CHAPTER 3

PACKET CLASSIFICATION

3.1 INTRODUCTION

Traditionally, Internet routers only provide best effort service by processing each incoming packet in the same manner. With the emergence of new applications, Internet Service Providers (ISPs) would like routers to provide different QoS levels to different applications. To meet these QoS requirements, routers need to implement new mechanisms, such as admission control, resource reservation, per-flow queuing, and fair scheduling. However, a prerequisite to deploying these mechanisms is that the router is able to distinguish and classify the incoming traffic into different flows. We call such routers flow-aware routers. A flow-aware router is distinguished from a traditional router in that it is capable of keeping track of flows passing by and applying different classes of service to each flow.

Flows are specified by *rules* and each rule consists of operations comparing packet fields with certain values. We call a set of rules a *classifier*, which is formed based on the criteria to be applied to classify packets with respect to a given network application. Given a classifier defining packet attributes or content, packet classification is the process of identifying the rule or rules within this set to which a packet conforms or matches [1]. To illustrate the kinds of services that could be provided by a flow-aware router with packet classification capability, we use an example classifier shown in Table 3.1. Assume this classifier is stored in the router R in the example network shown in Figure 3.1.

High Performance Switches and Routers, by H. Jonathan Chao and Bin Liu
Copyright © 2007 John Wiley & Sons, Inc.

TABLE 3.1 Classifier Example

	Network-Layer		Transport-Layer			
Rule	Destination	Source	Protocol	Destination	Application-Layer Protocol	Action
R_1	128.238/16	*	TCP	= telnet	*	Deny
R_2	176.110/16	196.27.43/24	UDP	*	RTP	Send to port III
R_3	196.27.43/24	134.65/16	TCP	*	*	Drop traffic if rate > 10 Mbps
R_4	*	*	*	*	*	Permit

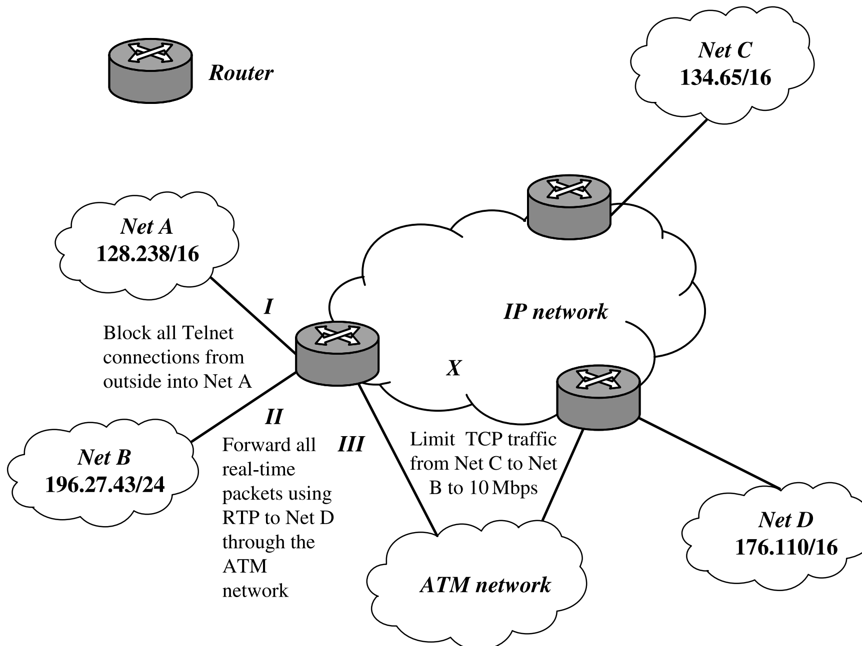


Figure 3.1 Network example with classifier.

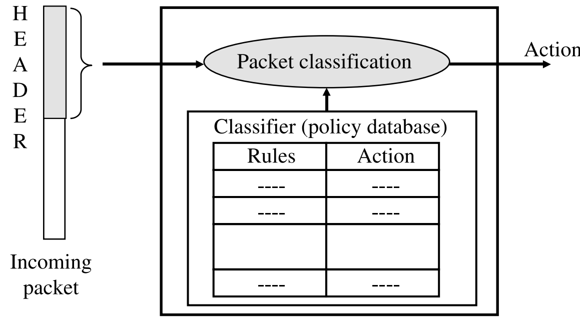
With only four rules in the example classifier, the router X provides the following services:

Packet Filtering. Rule R_1 blocks all telnet connections from outside into Net A, which may be a private research network.

Policy Routing. Rule R_2 enables the router to forward all real-time traffic using real-time transport protocol (RTP) in the application layer from Net B to Net D through the ATM network at the bottom of Figure 3.1.

Traffic Policing. Rule R_3 limits the total transmission control protocol (TCP) traffic rate from Net C to Net B up to 10 Mbps.

A formal description of the rule, classifier, and packet classification is given in the work of Lakshman and Stiliadis [2]. We will use these symbols and terminologies throughout this chapter.



Matching the packet header to the rules in the classifier

Figure 3.2 Packet classification [11].

