept of common-branches is proposed. Another hierarchical structure called the Fat Inverted Segment (FIS) tree was proposed in [8].

Tuple-Space Search proposed in [20] partitions the rules of a classifier into different tuple categories based on the number of specified bits in the rules. The scheme then uses hashing among rules within the same category. The main disadvantage of this scheme is the use of hashing that makes the time complexity of searches and updates nondeterministic. Lakshman and Stiliadis proposed a bitmap-intersection scheme [14]. This scheme first searches each dimension separately to yield the set of rules that matches the packet in that particular dimension. The search algorithm can be a binary trie or any 1D IP lookup scheme. These sets are then intersected by using bitmaps to yield the set of rules that match in all dimensions. One drawback of this scheme is that it needs parallel hardware assistance and is impractical for large classifiers due to its large memory consumption. A modified version of the bitmap-intersection scheme is presented in [2]. Gupta and McKeown proposed a scheme called *Recursive Flow Classification* (RFC) [10]. This scheme is very fast but it requires precomputation, a large amount of memory, and parallel hardware support.

The related works described above are in no way a complete list in the literature. Readers interested in other schemes can refer to the two survey papers in [23] and [12]. The most recent paper using a hardware-based smart rule cache for fast searches can also be found in [6].

3 Problem Statement and Notations

The notations needed in this paper are first summarized in Table 1. The process of packet classification (PC) classifies packets into flows. A predefined *rule* is used to identify a flow. All rules defined in a router are collected as a *classifier.* Therefore, the process of the packet classification is to search a classifier against one or more fields of the packet header to find all the matched rules or the least cost one among them if each rule is also associated with a cost. For a d-dimensional packet classification, each rule R in the classifier consists of d components, $R = [F_1, \ldots, F_d]$, where $F_i = [L_i, U_i]$ is a range of values from L_i to U_i for i $= 1$ to d. The search for an incoming packet p in the classifier is done by presenting the header fields $[f_1, \ldots, f_d]$ of p as the keys, where each f_i is a singleton value. The rule R is said to match packet p, if for all dimensions i, the field value f_i of packet p lies in the range F_i. The PC problem is to determine the least cost rule that matches the packet. For example, the layer-four switching of the Internet protocol studied in this paper consists of five dimensions: the source address, destination address, source port, destination port, and protocol number. Table 2 shows an example of 5D real-life classifier in which by convention, the first rule R1 has the highest priority and the last rule R4 has the lowest priority. Table 3 illustrates the classification results for three incoming packets.

4 One-Dimensional Packet Classification

We give the formal definition of 1D packet classification as follows: Given a set of n rules, $\mathcal{G} = \{G_i = (R_i, C_i) | R_i = [L_i, U_i]$ is a W-bit range and C_i is the priority of G_i for $i = 0, \ldots, n-1\}$, the process of 1D packet classification for a address p is to find the range $G_h \in \mathcal{G}$, such that R_h contains p and C_h is the highest. The longest prefix matching (LPM) of the IP lookup problem is a special case of the 1D packet classification, where the range of the values specified in the only dimension follows the prefix format and longer prefixes get the higher priorities. The IP lookup problem has been studied extensively in the literature [17]. However, if range R_i

TABLE 1
Notations

W	The (maximum) number of bits in the address space of prefixes or ranges.
Range	A W-bit range $[L, U]$ satisfies that $0 \le L \le 2^W-1$, $0 \le U \le 2^W-1$, and $L \le U$.
Min(R), Max(R)	The lowest and highest addresses of $R = [L, U]$, i.e., $Min(R) = L$ and $Max(R) = U$.
Prefix	A W-bit prefix T of length i is represented in the ternary format as $t_{W-1}\ldots t_{W-i}*\ldots*$ or $t_{W-1}\ldots t_{W-i}*$, where $t_j = 0$ or 1 for $W-1 \ge j \ge W-i$. The prefix can also be represented as the range $[t_{W-1}\ldots t_{W-i}0\ldots0, t_{W-1}\ldots t_{W-i}1\ldots1]$.
LCA	The *Longest Common Ancestor* of two prefixes $A = a_{W-1}\ldots a_0$ and $B = b_{W-1}\ldots b_0$ is $LCA(A, B) = c_{W-1}\ldots c_i*$, where $c_k = a_k = b_k$ and $c_k \ne *$ for $W-1 \ge k \ge i$ and $a_{i-1} \ne b_{i-1}$.
Elementary intervals	Let the set of k elementary intervals constructed from a set $\mathcal{R}$ of W-bit ranges be $\mathcal{X} = \{X_i \mid X_i = [e_i, f_i]$, for $i = 1$ to $k\}$. $\mathcal{X}$ must satisfy the following: (I) $e_1 = 0$ and $f_k = 2^W-1$, (II) $f_i = e_{i+1}-1$ for $i = 1$ to $k-1$, (III) all addresses in X_i are covered by the same subset of $\mathcal{R}$ (called the range matching set of X_i) denoted by EI_i, and (IV) $EI_i \ne EI_{i+1}$, for $i = 1$ to $k-1$. For example, given a 4-bit range set $\mathcal{R} = \{[3,12], [6,9]\}$, the set of elementary intervals is $\{[0,2], [3,5], [6,9], [10,12], [13,15]\}$.

TABLE 2
A Real-Life Classifier in Five Dimensions

Rule	Network-layer		Transport-layer			Action
	Dst. (address/length)	Src. (address/length)	Dst port	Src. port	Protocol #	
R1	140.116.246.69/32	140.116.82.11/32	*	*	*	Deny
R2	140.126.53.0/24	140.116.100.157/32	eq www(80)	*	tcp	Deny
R3	140.116.247.4/32	140.116.160.0/24	gt 1023	*	udp	Permit
R4	0.0.0.0/0	0.0.0.0/0	*	*	*	Permit

TABLE 3
Classification Examples

Pkt	Network-layer		Transport-layer			Best matching rule, Action
	Dst. address	Src. address	Dest. port	Src. port	Protocol #	
P1	140.116.246.69	140.116.82.11	www	1222	tcp	R1, Deny
P2	140.126.53.10	140.116.100.157	www	1233	udp	R2, Deny
P3	140.116.247.4	140.116.160.10	1024	1235	tcp	R3, Permit

is arbitrary, the static or dynamic versions of segment trees or interval trees [4], [16], [18] may be the better solution.

We know that algorithms designed to achieve better update speeds usually compromise search speed and consume more memory. Large memory consumption usually results from the fact that these algorithms must be implemented with pointers. On the contrary, the proposed algorithms employ the linear lists (arrays) as the primary search structures for both prefixes and ranges. For prefixes, we use the *binary prefix search (BPS)* algorithm proposed in [3] that uses the binary search on a sorted list of the original prefixes and a small number of auxiliary prefixes. For ranges, we design an algorithm called *binary range search (BRS_Int)* that uses a new endpoint definition for ranges. The enclosure relationship is removed by the concept of elementary intervals. Both the proposed algorithms for prefixes and ranges will be optimized by a number of heuristics.

4.1 Proposed Binary Search for Prefixes

Our main idea in the proposed *BPS* is based on the comparison of prefixes of different lengths [3] which are defined as follows:

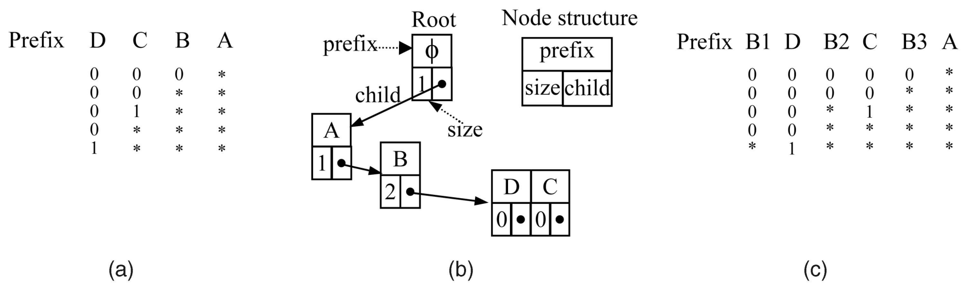
Definition 1. *The inequality* $0 < * < 1$ *is used to compare two prefixes in ternary format.*

For example, we have $A < B < C$ for 8-bit prefixes, $A = 0000{-}0***$, $B = 0000{-}0100$, and $C = 10**{-}****$. In order to have an efficient implementation to compare prefixes of different lengths, we propose a new prefix representation of length $(W+1)$ bits in [19] as follows:

Definition 2. *For a prefix of length* i, $b_{W-1}, \ldots, b_{W-i^*}$, *where* $b_j = 0$ *or* 1 *for* $W-1 \ge j \ge W-i$, *its binary representation is* $b_{W-1}, \ldots, b_{W-i}10, \ldots, 0$ *with* $W-i$ *trailing zeros.*

For example, prefix $0000{-}0***$ can be represented as 0000-0100-0. The $(W+1)$-bit binary prefix representation is shorter than $W + \log W$ bits for the *length format* of *IP/len* or $2W$ bits for the *mask format* of *IP/mask*. As a result, a $(W+1)$-bit binary comparison operation is sufficient to compare two prefixes of different lengths. With this definition, a very fast binary search can be applied on a linear list of sorted disjoint prefixes. However, the prefix enclosure defined as follows makes the binary search complicated.

Definition 3. Prefix enclosure. *A prefix is said to be enclosed or covered by another prefix if the address space covered by the former is a subset of that covered by the latter. We use* $B \supseteq A$ *or* $A \subseteq B$ *to denote that prefix* B *encloses prefix* A, *where* $\supseteq$ *or* $\subseteq$ *is the enclosure operator. If neither* A *encloses* B *nor* B *encloses* A, *we say that* A *and* B *are* disjoint.



Prefix	D	C	B	A
	0	0	0	*
	0	0	*	*
	0	1	*	*
	0	*	*	*
	1	*	*	*

Prefix	B1	D	B2	C	B3	A
	0	0	0	0	0	*
	0	0	0	0	*	*
	0	0	*	1	*	*
	0	0	*	*	*	*
	*	1	*	*	*	*

(a) (b) (c)

Fig. 1. The proposed data structures for a BPS. (a) The sorted list of original prefixes. (b) The hierarchical list. (c) The expansion list.

```
Expansion_traversal(node Root) // cnt is set to 0 initially.
Begin
01  If (Root.size = 0) Then ExpList[cnt++].prefix = Root.prefix;
02  Else
03      If ( (Root.prefix ≠ ϕ) and Min(Root.prefix) ≠ Min(Root.child[1])) Then
04          ExpList[cnt++].prefix = LCA(Min(Root.prefix), Root.child[1]);
05      For (i = 1 to Root.size) Do
06          Expansion_traversal(Root.child[i])
07          If ( (i ≠ Root.size) and Max(Root.child[i]) ≠ Min(Root.child[i+1]) – 1) Then
08              ExpList[cnt++].prefix = LCA(Root.child[i]), Root.child[i+1]);
09      End-Do
10      If ( (Root.prefix ≠ ϕ) and Max(Root.child[Root.size]) ≠ Max(Root. prefix)) Then
11          ExpList[cnt++].prefix = LCA(Root.child[Root.size], Max(Root. prefix));
End
```

Fig. 2. Constructing the expansion list stored in array *ExpList[]* from the hierarchical list.