1. A classifier C consists of N rules, R_j, $1 \leq j \leq N$, where R_j is composed of three entities:
   (a) A regular expression $R_j[i]$, $1 \leq i \leq d$, on each of the d header fields of a packet.
   (b) A number, $Pri(R_j)$, indicating the priority of the rule in the classifier.
   (c) An action, referred to as $Action(R_j)$.
2. An incoming packet P with the header considered as a d-tuple $(P_1, P_2, \ldots, P_d)$ is said to match R_j, if and only if, P_i matches $R_j[i]$, where $1 \leq i \leq d$.
3. Given an incoming packet P and thus the d-tuple, the d-dimensional packet classification problem is to find the rule R_m with the highest priority among all the rules R_j matching the d-tuple. As shown in Figure 3.2, a packet header, consisting of IP source address (32 bits), destination address (32 bits), source port number (16 bits), destination port number (16 bits), and protocol type (8 bits)[1], is used to match the rule(s) in the classifier. The one with the highest priority is chosen and its corresponding action is applied to the packet.

In the example classifier shown in Table 3.1, each rule has five regular expressions on five packet-header fields from network layer to application layer. Each expression could be a simple *prefix/length* or *operator/number* specification. The prefix/length specification has the same definition as in IP lookups, while the operator/number could be more general, such as equal 23, range 256–1023, and greater than 1023. Furthermore, a wildcard is allowed to be inserted to match any value. Note that R_4 in Table 3.1 matches with any incoming packet due to its 'all-wildcards' specification, which means the priorities of rules take effect when a packet matches both R_4 and the other rules.

Suppose there is a rule set $C = R_j (1 \leq j \leq N)$ and each rule R_j has d fields. The fields are labeled as $F_i (1 \leq i \leq d)$ and R_j is denoted as $\langle R_{j1}, R_{j2}, \ldots, R_{jd} \rangle$. Table 3.2 shows an example classifier with seven rules in four fields. The first two fields, F_1 and F_2, are specified in prefixes and the last two fields, F_3 and F_4, are specified in ranges. The last column shows the action associated with the rules. F_1 and F_2 can be handled more efficiently by using tries or TCAM as in Chapter 2. On the other hand, F_3 and F_4 can be handled more efficiently

[1] Herein, the classification is for IPv4 scenario only.

TABLE 3.2 Example Classifier with Seven Rules in Four Fields

Rule	F_1	F_2	F_3	F_4	Action
R_1	00*	110*	6	(10, 12)	Act_0
R_2	00*	11*	(4, 8)	15	Act_1
R_3	10*	1*	7	9	Act_2
R_4	0*	01*	10	(10, 12)	Act_1
R_5	0*	10*	(4, 8)	15	Act_0
R_6	0*	1*	10	(10, 12)	Act_3
R_7	*	00*	7	15	Act_1

by projecting the numbers into different ranges and then performing range lookup, to be described in later sections of this chapter. The seven rules are listed in the order of descending priorities, that is, R_1 has the highest priority. This rule set will be used to illustrate some of the algorithms described later.

Several performance metrics [3] are used to compare and analyze packet classification algorithms:

Search Speed. High-speed links require fast classification. For example, assuming a minimum-sized 40-byte IP packet, links running at 10 Gbps can carry 31.25 million packets per second (mpps). The classification time is limited to 32 ns.

Storage Requirement. Small storage means fast memory access speed and low power consumption, which are important for cache-based software algorithms and SRAM-based hardware algorithms.

Scalability in Classifier Size. The size of the classifier depends on the applications. For a metro/edge router performing microflow recognition, the number of flows is between 128k and 1 million. Obviously, this number increases as the link speed increases.

Scalability in the Number of Header Fields. As more complex services are provided, more header fields need to be included.

Update Time. When the classifier changes, such as an entry deletion or insertion, the data structure needs to be updated. Some applications such as flow-recognition require the updating time to be short. Otherwise, the performance of classification is degraded.

Flexibility in Specification. The ability of an algorithm to handle a wide range of rule specifications, such as prefix/length, operator/number, and wildcards, enables it to be applied to various circumstances.

Linear search is the simplest algorithm for packet classification. The rule set can be organized into an array or a linked list in order of increasing costs. Given an incoming packet header, the rules are examined one by one until a match is found. For a N-rule classifier, both the storage and query time complexity are $O(N)$, making this scheme infeasible for large rule sets.

Many efficient packet classification schemes have been proposed and are described in the following sections.

3.2 TRIE-BASED CLASSIFICATIONS

3.2.1 Hierarchical Tries

Hierarchical trie is a simple extension of a one-dimension trie to a multiple-dimension trie, with each dimension representing a field. It is also called multi-level tries, backtracking-search tries, or trie-of-tries [3].

Rule Storing Organization. A two-dimensional hierarchical trie representing the first two fields of rule set C in Table 3.2 is shown in Figure 3.3. Here, we only consider F_1 and F_2 because they are prefixes and can be easily processed by using tries. The ellipse nodes belong to the F_1-trie and round nodes belong to F_2-tries. The bold curved arrow denotes the next-trie pointer. Note that there are four F_2-tries because we have four distinct prefixes in the F_1 field of C. Each gray node is labeled with a rule R_j, which means that if this node is reached during a search, R_j is matched. In general, the hierarchical trie can be constructed as follows: a binary radix trie, called F_1-trie is first built for the set of prefixes $\{R_{j1}\}$ that belong to F_1 of all the rules. Secondly, for each prefix p in the F_1-trie, a $(d-1)$-dimensional hierarchical trie T_p is recursively constructed for those rules that exactly specify p in F_1, that is, the set of rules $\{R_j | R_{j1} = p\}$. Trie T_p is connected to p by a next-trie pointer stored in node p.

Classification Scheme. Classification for an incoming packet with the header $(v_1, v_2, \ldots, v_d)$ should be carried out in the following procedure: The query algorithm traverses the F_1-trie based on v_1; if a next-trie pointer is encountered, the algorithm goes on with the pointer and queries the $(d-1)$-dimensional hierarchical trie recursively.

For the above rule set C, given an incoming packet (001, 110), the search process starts from the F_1-trie to find the best matching prefix of '001'. After node 'D' in the F_1-trie is reached, the next-trie pointer is used to guide the search into the F_2-trie to find all matching prefixes of '110'. Apparently, both node R_1 and node R_2 are reached; however, only R_1 is recorded due to its higher priority. Now the search process backtracks to node 'B', which is the lowest ancestor of node 'D' in the F_1-trie. Again, we use the next-trie pointer here to

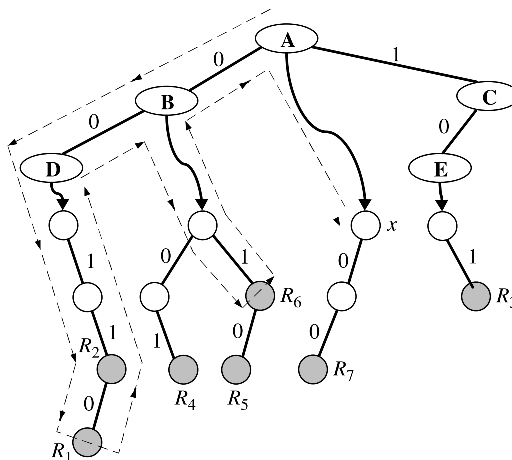


Figure 3.3 Hierarchical trie data structure for F_1 and F_2 of the rule set in Table 3.2.

search the F_2-trie. This procedure is repeated until no ancestor node of node 'D' is available to be searched. In this example, the search process ends up at node x and the entire traversing path is depicted by the dashed line in Figure 3.3. During this traversal, three matches are found, R_1, R_2, and R_6. R_1 is returned as the highest priority rule matched. The backtracking process is necessary since '001' of the incoming packet may match several prefixes in the first field and we have no knowledge in advance which F_2-trie contains prefix(es) that match '110'. Furthermore, all matches must be found to ensure that the highest priority one is returned.

Performance Comments. Hierarchical trie is one of most storage-economic algorithms. For a N-rule set, each of which is with d sub-fields and the maximum field length of each field is W, then the storage complexity is $O(dW)$. The data structure is straightforward and easy to maintain at the expenses of a longer searching time. Traversing the trie brings backtracking in an attempt to find all the matching rules since the priority level cannot be effectively reflected by this data structure. The search time complexity is $O(W^d)$. F_d-trie has a depth of W and thus takes $O(W)$ to search. F_{d-1}-trie also has a depth of W, where each node has a F_d-trie. The worst-case search time for the F_{d-1}-trie is thus $O(W^2)$. With induction, the time complexity becomes $O(W^d)$. Incremental updates can be implemented in $O(d^2W)$ because each field of the updated rule is stored in exactly one location at F maximum depth $O(dW)$.

3.2.2 Set-Pruning Trie

The set-pruning trie is a modified version of the hierarchical trie. Backtracking is avoided during the query process in a set-pruning trie.

Rule Storing Organization. In a set-pruning trie, each trie node (with a valid prefix) duplicates all rules in the rule sets of its ancestors into its own rule set and then constructs the next dimension trie based on the new rule set.

An example of the 2-dimensional set-pruning trie denoting F_1 and F_2 fields of the rule set C (Table 3.2) is shown in Figure 3.4. Note that in Figure 3.3 the rule sets of F_1-trie node A, B, and D are $\{R_7\}$, $\{R_4, R_5, R_6\}$, and $\{R_1, R_2\}$, respectively. While in Figure 3.4, they are $\{R_7\}$, $\{R_4, R_5, R_6, R_7\}$, and $\{R_1, R_2, R_4, R_5, R_6, R_7\}$, where rules have been duplicated.

Classification Scheme. The search process for a d-tuple consists of d consecutive longest prefix matching on each dimension of the set-pruning trie. Given a 2-tuple $(001, 110)$, the query path is depicted by the dashed line in Figure 3.4. R_1 is returned as the highest priority rule matched. Multiple rules may be encountered along the path and the one with the highest priority is recorded. The R_2 node on the path is supposed to include rules R_2 and R_6, but only R_2 is kept due to its higher priority.

Performance Comments. Hierarchical trie needs backtracking because the rule sets associated with the F_1-trie nodes are disjointed with each other. The set-pruning trie eliminates this need and decreases the query time complexity to only $O(dW)$ at the cost of increased storage complexity, $O(N^d dW)$, since a rule may need to be duplicated up to N^d times. The update complexity is also $O(N^d)$.