By keeping a linear list of prefixes sorted, we cannot guarantee finding the correct LPM if we perform the binary search on the list. Consider the binary search operation for address $Dst = 00010$ on the sorted list of four 5-bit prefixes in Fig. 1a. The first prefix to be compared is the middle one $C = 001^{**}$. Prefix C does not match Dst and the search continues. Because Dst is smaller than C, prefix D is compared. Prefix D does not match address Dst either. Obviously, the final LPM should be B. Prefix B did not get any chance to be examined in the process of the binary search. To solve the problem, we generate some *auxiliary* prefixes that inherit the routing information of the original LPM (e.g., B) and place them at the locations where the binary search operations can find them. For example, if we insert an auxiliary prefix 00^{***} inheriting B's routing information, then the search operation for address 00010 will succeed. Therefore, the feasible way is to generate two auxiliary prefixes from B that cover both sides of prefix C.

We use a two-step method to construct the *expansion list* consisting of the sorted original and auxiliary prefixes. In the first step, the hierarchical list is constructed from the original prefixes such that the address space covered by the parent node contains that covered by its children. Fig. 1b illustrates the hierarchical list constructed from the prefixes in Fig. 1a. In the hierarchical list, the node structure contains three fields: the *prefix* which is an original prefix, the *size* which records the number of prefixes in the next level pointed to by the current prefix, and the *child* which is the pointer pointing to the next level. The prefixes in one level of the hierarchical list are disjoint. The hierarchical list used for obtaining the expansion list is more efficient than the binary trie used in [3]. The expansion list is obtained by a simple in order traversal called *Expansion_traversal*() as shown in Fig. 2. Auxiliary prefixes are generated recursively if the condition in lines 3, 7, or 10 of Fig. 2 is true.

For the example shown in Fig. 1, prefix B encloses D and C. Therefore, three auxiliary prefixes—$B1$, $B2$, and $B3$—are generated from prefix B, such that $B1 < D < B2 < C < B3$. $B1$ is $LCA(Min(B), D)$, which is the longest common ancestor of the lowest address of prefix B and prefix D. Similarly, $B2$ is $LCA(D, C)$ and $B3$ is $LCA(C, Max(B))$. The final result is shown in Fig. 1c. Notice that one of the auxiliary prefixes may be the same as the original prefix that generates them. As in the above example, $B3$ is the same as B.

It can be seen that k disjoint prefixes covered by an enclosure prefix will result in at most $2k + 1$ prefixes in the expansion list. Therefore, it is not difficult to show that the worst-case number of prefixes in the expansion list is less than $2N$ for a routing table of N prefixes (see [3]). The binary search algorithm [3] on the expansion list is illustrated in Fig. 3. When the destination IP address (*Dst*) is found to be located between two adjacent prefixes (say $P1$ and $P2$), that is, $P1 \le Dst \le P2$, additional operations are needed to check if either $P1$ or $P2$ matches the destination IP address, and if both prefixes match the destination address, then the LPM is the longer one, as shown in lines 2-7 of Fig. 3.

Discussions. The binary range search (*BRS*) proposed in [15] is another data structure that can store the original prefixes in an array. We will show that *BPS* performs better than *BRS*. The differences between *BPS* and *BRS* are described as follows: First, each prefix is identified as an individual entity instead of as the two endpoints for a range in *BRS*. Moreover, only one port information is associated with each prefix in *BPS* while two ports such as "$>$" and "$=$" ports are associated with each endpoint in *BRS*. Second, unlike *BRS*, the interval not covered by any original prefix will not be stored in the expansion list of *BPS*.

```
BPS_search(Element List[], Index L, Index R, Address Dst)
Begin
// List[] is an array storing the expansion list of sorted prefixes.
// L and R are the left and right indices of the List array and Dst is the target IP address.
// ⊒ is the enclosure operator
01    If (L+1 = R) Then
02       If (length(List[L].prefix) ≥ length(List[R].prefix)) Then
03           If (List[L].prefix ⊒ Dst) Then return L;
04           If (List[R].prefix ⊒ Dst) Then return R;
05       Else
06           If (List[R].prefix ⊒ Dst) Then return R;
07           If (List[L].prefix ⊒ Dst) Then return L;
08       return –1;
09   M = ceiling( (L+R)/2 );
10   If (List[M].prefix = Dst) Then return M;
11   If (List[M].prefix < Dst) Then return BPS_search(List, M, R, Dst);
12   return BPS_search(List, L, M, Dst);
End
```

Fig. 3. The proposed BPS which returns the index of the matched element in the array list. If no match is found, -1 is returned.

TABLE 4
The Average Performance of the BPS and BRS [14], Where N_{BRS} and N_{BPS} Are the Number of Elements in BRS and BPS, Respectively

Table	Funet-2000-4: 41,709 prefixes				AS6447-2002-4: 120,629 prefixes			
	N_{BRS} or N_{BPS}	Memory (KB)	Search time (μs)	Update time (μs)	N_{BRS} or N_{BPS}	Memory (KB)	Search time (μs)	Update time (μs)
BRS	63,598	372	0.21	8,500	232,887	1,364	0.33	21,600
BPS	48,587	237	0.16	5,200	145,737	712	0.26	16,300
BRS-16	58,623	526	0.11	6.4	217,146	1,104	0.18	7.45
BPS-16	40,329	396	0.10	3.6	117,968	601	0.13	5.25

Therefore, the size of the expansion list in *BPS* will be smaller than the size of the endpoint list in *BRS*.

The first two rows of Table 4 show the average performance results of BPS and BRS for two typical routing tables. BPS-16 and BRS-16 will be discussed later in Section 4.2.1. The detailed simulation environment is similar to that in [3], described in Section 6. We can see that BPS performs much better than BRS in search speed, update speed, and memory consumption. The update speeds of BPS and BRS are very slow, which is five orders of magnitude slower than the search speed. Notice that the search time ratio of BRS and BPS should be close to $\log(N_{BRS})/\log(N_{BPS})$ theoretically because they both follow the binary searches. Our results for these two tables are better than the theoretical results because the node structure of BPS is smaller than that of BRS. In addition to the two tables in Table 4, We also verify the search performance improvement for BPS with 10 other routing tables of sizes from 79,530 to 191,810. The search time ratio of BRS and BPS is between 1.13 and 1.41.

4.2 Optimizations for the Proposed Binary Prefix Search

To improve the update speed and the search speed, the simplest solution may be the use of a *k-bit segmentation table* [3], [9]. In this section, we discuss four approaches to the implementation of the k-bit segmentation table. For the moment, we assume that the most significant k bits in the W-bit address space are used to construct the k-bit segmentation table. Later on, we shall discuss the better schemes for selecting these k bits.

4.2.1 Simple k-Bit Segmentation Table

This approach is the traditional k-bit segmentation table that is implemented as an array of 2^k elements. This k-bit segmentation table divides the W-bit address space into 2^k segments of 2^{W-k} addresses. Each element in k-bit segmentation table contains two fields, namely, the default port of the segment and the pointer pointing to the prefix subset that contains the prefixes of lengths longer than k covered by the address space of the segment. When the update operations are needed, each element needs one more field, called the *enclosure list*, in which the prefixes that completely cover the address space of the corresponding segment are stored. An expansion list for the proposed BPS is built for the prefixes of each segment. Specifically, if an original prefix is longer than k bits and the value of the first k bits of the prefix is p, it is stored in the pth segment. If an original prefix P is of length $l \leq k$, it is stored in the enclosure list of the 2^{k-l} segments whose address spaces are completely covered by P. The port of the longest prefix in the enclosure list is stored in the default port field of the segment. Notice that the memory space for the enclosure lists is not usually considered as a requirement for the search operations. The enclosure lists are only required for update operations and are thus stored in the memory space independent of that used for searches. We adopt the following update process. First, the affected segments are computed from the prefix (say P) to be inserted or deleted. Second, if the length of P is shorter than or equal to k, only the default port fields and enclosure lists of the affected segments need to be updated. If P is longer than k, only the expansion list pointed to by the pointer field of the affected segment needs to be updated. We update each affected segment sequentially. When a segment is busy on the update process, only the search operations accessing this segment are suspended until the update process is completed. No other search operations are postponed.

The search algorithm works as follows: The first k bits of the destination address are used to index the k-bit segmentation table. It takes only one memory access to locate the target segment consisting of a subset of the original prefixes to be matched with the remaining $W - k$ bits of the destination address. If the LPM is found, then the search succeeds. Otherwise, the default prefix recoded in that segment is the LPM. As we can see, the larger is k, the smaller is the maximum expansion list among all segments, and thus, the faster is the search speed.

Consider the update process. When the prefix to be inserted or deleted is longer than k bits, only the expansion list of the segment that covers the prefix needs to be updated. However, when the prefix to be inserted or deleted is of a length l that is smaller than or equal to k, the enclosure lists of 2^{k-l} segments have to be updated. Therefore, in order to minimize the worst-case update overhead, k must not be very large. This contradicts the larger k required for a faster search speed.

Table 4 also shows the performance of *BPS* and *BRS* [15] with a 16-bit segmentation table, denoted by *BPS*-16 and *BRS*-16, respectively. Obviously, the update speeds of both *BPS*-16 and *BRS*-16 are much better than their counterparts, *BPS* and *BRS*, respectively. Their search speeds are also improved. However, the memory requirement increases too.

4.2.2 k-Bit Segmentation Table and k_Set

This approach is a variant of the first approach for improving the update speed and memory requirement. The original prefixes of lengths shorter than k bits are not duplicated in the enclosure list of the k-bit segmentation table. Instead, they are placed in an additional subset, called k_Set. As in the first approach, the expansion lists of nonempty segments need to be constructed. The expansion list constructed from the prefixes in k_Set is also needed.

When performing a search, the first k bits of the destination address are used to locate the target segment. Then, the BPS is performed in the expansion list of the target segment to find the LPM. If no LPM is found, an additional BPS needs to be executed in the k_Set expansion list. When a prefix is to be inserted or deleted, only the prefix subset in the corresponding segment or the k_Set needs to be updated. Since either the segment size or k_Set size is much smaller than the original prefix set, the update performance will be greatly improved.

If we choose a large k, one problem of this approach is that the k_Set could be larger when compared to the segments. A large k_Set degrades the performance of the worst-case search and update operations. Fortunately, we can reuse the idea of the segmentation table to further reduce the search space in the k_Set. Because the k_Set contains only the prefixes of a length shorter than k, we can build an l-bit segmentation table and an l_Set for k_Set. We denote this further improvement as the (k, l)-bit segmentation table approach. The worst-case search operations now include the searches in the target segment, the target segment from k_Set, and the l_Set. Since there is no prefix of length shorter than 8 in the real routing tables, the best choice of l is 8. By choosing 8 for l, the l_Set will be empty. As a result, the worst-case number of prefix subsets that must be searched is still two. Thus, the worst-case search and update performance will be improved. For example, the fourth row of Table 5 shows the sizes of the largest segment and k_Set for two real routing tables. We can see that k_Set is much larger than the largest segment. After applying the 8-bit segmentation table to k_Set, the largest segment of the 8-bit segmentation table becomes much smaller than k_Set.