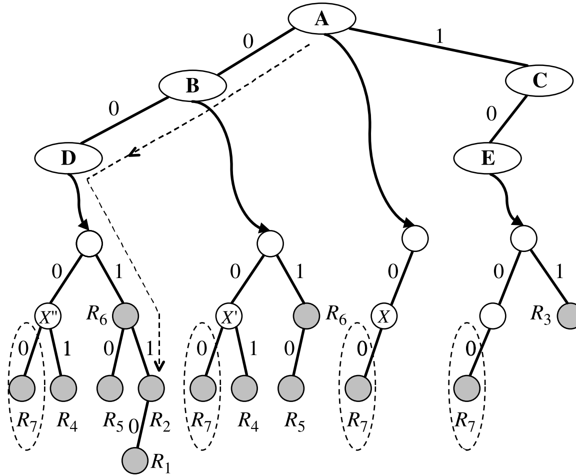


Figure 3.4 Set-pruning trie data structure for the rule set in Table 3.2.

3.2.3 Grid of Tries

Srinivansan et al. [4] proposed the grid-of-tries data structure for 2D (2-dimensional) classification, which reduces the storage complexity to $O(NdW)$, as in the hierarchical trie, while still keeping the query time complexity at $O(dW)$ by pre-computing and storing the so-called switching pointers in some F_2-trie nodes. It is mentioned above that the F_1-trie node of the set-pruning trie duplicates rules belonging to its ancestors. This procedure could also be interpreted that the F_1-trie node merges the F_2-tries of its ancestors into its own F_2-trie. For instance, R_7 in F_2-trie of node A in Figure 3.4 is duplicated three times. Assuming that the F_2-trie belonging to node B is denoted as F_2-B-trie, the only difference between the two F_2-B-tries in Figures 3.3 and 3.4 is that node R_7 is duplicated in the set-pruning trie. Now instead of node duplication, a switching pointer labeled with '0' is incorporated at node x' and points to node R_7 in the F_2-A-trie as shown in Figure 3.5. The switching pointers are depicted by the dashed curved arrows. In fact, the switching pointer labeled '0' at node x' replaces the 0-pointer in the set-pruning trie.

If the hierarchical trie and set-pruning trie have been built for a classifier C, the grid-of-tries structure of C could be constructed by adding switching pointers to the F_2-tries of the hierarchical trie with comparison to that of the set-pruning trie. A switching pointer, p_s, labeled with $0/1$ is inserted at node y whenever its counterpart in the set-pruning trie contains a $0/1$-pointer to another node z while y does not. Node z may have several counterparts in the hierarchical trie, but p_s points to the one contained in the F_2-trie that is 'closest' to the F_2-trie containing node y. For instance, node x and node x' in Figures 3.4 and 3.5 are both counterparts of node x'' in Figure 3.4. However, the switching pointer at node y points to node x' since node B is closer to node D than node A. If the switching pointers are viewed the same as $0/1$-pointers, the query procedure is identical as in the set-pruning trie.

The grid-of-tries structure performs well on both query time and storage complexity, but incremental updates are complex since several pointers may point to a single node. If the node is to be removed, a new node needs to be created and the pointers need to be updated to point to the new node.

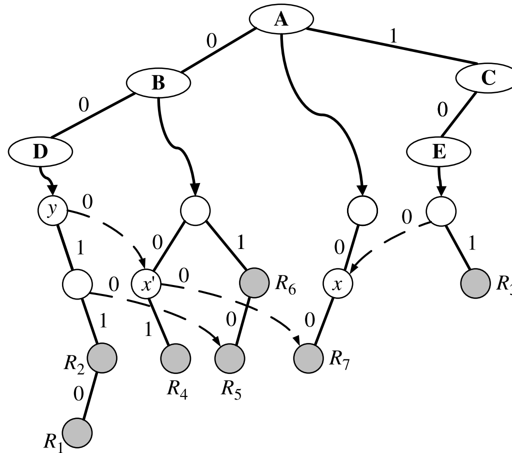


Figure 3.5 Example of the grid-of-tries structure for the rule set in Table 3.2.

3.2.4 Extending Two-Dimensional Schemes

Baboescu et al. [5] introduced a novel classification method EGT-PC for core routers. The key idea was to make use of the characteristics of rule databases they discovered in core routers to reduce the complexity of multi-dimensional search to that of a 2D search. By observing statistics from classifiers in Tier 1 ISP's core routers, they found that every packet matches at most a few distinct source–destination prefix pairs (SIP, DIP) presented in the rule set. In other words, if we project the rule set to just the source and destination fields, no packet matches more than a small number of rules in the new set of projected rules. Note that this is emphatically not true for single fields because of wildcards: a single packet can match hundreds of rules when considering any one field in isolation. Based on this character, they present the idea of a simple 2D classification method, as shown in Figure 3.6.

The idea is first to use any efficient 2D matching scheme to find all the distinct source–destination prefix pairs $(S_1, D_1), \ldots, (S_t, D_t)$ that match a header. For each distinct pair (S_i, D_i) there is a linear array or list with all rules that contain (S_i, D_i) in the source and

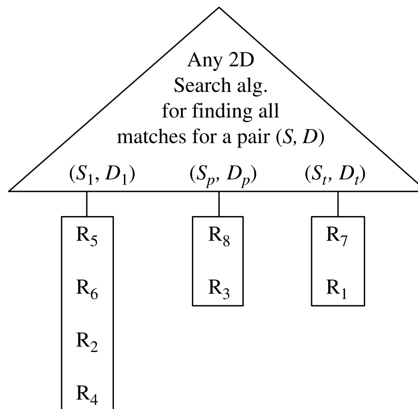


Figure 3.6 Extended 2D search policy [5].

destination fields. As shown in Figure 3.6 (S_1, D_1) contain rules, R_5, R_6, R_2, and R_4. Note that a rule can only be associated with a source–destination prefix pair. On the other hand, one may wish to replicate rules to reduce the number of source–destination prefix pairs considered during the search to reduce the searching time. When searching for a rule for a given key, multiple source–destination prefix pairs can match with the key. For instance (*, 000) and (1*, 0*) match with a prefix key (111, 000). As a result, the rules in each matched (S, D) pair will be further searched against with the rest part of the key. For example, if (S_1, D_1) is matched, all its rules, R_5, R_6, R_2, and R_4, are searched against with, for instance, the port numbers of the key.

3.2.5 Field-Level Trie Classification (FLTC)

A field-level trie classification (FLTC) uses a field-level trie (FLT) structure that is organized in a hierarchical structure field by field [6]. The classification data structure has been optimized so that TCAM and multiway search are deployed for prefix and range fields, respectively. The query (search) process is also carried out field by field. With proper implementation, each query only requires a few memory accesses on average, and thus very high-speed classification can be achieved. The storage requirement of the FLTC is reasonable due to the node-sharing property of the FLT. Although node sharing makes the updating processes (insertion and deletion) less straightforward, the complexity of the update operation remains low because each operation only affects a small part of the data structure. The FLTC can easily support large classifiers, for example, with 100,000 to 1 million rules, without compromising query performance.

The FLT structure targets classifiers with multiple fields, each one of which is specified in either prefix format or range format. Figure 3.7 shows the FLT constructed from the classifier in Table 3.2. The FLT is defined to have the following properties:

1. It is organized in a hierarchical structure field by field. The depth of an FLT equals the number of fields, d. In Figure 3.7, there are four levels of nodes, organized from F_1 to F_4.[2]
2. Each node in the FLT contains a rule set, which is also a subset of its parent node's rule set. The root node of the FLT is defined to contain all the rules in the classifier.
3. Node a in the ith level[3] generates its child nodes in the $(i + 1)$th level based on the F_i values of all the rules contained in node a. Depending on F_i's specification, there are two different procedures for child-node generation:
   - If F_i is specified in prefix format, the number of child nodes of a equals the number of different prefixes contained in the F_i field of a's rule set. Each child node is associated with a different prefix. Assuming that child node b is associated with prefix p, the F_i value of rule r contained in b's rule set is either the same as or a prefix of p. For instance, the root node in Figure 3.7 contains all seven rules and there are four different prefixes, *, 0*, 00*, and 10*, in field F_1, so four child nodes are generated. Node x associated with prefix 0* contains four rules, R_4–R_7. The F_1 value of R_4–R_6 is 0*, which is the associated prefix; the F_1 value of R_7 is *, which is a prefix of 0*.

[2]Note that the gray nodes in the bottom do not form a separate level. They are shown only to indicate which rule is matched when the query (classification) process terminates at the fourth level.
[3]The root node is defined to be in the first level.