TABLE 5
The Maximum Number of Prefixes in a Segment for an (8, 16)-Bit Segmentation Table

Funet-2000-4			AS6447-2002-4		
16-bit segmentation			16-bit segmentation		
Max segment	k_Set		Max segment	k_Set	
261	541		297	806	
	8-bit segmentation			8-bit segmentation	
	Max seg	l_Set		Max seg	l_Set
	32	0		35	0

4.2.3 Simple k_Set

In this approach, the k-bit segmentation table does not physically exist and the original prefixes of lengths longer than or equal to k are not divided into 2^k subsets. As in the second approach, the original prefixes of lengths shorter than k are placed into the k_Set. Other prefixes are placed into a single subset called the $main\_Set$. The expansion lists for both the $main\_Set$ and k_Set are constructed for the search process. We first search the $main\_Set$. If a match is found, it must be the longest match. Otherwise, we have to search the k_Set for a match. Compared to the original *BPS*, the number of auxiliary prefixes in the expansion list is reduced, and thus, the memory storage is also reduced. Obviously, the update process is as simple as the second approach. As stated in the second approach, the same idea for this approach can be reused for k_Set. For example, we can use a simple l_Set approach to divide the k_Set into two subsets containing the prefixes of lengths from $k - 1$ to l and the prefixes of lengths shorter than l, respectively.

4.2.4 k_Set and Hashing

This approach is similar to the third one except that it uses a hashing table to evenly divide the prefixes in $main\_Set$ into many small subsets. The prefixes in $main\_Set$ are of lengths greater than or equal to k. This approach selects j bits out of the first k bits to divide $main\_Set$ into 2^j subsets. The expansion lists of all 2^j subsets need to be constructed to apply the proposed BPS. The second approach is a special case of this approach when $j = k$.

A greedy scheme or a more intelligent scheme can be developed to find the best j bits to partition $main\_Set$. Theoretically, there are totally $C_j^k = k(k-1) \times \cdots \times (k-j+1)/j!$ possible choices for selecting j bits out of k bits. The best selection of j bits must minimize the total memory requirement for the routing table and minimize the total prefixes in the k_Set and the largest segment. The latter

Range	0 | 3	5 | 5	7 | 17	10 | 15	21 | 28	23 | 23
Port or ID	A	B	C	D	E	F

(a)

Endpoint	0	3	5	7	10	15	17	21	23	28	31
= Port	A	A	B	C	D	D	C	E	F	E	ϕ
> Port	A	ϕ	ϕ	C	D	C	ϕ	E	E	ϕ	-

(b)

Endpoint	3	4	5	6	9	15	17	20	22	23	28	31
Port	A	ϕ	B	ϕ	C	D	C	ϕ	E	F	E	ϕ

(c)

Fig. 4. Different approaches for ranges. (a) The list of six 5-bit original ranges. (b) The binary range approach proposed in [14]. (c) The proposed binary range approach (BRS_Int).

```
BRS_Int(Element E[], Index L, Index R, Address Dst)
Begin // E[] is the expansion list of sorted integer endpoints and the associated ports.
      // L and R are the left and right indices of E[] and Dst is the destination IP address.
    If ($Dst \le E[L].endpoint$) Then return E[L].port;
    $M = \lceil (L+R)/2 \rceil$;
    If ($E[M].endpoint \le Dst$) Then return BRS_Int(E, M+1, R, Dst);
    return BRS_Int(E, L, M, Dst);
End
```

Fig. 5. The proposed BRS for the integer number endpoints.

implicitly results in the best worst-case search speed. The exhaustive search to check all the possible combinations is too time-consuming. However, a more intelligent scheme may be too difficult, if not impossible. A better choice may be the greedy algorithm that selects one bit at a time with the specification of optimizing one metric that we are concerned with. One example of a greedy algorithm can be found in [25]. We will not describe any details on how to select the best j bits in this paper.

4.3 Proposed Binary Search for Ranges

How to define the endpoints of a range has a strong relationship to BRS performance. Traditionally, the two endpoints of a range $[L, U]$ are L and U. Additional information must also be recorded to identify if the destination address is equal to L or U of a range. For example, the *BRS* proposed in [15] records the "$=$" and "$>$" ports for solving the cases when the destination address is equal to an endpoint and when the destination address locates between two endpoints, respectively. With "$=$" and "$>$" ports, the BRS can be performed as follows: If the destination address d is equal to an endpoint $E[i]$ (or between two endpoints $E[i]$ and $E[i+1]$), then the final result is $E[i].=port$ (or $E[i].>port$). Fig. 4b shows the expansion list for five original ranges in Fig. 4a based on the BRS in [15]. In this paper, we propose a BRS based on a different endpoint definition (called BRS_Int) as in [4], where each new endpoint is associated with only one port. As a result, memory requirement of the proposed binary search is smaller than the BRS proposed in [15].

Definition 4 [4]. *The two integer endpoints of a range $[L, U]$ are defined as $L-1$ and U when $L \neq 0$, but only one endpoint U is defined when $L = 0$.*

The endpoint 0 is only stored when the lower address of a range is 1 (e.g., $[1, U]$) or when the upper address of a range is 0 (i.e., [0, 0]). Let the expansion list be $E[0, \ldots, s-1]$. The elementary intervals formed by the expansion list are $[0, E[0]]$, $[E[i]+1, E[i+1]]$ for $i = 0$ to $s-2$, and $[E[s-1]+1, 2^W-1]$ if $E[s-1] \neq 2^W-1$. To simplify the search operation, if $E[s-1] \neq 2^W-1$, we add an additional element $E[s]$ of value 2^W-1. In the expansion list, $E[0].port$ is the port associated with the elementary interval $[0, E[0]]$ and $E[i].port$ for $i = 1$ to $s-1$ is the port associated with $[E[i-1]+1, E[i]]$. For example, the expansion list of the range set in Fig. 4a can be constructed as shown in Fig. 4c.

The proposed binary search algorithm is formally shown in Fig. 5. Basically, the search determines two cases where $Dst \le E[0].endpoint$ and $E[i].endpoint < Dst \le E[i+1].endpoint$ for i > 0. For the first case, the matched port is $E[0].port$. For the second case, the matched port is $E[i+1].port$. If the destination address *Dst* is smaller than or equal to the $E[L].endpoint$, then the matched port is $E[L].port$. Otherwise, we compute the middle index $M = \lceil (L+R)/2 \rceil$. Depending on whether $E[M].endpoint \le Dst$ or not, we recursively call $BRS\_Int(E, M+1, R, Dst)$ or $BRS\_Int(E, L, M, Dst)$.

4.4 Optimizations for the Proposed Binary Range Search

Four approaches for the k-bit segmentation table are discussed. Because these four approaches are similar to their counterparts for prefixes, only the differences will be described.

4.4.1 Simple k-Bit Segmentation Table

An array of 2^k elements is used as the k-bit segmentation table. If a range overlaps with any part of the address space of a segment, it is stored in the range subset of that segment. For each nonempty segment, the proposed BRS (BRS_Int) is performed.

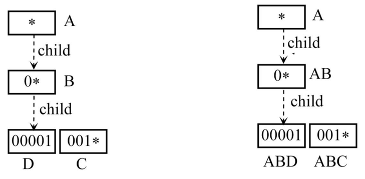
4.4.2 k-Bit Segmentation Table and k_Set

The range overlapped with at least three segments is placed into the k_Set; otherwise, it will be placed into the segments with which it overlaps. As in the case for prefixes, the expansion lists for all nonempty segments and the k_Set expansion list need to be constructed.

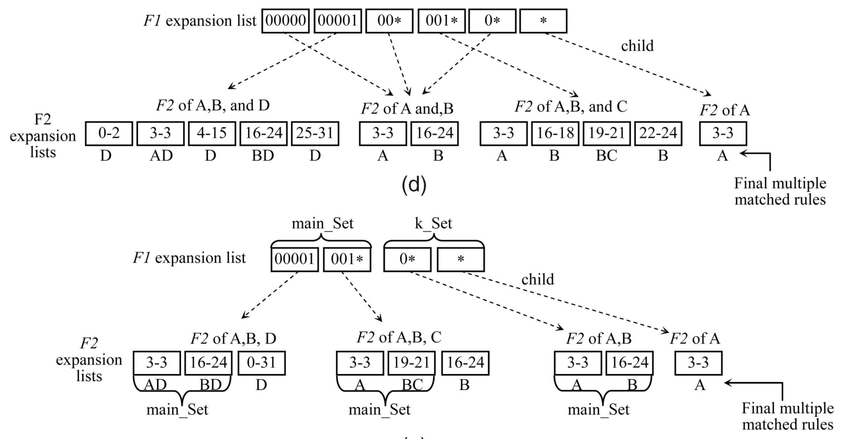
The update process for a range R is limited to the segment that completely covers R, the segments partially overlapped with R, or the k_Set. In the worst case, the expansion lists of

Rule (Type)	F1 (Prefix)	F2 (Range)
A	*	3
B	0*	16-24
C	001*	19-21
D	00001	0-31

(a)



(b) (c)



(d)

(e)

Fig. 6. A 2D packet classification example consisting of $F1$ prefixes and $F2$ ranges. (a) Example. (b) $F1$ hierarchical list. (c) Rule pushing operation. (d) The complete hierarchical expansion list. (e) Simple k_Set approach of $k = 2$ with rule duplication.

three segments need to be modified. The size of the original expansion list is much larger than the segment expansion list when k is large, say $k > 8$. Therefore, the gain in search speed from shrinking the segment expansion list will be much more than the loss in searching the k_Set expansion list. The overall search speed will also be improved.

4.4.3 Simple k_Set

The k-bit segmentation table does not physically exist. The ranges that completely cover any segment or the ranges that overlap with at most two segments are placed into the k_Set. Other ranges are placed into the $main\_Set$.

4.4.4 k_Set and Hashing

This approach is similar to the prefix counterpart in Section 4.2.4.

5 Multidimensional Packet Classification

In the multidimensional PC problem, we are given a set of rules $R = \{R_1, \dots, R_n\}$ over d fields (dimensions). A d-dimensional rule is in the form of $R_i = (F1_i, \dots, Fd_i)$, where Fk_i for $k = 1$ to d, called the kth filter, is a W_i-bit prefix or range. Each rule is associated with a cost or priority. We use $R = (F1, \dots Fd)$ when no confusion is incurred. The header field values of a packet are denoted as $(P1, \dots, Pd)$. A rule is applicable to a packet if for each dimension k, the field value Pk of the packet lies in the range covered by Fk. The multidimensional PC problem is to find the rule with the least cost or the highest priority that applies to the packet.

5.1 Hierarchical Expansion List

We now illustrate how the proposed *BPS* and the proposed BRS (BRS_Int) can be applied to the multidimensional PC problem. The basic data structure proposed in this paper for the multidimensional PC is a hierarchical expansion list of *BPS* and BRS_Int. The hierarchical data structure is built as follows:

1. Build the $F1$ hierarchical list based on the $F1$ field values of the rules.
2. Push all subrules $(F2, \dots, Fd)$ associated with the elements of $F1$ hierarchical list in higher levels to their children in lower levels. We call this operation the *rule pushing*.
3. Perform an expansion traversal as shown in Fig. 2 in the $F1$ hierarchical list to construct the $F1$ expansion list. Now, each element of the $F1$ expansion list contains a number of $(d-1)$-dimensional subrules $(F2, \dots, Fd)$.
4. Continue steps 1, 2, and 3 for each dimension except for the rule pushing operation (step 2) which is not required for the last dimension.

Consider an example 2D rule set in Fig. 6a. We first build the $F1$ hierarchical list according to the $F1$ values of all rules as shown in Fig. 6b. Second, the rule pushing operation is performed for field $F1$ as shown in Fig. 6c. Then, the $F1$ expansion list is obtained by performing the expansion traversal on the $F1$ hierarchical list. Each element of the $F1$ expansion list now contains a number of subrules consisting of only $F2$ fields. By repeating the same construction step, all $F2$ hierarchical lists from the subrules $(F2)$ are constructed. Since $F2$ is the last dimension, no rule pushing is required. Finally, we perform the expansion traversals on all the $F2$ hierarchical lists to obtain the $F2$ expansion lists. The complete hierarchical expansion list is shown in Fig. 6d. Notice that we show the final multiple matched rules at the bottom of Fig. 6d. If a

```
// Addresses P[1], …, P[d] are the field values in the headers of a packet
// L and R the bounds of array List
// f[] is a global array and f[m] is the field format for dimension m for m = 1 to d
Proposed_PC_Algorithm(Element List[], Index L, Index R, Address P[])
Begin
    Matched_Rule = ϕ; m = 1;
    While (List != NULL) Begin
        If (f[i] is in prefix format) Idx = BPS_search(List, L, R, P[m])
        Else Idx = BRS_Int(List, L, R, P[m]);
        If (Idx = –1) Then return Matched_Rule;
        Matched_Rule = List[Idx].rule_number;
        List = List[Idx].child;
        m = m + 1;
    End
    return Matched_Rule;
End
```

Fig. 7. The proposed multidimensional classification algorithm.

priority is given to each rule, the highest priority rule can be obtained easily.

Now, we describe how the proposed data structure is matched with the headers of the incoming packets. The complete multidimensional packet classification algorithm *Proposed_PC_Algorithm()* is illustrated in Fig. 7. *Proposed_PC_Algorithm()* uses either the proposed binary prefix search *BPS_search()* or the proposed binary range search $BRS\_Int()$ to find the matched rules against the header fields of the incoming packet. The output of *Proposed_PC_Algorithm()* is a set of matched rules. Assume that the headers of the incoming packet are (0, 3) and that the rule set is from Fig. 6a. The $F1$ expansion list in Fig. 6d is first checked against the $F1$ field of the packet with address 0. A binary search finds that the longest prefix match is the first element 00000. Then, by following the child pointer from element 00000, the binary search is performed against the field $F2$ of the packet with address 3. Interval [3, 3] is matched, and thus, the final matched rule is A.

5.2 Optimizations

The number of the expansion lists and the total size of all these expansion lists have a direct impact on the memory requirement and search speed. By carefully analyzing the hierarchical expansion lists, we develop the following optimization techniques to reduce the memory space needed by the proposed algorithm and improve the search speed. The first two techniques are basically designed for saving memory. However, the other two techniques are designed for improving the search speed.

5.2.1 List Sharing

Many auxiliary prefixes or default intervals are generated from the same original prefix and range fields of the original rules and thus share the same set of subrules. Therefore, some expansion lists pointed to by these auxiliary prefixes or default intervals are very likely the same and can be shared. For the example in Fig. 6d, the auxiliary prefixes 00000, 00^*, and 0^* generated by BPS_Int on the $F1$ field point to the same set of subrules [3, 3] and [16, 24], and thus, they share the same $F2$ expansion list of [3, 3] and [16, 24].

5.2.2 Sequential Search

If the number of subrules left to be matched in a node of the hierarchical list is less than a predefined threshold, it is reasonable to perform a sequential search instead of a more complicated search algorithm like the ones proposed in this paper.

5.2.3 Hierarchical k-Bit Segmentation Table

In this optimized scheme, we avoid generating too many auxiliary prefixes or default intervals by using the four approaches of the k-bit segmentation table proposed earlier for 1D prefixes and ranges in a dimension-by-dimension manner.

Consider the rule set in Fig. 6a and the simple k_Set approach with $k = 2$. We first construct the $F1$ expansion list. Because rules A and B are of a length shorter than 2, we have $main\_Set = \{C, D\}$ and $k\_Set = \{A, B\}$. Since $main\_Set$ and k_Set are smaller than the original rule set, two shorter $F1$ expansion lists are obtained. After the $F1$ expansion lists are obtained, we have two options to construct the $F2$ expansion lists: with or without rule duplication.

Option with rule duplication. If any rule in k_Set covers a rule in $main\_Set$, its $(d-1)$-dimensional subrule is duplicated in the $main\_Set$. Fig. 6e illustrates the complete expansion lists with rule duplication. Because the root of each subhierarchical expansion list is associated with a k_Set expansion list, the classification algorithm needs to be modified to comply with the modified data structure as follows: We first search the $F1$ expansion list of the $main\_Set$. If a matched element is found, we continue the search in the $F2$ expansion lists pointed to by the matched element. Whether or not a final matched rule is found, we do not go back to search the $F1$ expansion list of k_Set. If the search is not satisfied with the $F1$ expansion list of the $main\_Set$, the $F1$ expansion list of k_Set must be searched. In the worst-case scenario, both the $main\_Set$ and the k_Set need to be searched in each level of the hierarchical expansion lists. However, based on our performance experiments, searching two small expansion lists of the $main\_Set$ and k_Set is faster than searching the large expansion list of the union of the $main\_Set$ and the k_Set. Thus, both the search speed and memory consumption improve over the original hierarchical expansion list.

Option without rule duplication. No $(d-1)$-dimensional subrule of k_Set in $F1$ field is duplicated in $main\_Set$. As a result, the $(d-1)$-dimensional subrule set pointed to by any element in the $F1$ expansion list becomes smaller than the original one. The memory consumption is further reduced. However, we need to search both the $main\_Set$ and the k_Set in each level of the hierarchical expansion lists. The search speed may be worse than the original one. However, if the depth of the hierarchical expansion list is not high (e.g., only $F1$ and $F2$ fields are used to construct the hierarchical expansion list in the 5D packet classification), this approach may be a good choice.

Selecting the dimension order. It is not difficult to learn that the performance in terms of memory consumption, search speed, or update speed will be different if the dimension order used by the proposed hierarchical scheme is different. Some packet classification algorithms proposed in the literature are restricted to the prefix fields. Therefore, only the source and destination address fields are used to construct the desired data structure and the subsequent subrules will be processed linearly. Examples are the trie-based schemes such as hierarchical trie, grid of trie, and extended grid of trie with path compression, to name a few.

We use the following heuristics to select the best dimension order. We only discuss how to select the first dimension since the process of selecting the second and succeeding dimensions is similar. The first factor to consider is the size of the $F1$ expansion list denoted as $F1\_size$. The second factor is the size of the largest $F2$ expansion list denoted as max_F2_size. For the example in Fig. 6d, the size of the $F1$ expansion list is 6 and the size of the largest among all the $F2$ expansion lists is 5. We select the dimension such that the sum of $F1\_size$ and max_F2_size is minimal. Based on our experiments, the memory sizes are much different if the dimension order is different, but the search performance has no significant difference. Although other heuristics proposed in HiCuts [14] and HyperCuts [19] can also be used, their performance gains strongly depend on the characteristics of the rule tables. Therefore, it is not easy to evaluate which heuristic performs better than the others. Notice that in the proposed scheme, the first two dimensions have the strongest impact on the overall performance in memory consumption. If more preprocessing time is allowed, an exhaustive search may be a better choice.

5.2.4 d-Dimensional $k_1 \times \cdots \times k_d$-Bit Segmentation Table

We have shown that the k-bit segmentation table can be used one dimension at a time sequentially. In fact, the concept of the k-bit segmentation table can be applied for more than one dimension at once. Therefore, the *d-dimensional $k_1 \times \cdots \times k_d$-bit segmentation table* is proposed. The modular packet classification [24] and HyperCuts [19] employed a similar technique. Because the size of the d-dimensional $k_1 \times \cdots \times k_d$-bit segmentation table is selected statically, one main difference from the modular packet classification [24] and HyperCuts [19] is that the proposed *$k_1 \times \cdots \times k_d$-bit segmentation table* has a better performance in update speeds.

In the proposed $k_1 \times \cdots \times k_d$-bit segmentation table, the address space $2^{W_1} \times \cdots \times 2^{W_d}$ is divided into $2^{k_1} \times \cdots \times 2^{k_d}$ hyperrectangles of size $2^{W_1-k_1} \times \cdots \times 2^{W_d-k_d}$. We shall use the names hyperrectangle, rectangle, and segment interchangeably. Each hyperrectangle is associated with a small number of rules. As in the 1D k-bit segmentation table implemented with an array of 2^k elements, if only one memory reference is needed to reach the small set of rules associated with the hyperrectangle that matches the incoming packet, then the search performance will be greatly improved. In the real 5D classifier, a 2D $k_1 \times k_2$-bit segmentation table may have been sufficient to divide the total address space into smaller rectangles such that the number of associated rules in each rectangle is not large. In other words, a larger segmentation table like the 3D $k_1 \times k_2 \times k_3$-bit segmentation table may not be needed. Subsequently, we only discuss the 2D $k_1 \times k_2$-bit segmentation table in the $W_1 \times W_2$ address space to illustrate the idea of the proposed d-dimensional $k_1 \times \cdots \times k_d$-bit segmentation table. Since all the proposed optimization techniques such as simple array of 2^k elements, k_Set, etc. can be applied directly, we only explain the proposed scheme using an array of 2^k elements and k_Set.