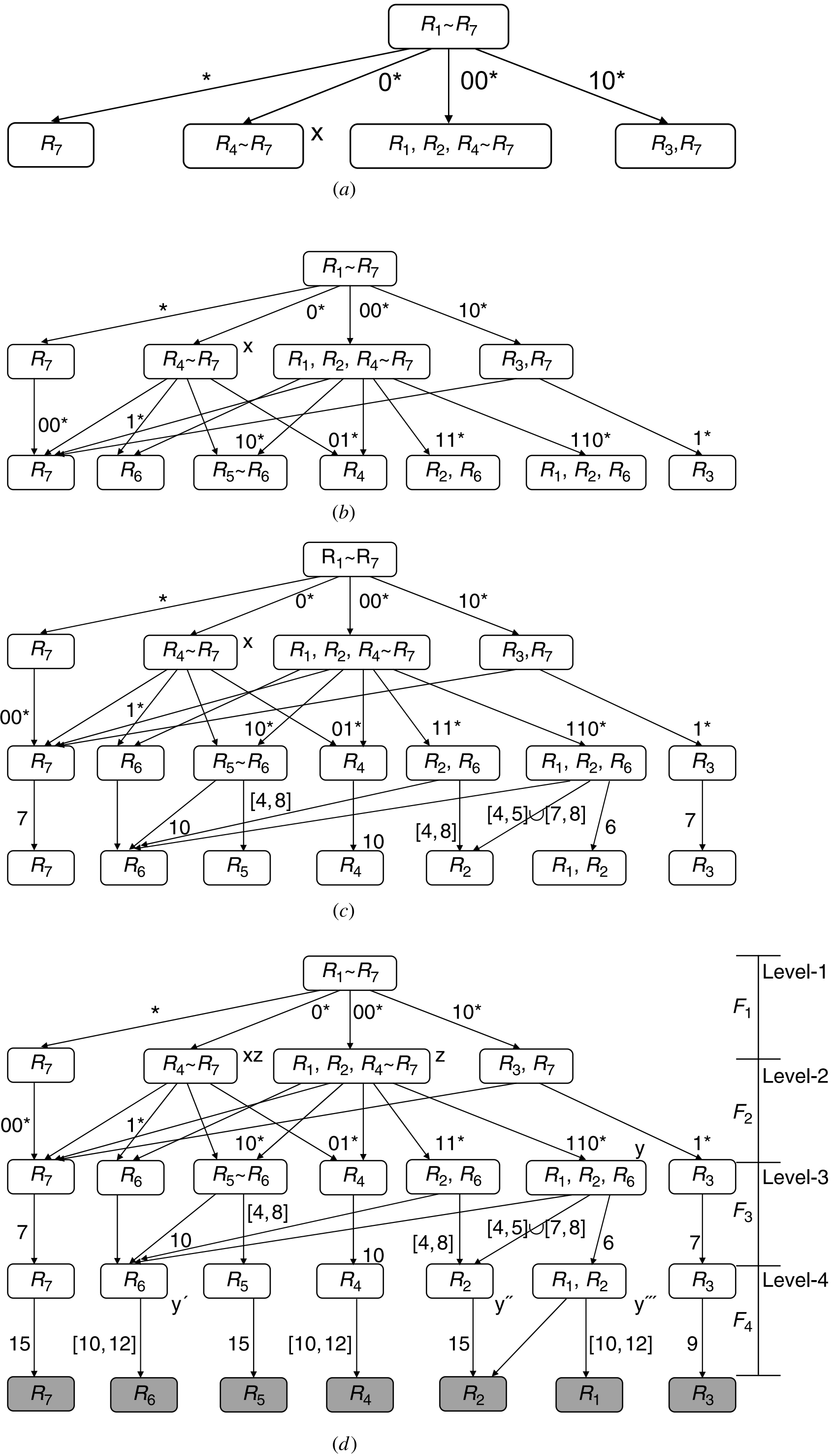


Figure 3.7 Example of the four-dimensional FLT. (*a*) Level 1; (*b*) Level 2; (*c*) Level 3; (*d*) Level 4.

- If F_i is specified in range format, we first project all the ranges (taken from the F_i fields of a's rule set) onto a number line and obtain a set of intervals. For each interval I, a child node b is generated. A rule r is contained in b's rule set if, and only if, the range specified by the F_i field of r covers I. For instance, node y generates three child nodes, node y' with interval [10, 10], which is a single point, node y''' with interval [6, 6], and node y'' with intervals [4, 5] and [7, 8] (in fact, there are two pointers both pointing to y''), as indicated in Figure 3.7.

4. The rule set of a node a in the ith level is unique among the rule sets of all the nodes in the ith level. If two nodes in the $(i-1)$th level, b and c, have a child node a in common, then only one node, which is a, is generated and they share it. Figure 3.7 shows that node sharing happens when a node is pointed to by multiple pointers.

Fields in Prefix Form. Since each field is normally specified in either the prefix or range forms and each specification has its own favored data structure and searching algorithms, we group the fields with the same specifications together. In most cases, a classifier has two groups of fields; the first group in prefix form and the second group in range form. In Table 3.2, F_1 and F_2 are in the first group and F_3 and F_4 are in the second. The FLT is organized so that fields of the first group appear in the upper levels and fields of the second one appear in the lower levels.

For the first group where only prefix fields exist, TCAM can be used for storing the prefixes and searching among them. Since TCAM can accommodate multiple fields simultaneously, the query of the first group of fields can be accomplished in a single TCAM access. Figure 3.8 shows the compressed FLT derived from the classifier in Table 3.2 and the trie structure in Figure 3.7. Now there is only one level existing for fields F_1 and F_2. The root node has seven child nodes initially lying in the third level in Figure 3.7. Each second-level node has an F_1/F_2 prefix pair associated with it. Each such prefix pair is the contents

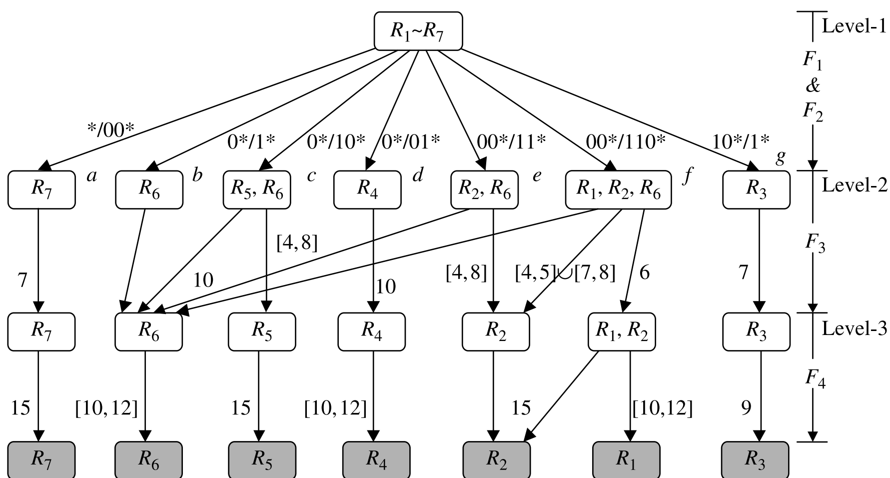


Figure 3.8 Example of the compressed four-dimensional FLT.