In the proposed 2D $k_1 \times k_2$-bit segmentation table, we have four kinds of rule subsets, namely, $k_1k_2\_Set$, k_1_Set, k_2_Set, and segment $S_{i,j}$ for $i = 0$ to $2^{k_1} - 1$ and $j = 0$ to $2^{k_2} - 1$. There is the 2^{k_2} k_1_Set and the 2^{k_1} k_2_Set. $S_{i,j}$ denotes the address area of $[i \times 2^{W_1-k_1}, (i+1) \times 2^{W_1-k_1} - 1] \times [j \times 2^{W_2-k_2}, (j+1) \times 2^{W_2-k_2} - 1]$. Assume that a rule R covers the area of $[X1, X2] \times [Y1, Y2]$. Rule R is duplicated in the $k_1k_2\_Set$, k_1_Sets, k_2_Sets, or segments based on the rule placement algorithm along with two parameters c_1 and c_2 as follows:

Let $i_2 = \lfloor X2/2^{W_1-k_1} \rfloor, i_1 = \lfloor X1/2^{W_1-k_1} \rfloor, j_2 = \lfloor Y2/2^{W_2-k_2} \rfloor$, and $j_1 = \lfloor Y1/2^{W_2-k_2} \rfloor$:

1. If $i_2 - i_1 + 1 > c_1$ and $j_2 - j_1 + 1 > c_2$, then R is put in the $k_1k_2\_Set$.
2. If $i_2 - i_1 + 1 \le c_1 \times c_2$ and $j_2 - j_1 = 0$, then R is put in S_{i,j_1} for $i = i_2, \ldots, i_1$. Otherwise, if $i_2 - i_1 + 1 > c_1$ and $j_2 - j_1 + 1 \le c_2$, then R is put in $k_1\_Set_j$ for $j = j_2, \ldots, j_1$.
3. If $i_2 - i_1 = 0$ and $j_2 - j_1 + 1 \le c_1 \times c_2$, then R is put in $S_{i_1,j}$ for $j = j_2, \ldots, j_1$. Otherwise, if $i_2 - i_1 + 1 \le c_1$ and $j_2 - j_1 + 1 > c_2$, then R is put in $k_2\_Set_i$ for $i = i_2, \ldots, i_1$.
4. If $i_2 - i_1 + 1 \le c_1$ and $j_2 - j_1 + 1 \le c_2$, then R is put in $S_{i,j}$ for $i = i_2, \ldots, i_1$ and $j = j_2, \ldots, j_1$.

A rule is placed in at most $c_1 \times c_2$ segments, at most c_2 k_1_Sets, at most c_1 k_2_Sets, or in the only $k_1k_2\_Set$. Fig. 8 shows the final rule placement result with $c_1 = 2$ and $c_2 = 1$ for an example rule set consisting of seven rules in which the first and the second fields are in the prefix and range formats, respectively. The hierarchical expansion list is constructed for each subset. The proposed BPS or BRS can be applied as follows: If the header values of the incoming packet fall into an address space covered by the segment $S_{i,j}$, then the hierarchical expansion lists of the rule subsets corresponding to $S_{i,j}$, $k_1\_Set_j$, $k_2\_Set_i$, and $k_1k_2\_Set$ are searched for the best match.

The main purpose of increasing k_1 and k_2 is to have a small segment. The larger the values of k_1 and k_2, the smaller is the address space covered by each segment, and thus, the smaller is the size of rule subset belonging to a segment. Thus, searching a segment will be faster when k_1 and k_2 are large. However, larger k_1 and k_2 will cause many rules that originally belong to a segment with smaller k_1 and k_2 to be moved to k_1_Set, k_2_Set, or $k_1k_2\_Set$. As a result,

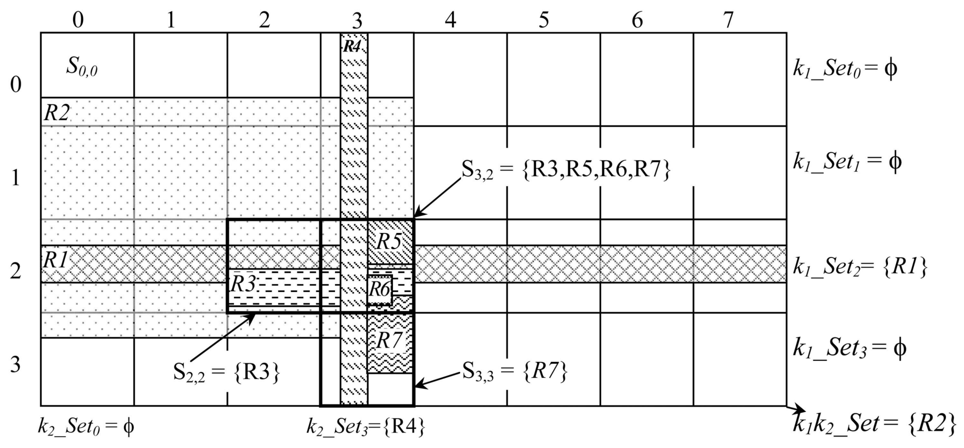


Fig. 8. A 2D 3×2-bit segmentation table with $c_1 = 2$ and $c_2 = 1$ for a set of seven rules.

the sizes of k_1_Set, k_2_Set, and $k_1k_2\_Set$ increase, and thus, performing searches in them slows down. To overcome this contradiction, we trade the memory space for faster search speed by introducing the rule duplication factors c_1 and c_2. Large c_1 and c_2 prevent k_1_Set, k_2_Set, or $k_1k_2\_Set$ from being increased too much while k_1 and k_2 increase. For example, when c_1 is set to 2 in the example of Fig. 8, rules $R3$ and $R5$ are duplicated into two segments. However, if c_1 is set to 1, rules $R3$ and $R5$ will be put in $k_1\_Set_2$ and $k_2\_Set_3$, respectively.

Notice that if we employ the approach similar to the simple *k-bit segmentation table*, k_1_Set, k_2_Set, and $k_1k_2\_Set$ can be completely removed by duplicating the rules that are supposed to be placed in k_1_Set, k_2_Set, or $k_1k_2\_Set$ into segments. In other words, a rule must be duplicated in the segment $S_{i,j}$ as long as it overlaps $S_{i,j}$. In this case, we only search one rule subset instead of four separate searches in the corresponding segment, k_1_Set, k_2_Set, and $k_1k_2\_Set$. At the first glance, we may think that the simple *k-bit segmentation table* approach may perform better than the approach using k_1_Set, k_2_Set, and $k_1k_2\_Set$. However, as our performance experiments show, the approach using k_1_Set, k_2_Set, and $k_1k_2\_Set$ performs better because any segment, k_1_Set, k_2_Set, or $k_1k_2\_Set$ is much smaller than the segment in the simple *k-bit segmentation table.* In other words, searching four small rule subsets is faster than searching a large rule subset by using the proposed binary prefix and range searches.

Further improvement. This further improvement is applicable to the applications that only need to find the highest priority rule instead of all the matched rules. Take as an example the case wherein all the four rule subsets, the segment $S_{i,j}$, k_1_Set, k_2_Set, and $k_1k_2\_Set$, need to be searched. We record the highest priority values of k_1_Set, k_2_Set, and $k_1k_2\_Set$ independently. Thus, after finding the matched rule with the highest priority value *pri* in $S_{i,j}$, if *pri* is higher than the highest priority among the rules in k_1_Set, we no longer need to search k_1_Set. A similar process can be applied to k_2_Set and $k_1k_2\_Set$ to avoid searching in k_2_Set and $k_1k_2\_Set$. Although this optimization must improve the search performance, the update process may be affected. For example, if the highest priority rule is deleted from k_1_Set, we have to search the second highest priority in k_1_Set to make it become the highest one. Fortunately, a simple priority queue can serve as the solution. As a result, the rule deletion speed will be as fast as the original one. The memory required for the priority queue is small and, thus, has a negligible impact on the total memory requirement for the proposed scheme.

5.3 Complexity

We only give the worst-case complexities of search and memory requirement for the hierarchical BPS tree (i.e., the unoptimized data structure). Other optimizations will have the same worst-case performance as the unoptimized one although their average performance will be much better. First, consider the $F1$ expansion list of a rule set containing N d-dimensional rules. We have shown that there are at most $2N - 1$ elements in the $F1$ expansion list. Each element of the $F1$ expansion list is a prefix or an interval that is assumed to be covered by $F1$ fields of $O(N)$ rules. In other words, the size of the set of $(d-1)$-dimensional rules pointed to by each element of the $F1$ expansion list is $O(N)$. As a result, the worst-case space complexity of the proposed data structure can be formulated in the following recursive equation: $S(N,d) = O(N) + O(N) \times S(N, d-1)$. Thus, the worst-case space complexity is $S(N,d) = O(N^d)$. In practice, the size of the set of $(d-1)$-dimensional rules pointed to by each element of the first-level $F1$ expansion list is much smaller than $O(N)$. Therefore, the practical memory requirement will be much smaller than the theoretical result. Also, the worst-case time complexity of a search operation is $O(d \times \log N)$ because the expansion list in each of the d levels has $O(N)$ elements. Similarly, the practical search performance will be much faster than the theoretical worst-case result.

6 Performance Evaluation

In this section, we evaluate the performance of the proposed classification algorithms and compare it with other prominent dynamic schemes in terms of search speed, memory requirement, and update speed. We use the *ClassBench* [22] to generate IP traffic and the rule sets that are close to real industrial classifiers. The real implementations of all the classification schemes based on the same rule tables and IP traffic allow us to have fair comparisons and obtain convincing results. The implementation platform is a Debian GNU/Linux 4.0 system running on a 3-GHz Pentium 4 containing a 16-Kbyte L1 cache, a 1-Mbyte L2 cache, and

512 Mbytes of main memory. The line size of both L1 and L2 caches is 64 bytes. We believe the results would be similar if the experiments are executed on other comparable platforms such as the Alpha. We conduct experiments for the 5D rule sets of up to 10,000 rules generated by *ClassBench.* We implement the following three proposed schemes. All these three proposed schemes use only the first two prefix fields to construct the hierarchical expansion lists. The remaining three fields are organized in the form of linear lists as the existing decision schemes like HyperCuts:

1. *Scheme H.* The basic hierarchical expansion list described in Section 5.1.
2. *Scheme* $H\_k(k_1, k_2)$. The hierarchical expansion list optimized by the simple k_Set approach without rule duplication as described in Section 5.2.3. k_1 and k_2 indicate that both the first and second prefix fields use the simple k_Set approach.
3. *Scheme* $H\_2k(k_1, k_2, c_1, c_2)$. The hierarchical expansion list optimized by a 2D $k_1 \times k_2$-bit segmentation table of the first two prefix fields as described in Section 5.2.4. For each search operation, we have to search four rule subsets that are a segment, k_1_Set, k_2_Set, and $k_1k_2\_Set$.
4. *Scheme* $H\_2k\_opt(k_1, k_2, c_1, c_2)$. The improved version of $H\_2k(k_1, k_2, c_1, c_2)$ optimized by the highest priority arrangements described in Section 5.2.4.

6.1 Characteristics of Rule Tables

The characteristics of different rule tables have a substantial impact on the performance of packet classification algorithms. Therefore, we shall briefly describe some well-known characteristics of the rule tables. The rule tables of various sizes generated by ClassBench with parameters "*random*," "*acl1_seed*," "*fw2_seed*," and "*ipc1_seed*" will be studied.