TABLE 3.3 TCAM Contents for the Compressed FLT

Entry #	Prefix Pair	Node Name	Sum of Lengths from Prefix Pair
1	*/00*	*a*	2
2	0*/1*	*b*	2
3	0*/10*	*c*	3
4	0*/01*	*d*	3
5	10*/1*	*g*	3
6	00*/11*	*e*	4
7	00*/110**	*f*	5

of an entry in the TCAM. The prefix pair is derived from the trie structure in Figure 3.7. For each node a in the third level in Figure 3.7, corresponding to the node in the second level in Figure 3.8, we find a path from the root node to a with the smallest sum of the prefix lengths. The prefixes along this path form the prefix pair associated with a in Figure 3.8. All prefix pairs are arranged in decreasing order of prefix length (the sum of the lengths of the two prefixes) in the TCAM. For prefix pairs with the same length, their relative order can be arbitrary. The contents of the TCAM for the compressed FLT in Figure 3.8 is shown in Table 3.3.

By arranging the prefix pairs in ascending order in the TCAM (meaning that the longest matching prefix pair will be found), we can guarantee the search result from the TCAM to be correct, for example, the appropriate node in the second level is determined to continue the entire query process. A brief proof follows.

When searching the TCAM with a key A/B, if two entries with prefix pairs A_1/B_1 and A_2/B_2 are matched, there will be two scenarios. Without loss of generality, we assume $A_1 \prec A_2$, meaning that A_1 is a prefix of A_2.

- In the first scenario, where $B_1 \prec B_2$, the length of A_2/B_2 is larger than that of A_1/B_1. Therefore, the entry with A_2/B_2 is output as the result. It is a correct result since all the rules in node a (corresponding to A_1/B_1) are also contained in node b (corresponding to A_2/B_2), which is guaranteed by the property of the FLT, and node b is selected.
- In the second scenario, where $B_2 \prec B_1$, another entry with prefix pair A_2/B_1 must exist, which is guaranteed by the generation process of the FLT. Since the length of A_2/B_1 is larger than that of both A_1/B_1 and A_2/B_2, the entry with A_2/B_1 is output as the result. It is a correct result because all rules in both node a and node b are contained in node c (corresponding to A_2/B_1).

The above conclusion can be easily extended to multiple prefix fields more than two. Given a packet header to be classified, the fields belonging to the first group are extracted and presented to the TCAM for searching. The output from the TCAM indicates a node in the second level to be accessed next. Since the TCAM has accommodated all the prefix fields, the rest of the query process relies on the range fields.

Fields in Range Specification. For the nodes existing in the second or lower levels, we propose using a multiway search tree (k-way search tree) to organize the data structure

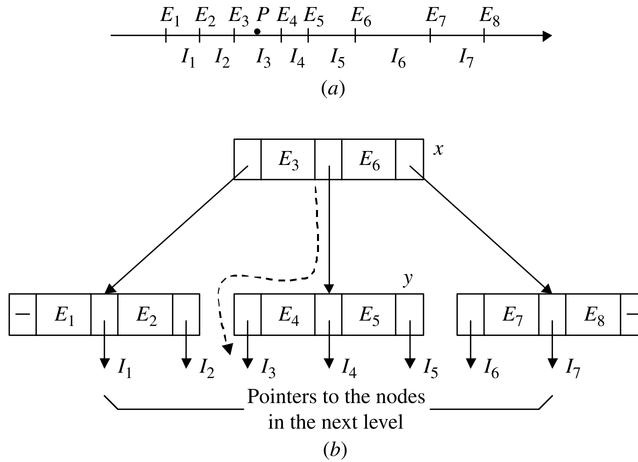


Figure 3.9 Example of the node structure organized in three-way search tree. (*a*) Derived intervals by range protection; (*b*) Node structure for a three-way search tree.

at each node. For example, there is a node a in the ith level ($i > 1$) of the compressed FLT. After projecting the F_i fields of the rules in a's rule set onto a number line, seven intervals, I_1 to I_7, are obtained with eight end points, E_1 to E_8, as shown in Figure 3.9*a*. If we use a three-way search tree to organize these intervals, the result is shown in Figure 3.9*b*. It is a two-layer tree with four blocks (to avoid confusion with the terms 'level' and 'node' in the FLT, we use the terms 'layer' and 'block' in the k-way search tree). Each block contains up to k pointers and $k - 1$ end points. The pointer in an internal block points to another block in the k-way search tree, while the pointer in a leaf block points to a $(i + 1)$th level node in the compressed FLT. We use an example to illustrate the searching process in the k-way search tree. Assuming point P exists in the interval I_3, the searching process starts from the root block x. By comparing P with the two end points, E_3 and E_6, stored in x, we know the order among them is $E_3 < P < E_6$. So the second pointer is followed to block y in the second layer. Similarly, by comparing P with the two end points, E_4 and E_5, we know that the first pointer associated with interval I_3 should be followed to a node in the next level of the compressed FLT.

The multiway search is an efficient algorithm for range lookup problems. The number of layers of a k-way search tree can be determined by $\log_k M$ where M is the number of intervals. From the implementation point of view, each block in the k-way search tree is a basic unit stored in memory, which requires one memory access for one read/write operation. Thus, during a search process, the number of memory accesses equals the number of layers of the k-way search tree, which is $\log_k M$. The number k here is limited by the block size, which is determined by the memory bandwidth.