In the *random* table, the number of distinct field values in any of the first four fields is very large. The numbers of wildcards in the first two prefix fields only account for 3 percent of the total numbers of rules and no wildcard is in any of the two range fields. The number of distinct field values is fixed at around 250 and there are very few wildcards in the F5 field (protocol). For ACL, Firewall, and IPC tables, one common feature is that the number of distinct field values in $F3$, $F4$, and $F5$ is fixed at one value or a small range of values. In ACL tables, the only $F3$ field value is the wildcard. Similarly, in Firewall tables, the only $F4$ field value is the wildcard. In ACL tables, the number of distinct $F1$ values varies around 6 percent to 45 percent of the total number of rules and the number of distinct $F2$ values is fixed at about 300 to 600. The numbers of wildcards in $F1$ and $F2$ fields are also fixed at a small range of values. Firewall and IPC tables have a similar trend in the numbers of distinct $F2$ field values. However, they have a large gap in the numbers of distinct $F1$ field values. Also, the number of wildcards in $F1$ and $F2$ fields is large for Firewall tables compared with other tables.

In addition to the characteristics described above, we also show the 2D prefix length distribution of the $F1$ and $F2$ field prefixes for the tables of 1,000 rules in Fig. 9. The length distributions for other tables of different sizes are similar and thus not shown. The bar charts on the right side are drawn by grouping the lengths of 8-32 together for later uses. These 2D prefix length distributions are useful for understanding how to choose proper k_1, k_2, c_1, and c_2 parameters for the proposed scheme H_2k. We divide the length distribution into four regions marked as A, B, C, and D by drawing two additional lines at length 8 of both fields. Thus, when we set k_1 and k_2 to 8 or less, any rule belonging to region D will be put into one segment and will not be moved to k_1_Sets, k_2_Sets, or $k_1k_2\_Set$ if c_1 and c_2 are set to large values. When c_1 and c_2 are set to 1, the rules in regions B, C, and A will be put into k_1_Sets, k_2_Sets, and $k_1k_2\_Set$, respectively. Increasing c_1 and c_2 will only cause the rules in regions B, C, and A to be relocated to some segments.

6.2 Search Speed and Memory Usage

In this section, we first show the average search times and memory requirements for the proposed scheme H and the existing schemes EGT, EGT-PC, and HyperCuts. The source codes of EGT, EGT-PC, and HyperCuts are obtained from [7] and [13]. Table 6 and Fig. 10 illustrate the detailed results of the rule tables generated by ClassBench within the "random" parameter. We can see that scheme H has a much better search performance (about 2 to 10 times faster) than EGT, EGT-PC, or HyperCuts. When the size of a table is 1,000 rules or less, scheme H also achieves a good performance in memory requirement compared with Hypercut-1 that performs the best in terms of memory usages. However, the memory requirements of scheme H explode for the tables of 2,000 rules or larger because many rules are duplicated in the hierarchical expansion lists. This is why we propose the optimized schemes $H\_k(k_1, k_2)$, $H\_2k(k_1, k_2, c_1, c_2)$, and $H\_2k\_opt(k_1, k_2, c_1, c_2)$ to solve the memory explosion problem for large tables.

Table 7 and Fig. 11 illustrate the search speeds and memory usages of scheme H_k with $(k_1, k_2) = (4, 4)$, (4, 8), (8, 4), and (8, 8) for the Random rule tables of 1,000 or more rules. As we can see that $H\_k(4, 4)$ has the best average search speed and $H\_k(8, 8)$ has the smallest memory usages for all the rule tables. Except for the table of 1,000 rules, H_k is better than Scheme H both in search speed and memory usage. The performance results of other setting such as (8, 16), (16, 8), or (16, 16) that we also experimented are not shown because they are not so good. In general, the size of the $main\_set$ in $F1$ ($F2$) expansion list decreases as k_1 (k_2) increases. When k_1 $(k_2) = 16$, $main\_set$ in $F1$ ($F2$) expansion list becomes empty and all $F1$ ($F2$) field prefixes are moved to k_Set. As a result, the memory requirements of scheme H_k with large k_1 and/or k_2 are close to that of scheme H.

Now, we show the performance results of scheme $H\_2k(k_1, k_2, c_1, c_2)$. The detailed results of Random table of 1,000 rules are first shown in Table 8 to illustrate the general performance trend of Random tables, where the average and maximum numbers of rules in the subsets, segments $S_{i,j}$, k_1_Set, k_2_Set, and $k_1k_2\_Set$, are listed. Because the number of rules in each subset is small, using the hierarchical expansion list of scheme H as the search data structure allows for a faster search speed and smaller memory usage than EGT, EGT-PC, and HyperCuts. For a fixed (k_1, k_2) pair in $H\_2k(k_1, k_2, c_1, c_2)$, while c_1 and c_2 increase, the sizes of k_1_Set, k_2_Set, and $k_1k_2\_Set$ are reduced but the average or maximal segment size is increased. Overall, the search times decrease and memory usages increase while c_1 and c_2 increase. The rule tables of 2,000 or more rules that we have experimented will show a similar performance trend.

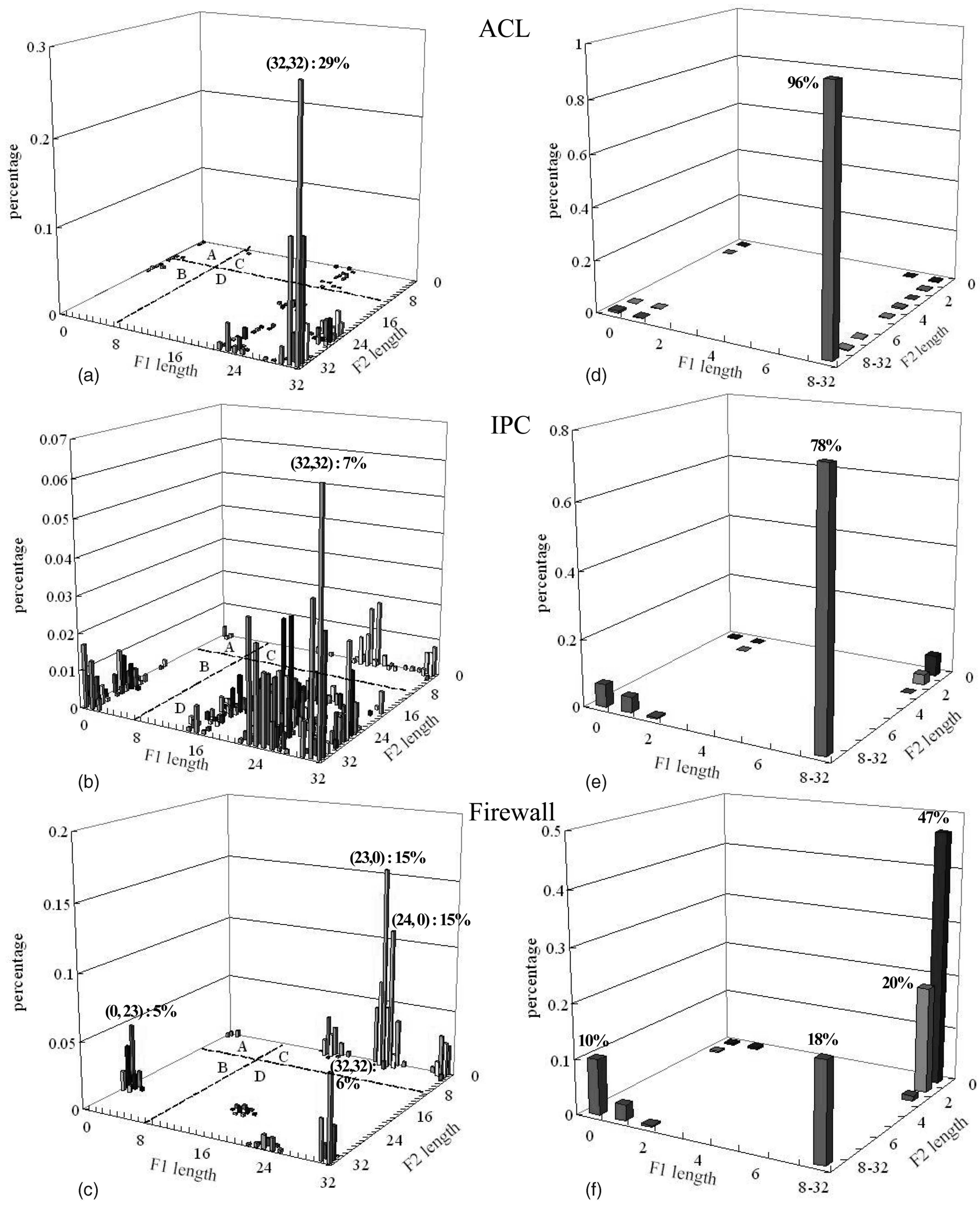


Fig. 9. Prefix length distribution for $F1$ and $F2$ prefix pairs of 1,000 rules.

To clearly represent the performance advantages of scheme H_2k, we show the best results based on the following selection criteria. We first sort the results in the increasing order of memory usages. Let M_{10} be the 10th smallest memory usage. Then, we select the best results as the one that has the shortest search time among the results whose memory usages are not more than M_{10}. Thus, the best result in Table 8 is $H\_2k(6, 6, 8, 8)$. Table 9 and Fig. 12 summarize the best settings of all tables of sizes 1,000 to 10,000. $H\_2k(8, 8, 8, 8)$ and H_2k_opt(8, 8, 8, 8) usually have the best search speed among all different settings. But they consume more memory than most of other settings and thus are not selected as the best settings. Compared to H_k as shown in Table 7, we can see that H_2k and H_2k_opt generally perform better than H_k in search speed and memory usage. H_2k_opt always performs better than H_2k both in search speed and memory usage, but the difference in search speeds between H_2k and H_2k_opt is not significant for the large tables consisting of 3,000 to 10,000 rules.

The results of the best settings based on the above selection criteria for all ACL, Firewall, and IPC tables are summarized in Table 10 and Fig. 13. For ACL and IPC tables,

TABLE 6
Performance of Scheme *H*, *EGT*, *EGT-PC*, and *HyperCuts* with Bucket Size$= 16$

Random Table Size		50	100	200	500	1,000	2,000	3,000	5,000	8,000	10,000
Search time (clock cycles)	EGT	550	673	1,219	1,006	1,470	2,847	2,680	3,229	4,222	5,974
	EGT-PC	670	782	959	1,502	1,656	2,749	2,868	3,263	4,740	5,723
	Hypercut-4	1,677	1,796	2,622	5,042	4,107	4,388	5,969	6,293	6,847	6,895
	Hypercut-1	1,674	1,701	3,237	4,736	3,326	4,277	5,164	5,652	6,113	6,868
	Scheme H	331	410	374	486	870	1,847	2,485	3,014	3,317	2,080
Memory usage (MB)	EGT	0.029	0.054	0.111	0.269	0.524	1.010	1.487	2.386	3.800	4.657
	EGT-PC	0.013	0.025	0.051	0.125	0.241	0.471	0.695	1.129	1.798	2.209
	Hypercut-4	0.004	0.009	0.038	0.179	0.395	0.661	3.420	6.301	9.872	10.90
	Hypercut-1	0.002	0.005	0.030	0.128	0.183	0.975	1.947	3.880	4.315	6.318
	Scheme H	0.003	0.016	0.049	0.268	0.350	1.270	3.640	9.290	26.20	51.33

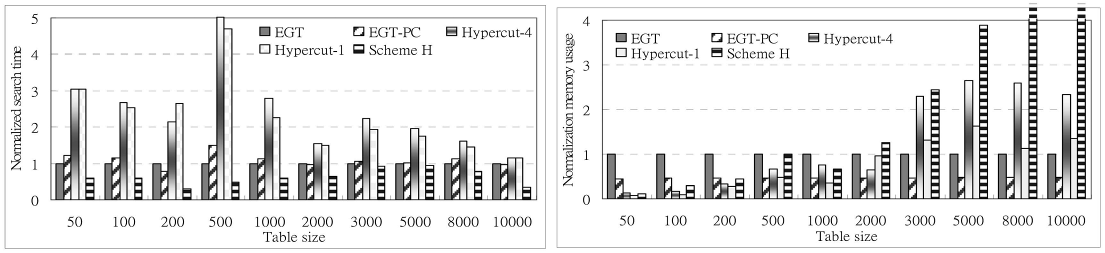


Fig. 10. Normalized performance from Table 6.

TABLE 7
Performance of Scheme H_k for Random Tables (the Best Results Are Shaded)

	(k_1, k_2)	1,000	2,000	3,000	5,000	8,000	10,000
Search time (clock cycles)	(4,4)	779	1,188	1,607	1,987	2,234	2,501
	(4,8)	883	1,247	1,685	1,992	2,288	2,618
	(8,4)	879	1,252	1,800	2,379	2,642	2,875
	(8,8)	901	1,255	1,856	2,459	2,705	2,979
Memory usage (MB)	(4,4)	0.07	0.21	0.42	1.00	2.29	3.50
	(4,8)	0.07	0.22	0.43	1.00	2.29	3.54
	(8,4)	0.09	0.21	0.39	0.88	1.29	1.95
	(8,8)	0.09	0.21	0.39	0.74	1.20	1.76

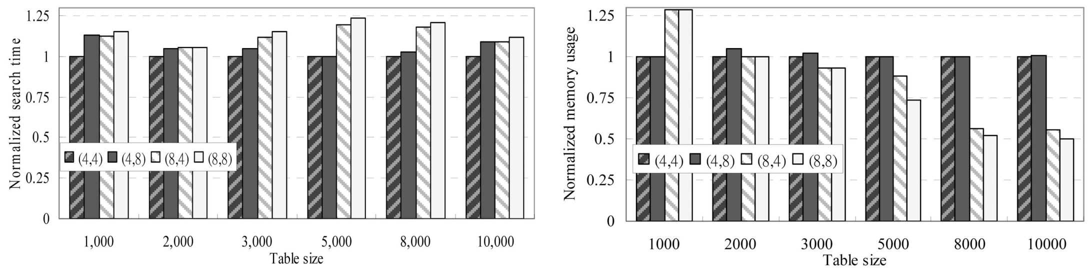


Fig. 11. Normalized performance of scheme H_k for random tables.

H_2k_opt is 1.66 to 1.93 times as fast as H_2k. However, for Firewall tables, H_2k_opt is only 1.07 to 1.12 times as fast as H_2k. For ACL tables with a fixed (k_1, k_2) pair, the search speed improves while c_1 and c_2 increase. The reason is as follows: The set of rules (say S), located in tuples of (i_1, i_2) for $i_1 = 8 \ldots 32$ and $i_2 = 1 \ldots 4$ and $6 \ldots 7$ as shown in Fig. 9d, are originally put in the corresponding k_1_Sets, when c_1 and c_2 are set to 1. When c_1 and c_2 increase, some of rules in ser S are relocated to some segments. As a result, the average and maximum numbers of rules in k_1_Set are reduced. On the other hand, the average and maximum numbers of rules in segments are not increased much. Thus, the search performance improves when c_1 and c_2 increase. For Firewall and IPC tables, no rule locates in tuples of (i_1, i_2) and (i_2, i_1) for $i_1 = 8, \ldots, 32$ and $i_2 = 3, \ldots, 7$ as shown in Figs. 9e and 9f. Thus, when k_1 and k_2 are fixed, setting c_1 and c_2 to larger values will not have any effect on performance. Now, we study why increasing k_1 and k_2 is useful for improving the search speed. As stated earlier, the rules belonging to region D in the 2D length distributions will not be affected when changing the values of k_1 and k_2. As shown in Figs. 9d, 9e, and 9f, there are 96 percent, 78 percent, and 18 percent of the rules located in region D for ACL, IPC, and Firewall tables, respectively. Therefore, increasing k_1 and k_2 can redistribute

TABLE 8
Performance of $H\_2k(k_1, k_2, c_1, c_2)$ for a Random Table of 1,000 Rules

(k_1,k_2)	(c_1,c_2)	Avg. / Max # of rules in segment	Avg. / Max # of rules in k_1_Set	Avg. / Max # of rules in k_2_Set	# of rules in k_1k_2_Set	memory usage (MB)	Avg. search speed (clock cycles)	Best result
(4,4)	(1,1)	2.93/9	7.50/13	7.00/12	17	0.035	942	
	(2,2)	3.98/10	5.37/9	5.00/9	8	0.049	927	
	(4,4)	6.15/12	3.19/5	3.50/8	2	0.077	785	
	(8,8)	8.53/16	1.37/3	2.06/4	0	0.121	649	
(5,5)	(1,1)	0.69/4	4.25/11	4.03/8	30	0.035	901	
	(2,2)	1.08/7	3.53/10	3.37/7	17	0.045	862	
	(4,4)	1.85/8	2.56/7	2.41/5	8	0.060	785	
	(8,8)	3.54/9	1.53/4	1.69/5	2	0.088	708	
(6,6)	(1,1)	0.16/3	2.48/7	2.28/6	45	0.047	834	
	(2,2)	0.34/4	2.08/7	1.87/6	30	0.051	818	
	(4,4)	0.69/7	1.72/6	1.55/5	17	0.059	802	
	(8,8)	1.34/8	1.25/4	1.11/4	8	0.075	728	√
(7,7)	(1,1)	0.04/2	1.44/5	1.25/6	55	0.095	866	
	(2,2)	0.09/3	1.19/4	1.12/6	45	0.096	800	
	(4,4)	0.26/3	1.01/4	0.91/5	30	0.099	771	
	(8,8)	0.59/5	0.83/4	0.75/5	17	0.107	755	
(8,8)	(1,1)	0.008/2	0.78/4	0.68/4	72	0.287	811	
	(2,2)	0.07/2	0.69/3	0.59/4	55	0.285	783	
	(4,4)	0.12/3	0.58/3	0.52/4	45	0.285	758	
	(8,8)	0.28/3	0.48/3	0.43/3	30	0.289	736	

TABLE 9
Performance of Schemes H_2k, H_2k_opt for the Random Rule Tables

Random Table Size		1,000	2,000	3,000	5,000	8,000	10,000
Search time (clock cycles)	$H\_2k(k_1,k_2,c_1,c_2)$	(6,6,8,8) 728	(7,7,2,2) 980	(6,6,4,4) 1,149	(5,5,1,1) 1,532	(6,6,4,4) 1,549	(6,6,4,4) 2,132
	$H\_2k\_opt(k_1,k_2,c_1,c_2)$	(6,6,8,8) 670	(6,6,4,4) 902	(6,6,4,4) 1,115	(5,5,1,1) 1,530	(7,7,4,4) 1,520	(6,6,4,4) 2,127
Memory usage (MB)	$H\_2k(k_1,k_2,c_1,c_2)$	(6,6,8,8) 0.075	(7,7,2,2) 0.136	(6,6,4,4) 0.190	(5,5,1,1) 0.212	(6,6,4,4) 0.582	(6,6,4,4) 0.833
	$H\_2k\_opt(k_1,k_2,c_1,c_2)$	(6,6,8,8) 0.075	(6,6,4,4) 0.118	(6,6,4,4) 0.190	(5,5,1,1) 0.212	(7,7,4,4) 0.537	(6,6,4,4) 0.833

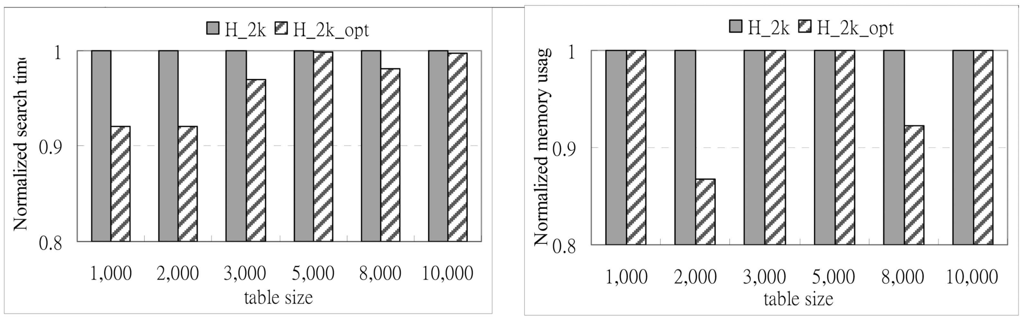


Fig. 12. Normalized performance of schemes H_2k, H_2k_opt for random tables.

much more rules in Firewall tables from k_1_Set, k_2_Set, and k_1k_2_Set to segments than ACL tables. As a result, setting k_1 and k_2 to large values always improves the search speeds for Firewall and IPC tables. However, the drawback of large k_1 and k_2 is the higher memory requirement because of rule duplications. In other words, $H\_2k(8, 8, c_1, c_2)$ always consumes much more memory than other settings. Thus, $H\_2k(7, 7, 1, 1)$ instead of $H\_2k(8, 8, 1, 1)$ is selected as the best setting for Firewall tables because $H\_2k(8, 8, 1, 1)$ has a much higher memory requirement. IPC tables have a similar performance trend as Firewall tables.

6.3 Update Speed

In addition to the demands for high search speed and low memory requirement, the multidimensional PC algorithms require a fast rule table update performance. However, it is not easy to come up with a fast update algorithm based on the data structures of the existing PC schemes because the rule duplication and data structure precomputation are usually needed. For example, in EGT, HiCut, and HyperCuts, if a rule contains wildcards in the dimensions selected for cutting, then it will be duplicated in all the rule subsets partitioned by the algorithms. As a result, when inserting or deleting this kind of rule, the data structures of many involved rule subsets have to be updated, making the update performance slower. Although the HyperCuts scheme extracts all the rules containing wildcards in both the IP source and the destination fields into a set called *W-Set*, and the remaining rules are put in another set called the *R-Set* for independent decision tree processing,

TABLE 10
Performance of Schemes H_2k and H_2k_opt

		Table Size	1,000	2,000	3,000	5,000	8,000	10,000
ACL	Search time (clock cycles)	H_2k	(4,4,8,8) 695	(4,4,8,8) 731	(4,4,4,4) 792	(4,4,1,1) 854	(6,6,2,2) 855	(7,7,2,2) 864
		H_2k_opt	(5,5,1,1) 405	(4,4,2,2) 390	(5,5,8,8) 437	(4,4,1,1) 442	(6,6,2,2) 444	(5,5,2,2) 483
	Memory usage (MB)	H_2k	(4,4,8,8) 0.024	(4,4,8,8) 0.045	(4,4,4,4) 0.060	(4,4,1,1) 0.085	(6,6,2,2) 0.188	(7,7,2,2) 0.326
		H_2k_opt	(5,5,1,1) 0.022	(4,4,2,2) 0.033	(5,5,8,8) 0.063	(4,4,1,1) 0.085	(6,6,2,2) 0.188	(5,5,2,2) 0.266
FW	Search time (clock cycles)	H_2k	(7,7,1,1)					
			581	678	661	803	867	817
		H_2k_opt	(7,7,1,1)					
			541	601	598	731	774	761
	Memory usage (MB)	H_2k	(7,7,1,1)					
			0.089	0.117	0.144	0.196	0.277	0.330
		H_2k_opt	(7,7,1,1)					
			0.089	0.117	0.144	0.196	0.277	0.330
IPC	Search time (clock cycles)	H_2k	(7,7,1,1)		(6,6,1,1)			
			818	903	1044	1109	1320	1124
		H_2k_opt	(7,7,1,1)		(6,6,1,1)			
			494	544	581	587	706	585
	Memory usage (MB)	H_2k	(7,7,1,1)		(6,6,1,1)			
			0.089	0.112	0.085	0.127	0.197	0.233
		H_2k_opt	(7,7,1,1)		(6,6,1,1)			
			0.089	0.112	0.085	0.127	0.197	0.233

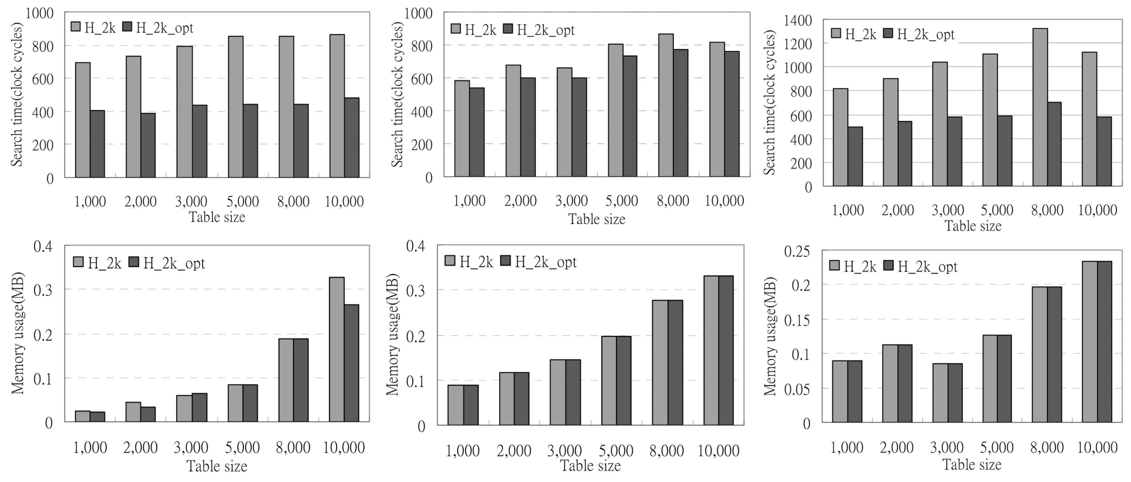


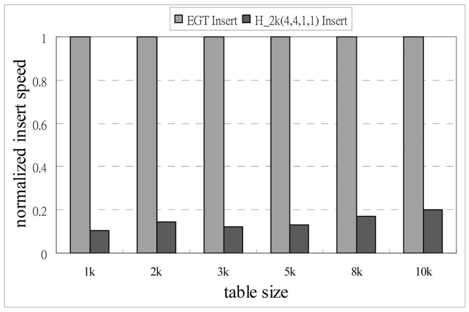
Fig. 13. Histogrammed performance of schemes H_2k and H_2k_opt.

rule duplication still exists, and thus, the update problem is not solved completely. EGT avoids the rule duplication by precomputing the *switch pointers.* However, precomputation also results in slow update performance. In HiCut and HyperCuts, precomputation is also needed to provide better cutting (i.e., balanced decision tree). The decision tree built without precomputation will have a longer search path and will thus slow down the search performance. In scheme $H\_2k(k_1, k_2, c_1, c_2)$, we can control the degree of rule duplication by setting different values for c_1 and c_2. Obviously, when we set $c_1 = c_2 = 1$, there is no rule duplication across the segments, k_1_Set, k_2_Set, and $k_1k_2\_Set$. Since the rule duplication only occurs inside a segment, k_1_Set, k_2_Set, or $k_1k_2\_Set$ that is usually small, the impact on update performance will be minimal.

To quantitatively evaluate the update performance, we conduct the experiments as follows: We first randomly extract 10 percent of the rules from the classifier and build the search structure according to the remaining 90 percent of the rules. We then randomly insert the extracted rules into the search structure to obtain the insertion time. After that, we randomly select another 10 percent of rules from the classifier and delete them from the search structure to obtain the deletion times. The average of insertion and deletion times is obtained as the update time. Because the available source codes of EGT, EGT-PC, and HyperCuts in [7] and [13] do not support rule updates, we only compare the update performance of the proposed scheme $H\_2k(k_1, k_2, c_1, c_2)$ with our implementation of EGT with the support of update. As shown in Table 11 and Fig. 14, the insertion and deletion performance of the scheme $H\_2k(k_1, k_2, c_1, c_2)$ are about six to seven and five to nine times better than EGT, respectively.

TABLE 11
Update Performance of Schemes $H\_2k(k_1, k_2, c_1, c_2)$ and EGT (in Clock Cycles)

Table	1,000	2,000	3,000	5,000	8,000	10,000
EGT Insert	677,393	735,698	1,538,633	2,638,801	3,759,670	4,295,641
H_2k(*4,4,1,1*) Insert	69,740	107,180	186,840	347,747	642,310	867,908
EGT Delete	700,441	763,356	1,601,803	2,758,986	3,913,893	4,502,034
H_2k(*4,4,1,1*) Delete	56,356	89,257	167,530	329,500	621,438	844,828



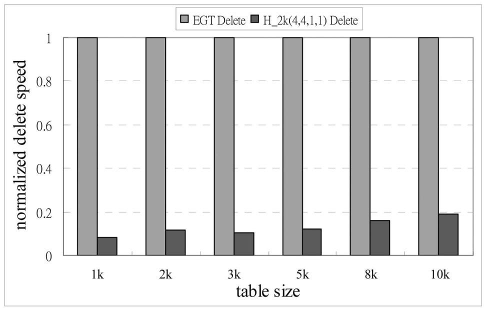


Fig. 14. Normalized update performance of schemes $H\_2k(4,4,1,1)$ and EGT.

7 Conclusions

In this paper, we have proposed a set of efficient algorithms for multidimensional packet classification. All the proposed algorithms are based on the hierarchical data structures of binary range and prefix searches augmented with a multidimensional segmentation table. The experimental results showed that our scheme outperforms other schemes not only in terms of classification speed and memory usage but also in terms of update speed.

References

[1] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is There an Alternative to CAMs?" *Proc. IEEE INFOCOM,* 2003.
[2] F. Baboescu and G. Varghese, "Scalable Packet Classification," *Proc. ACM SIGCOMM,* 2001.
[3] Y.-K. Chang, "Fast Binary and Multiway Prefix Searches for Packet Forwarding," *Computer Networks,* vol. 51, no. 3, pp. 588-605, Feb. 2007.
[4] Y.-K. Chang and Y.-C. Lin, "Dynamic Segment Trees for Ranges and Prefixes," *IEEE Trans. Computers,* vol. 57, no. 6, pp. 769-784, June 2007.
[5] E. Cohen and C. Lund, "Packet Classification in Large ISPs: Design and Evaluation of Decision Tree Classifiers," *Proc. ACM SIGMETRICS/Performance '05,* June 2005.
[6] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire Speed Packet Classification without TCAMs: A Few More Registers (and a Bit of Logic) Are Enough," *Proc. ACM SIGMETRICS '07,* June 2007.
[7] *EGT and EGT-PC,* http://www.ial.ucsd.edu/classification/, 2008.
[8] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. IEEE INFOCOM '00,* pp. 1193-1202, Mar. 2000.
[9] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE INFOCOM '98,* pp. 1240-1247, Apr. 1998.
[10] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. ACM SIGCOMM '99,* pp. 147-160, Aug. 1999.
[11] P. Gupta and N. McKeown, "Packet Classification Using Hierarchical Intelligent Cuttings," *Proc. Seventh Hot Interconnects (HOTI '99),* Aug. 1999.
[12] P. Gupta and N. Mckeown, "Algorithms for Packet Classification," *IEEE Network,* vol. 15, no. 2, pp. 24-32, 2001.
[13] *Hypercut,* http://www.arl.wustl.edu/~hs1/project/hypercuts.htm, 2008.
[14] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," *Proc. ACM SIGCOMM '98,* pp. 203-214, 1998.
[15] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Trans. Networking,* vol. 7, no. 3, pp. 324-334, June 1999.
[16] H. Lu and S. Sahni, "$O(\log n)$ Dynamic Router-Tables for Prefixes and Ranges," *IEEE Trans. Computers,* vol. 53, no. 10, pp. 1217-1230, Oct. 2004.
[17] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network,* vol. 15, no. 2, pp. 8-23, Mar./Apr. 2001.
[18] S. Sahni and K. Kim, "$O(\log n)$ Dynamic Router-Table Design," *IEEE Trans. Computers,* vol. 53, no. 3, pp. 351-363, Mar. 2004.
[19] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," *Proc. ACM SIGCOMM '03,* pp. 25-29, Aug. 2003.
[20] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," *Proc. ACM SIGCOMM '99,* pp. 135-146, Aug. 1999.
[21] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvagel, "Fast and Scalable Layer Four Switching," *Proc. ACM SIGCOMM '98,* pp. 191-202, Aug. 1998.
[22] D.E. Taylor and J.S. Turner, "ClassBench: A Packet Classification Benchmark," *Proc. IEEE INFOCOM '05,* May 2005.
[23] D.E. Taylor, "Surveys and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys,* vol. 37, no. 3, pp. 238-275, Sept. 2005.
[24] T. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," *Proc. IEEE INFOCOM,* 2000.
[25] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," *Proc. IEEE INFOCOM '03,* May 2003.

Yeim-Kuan Chang received the MS degree in computer science from the University of Houston at Clear Lake in 1990 and the PhD degree in computer science from Texas A&M University, College Station, in 1995. He is currently an associate professor in the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan. His research interests include Internet router design, computer networking, computer architecture, and multiprocessor systems. He is a member of the IEEE Computer Society.