The query process of an FLT starts from the TCAM for all the prefix fields. After reaching the range field, the query process proceeds one level (or one field) at a time and at each level a k-way search is performed to find the next-level node to be accessed. The query process terminates when a leaf node is reached and a matched rule (if it exists) is returned as the result.

3.3 GEOMETRIC ALGORITHMS

3.3.1 Background

As mentioned before, each field of a classifier can be specified in either a prefix/length pair or an operator/number form. From a geometric point of view, both specifications could be interpreted by a range (or interval) on a number line. Thus, a rule with two fields represents a rectangle in the 2D Euclidean space and a rule with d fields represents a d-dimensional hyper-rectangle. The classifier is a set of such hyper-rectangles with priorities associated. Given a packet header (d-tuple), it represents a point P in the d-dimensional space. The packet classification problem is equivalent to finding the highest priority hyper-rectangle that encloses P. Figure 3.10 gives the geometric representation of F_1 and F_2 of the classifier in Table 3.2 with rectangles overlapped according to their priorities. Given the point P(0010, 1100), it is straightforward to figure out the highest priority matching rule R_1.

There are several standard problems in the field of computational geometry that resemble packet classification [7]. One is the point location problem that is defined as finding the enclosing region of a point, given a set of non-overlapping regions. Theoretical bounds for point location in N (hyper-)rectangular regions and $d > 3$ dimensions are $O(\log N)$ time with $O(N^d)$ space, or $O((\log N)^{d-1})$ time with $O(N)$ space. Packet classification is at least as hard as point location since (hyper-)rectangles are allowed to overlap. This conclusion implies that the packet classification is extremely complex in the worst case.

Packet classification algorithms of this category always involve the range interpretation on certain fields of the classifier. If the prefixes or ranges in one field of a classifier are projected on the number line $[0, 2^W - 1]$, a set of disjoint elementary ranges (or intervals) is obtained and the concatenation of these elementary ranges forms the whole number line. For instance, there are four ranges on F_1 dimension and five ranges on F_2 dimension, as shown in Figure 3.10. Given a number Z on the number line, the range lookup problem is defined as locating the elementary range (or interval) containing Z. One way to locate the number on a range is to use the k-way search described in previous section. For simplicity, we use range (or interval) instead of elementary range (or interval) unless explicitly specified. It is clear that a prefix represents a range on the number line. On the other hand, an arbitrary

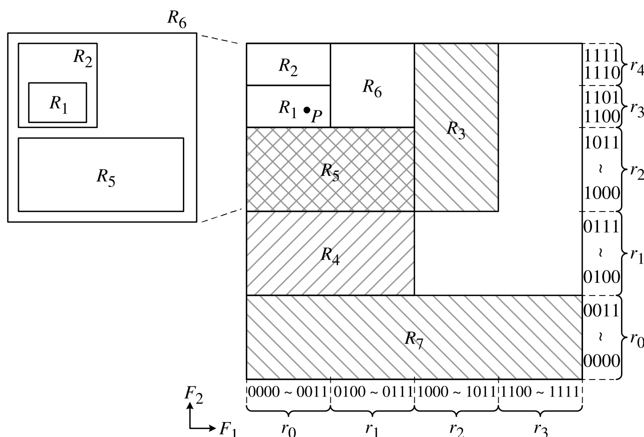


Figure 3.10 Geometric representation of the classifier in Table 3.2.

range may need up to $2W - 2$ prefixes for representation [3]. This is a useful conclusion for analyzing the increased storage complexity of some classification algorithms that only support prefix specification well. The process of transforming an arbitrary range into one or several prefixes is called range splitting.

3.3.2 Cross-Producting Scheme

Srinivansan et al. [4] proposed a cross-producting scheme that is suitable for an arbitrary number of fields and either type of field specification. The cross-producting scheme works by performing d range lookup operations, one on each field, and composing these results to index a pre-computed table that returns the highest priority rule matched.

Refer to F_1 and F_2 of classifier C of Table 3.2. For the first step, the rule specifications in the F_1 and F_2 fields are projected on two number lines vertical to each other and two sets of ranges $\{r_1[0], \ldots, r_1[3]\}$ and $\{r_2[0], \ldots, r_2[4]\}$, are obtained as shown in Figure 3.11. Each pair of ranges $(r_1[i], r_2[j])$, corresponds to a small rectangle with a pre-computed best matching rule written inside ('—' means no matching rule exists). The entire pre-computed table is shown in Figure 3.11 if we organize the table in a 2D matrix format. Thus, given a 2-tuple (p_1, p_2), two range lookups are performed on each range set and the two matching ranges returned are composed to index the pre-computed table. For instance, if $p_1 = 0010$ and $p_2 = 1100$, the two returned ranges $(r_1[0], r_2[3])$, tell us that R_1 is the best matching rule. Regarding the generic d-dimensional classifier, d sets of ranges are obtained by projecting the rule specification on each dimension and each item in the d-dimensional cross-product table could be pre-computed in the same way as the above example.

The cross-producting scheme has a good query time complexity of $O(d \cdot t_{RL})$, where t_{RL} is the time complexity of finding a range in one dimension. However, it suffers from a memory explosion problem; in the worst case, the cross-product table can have $O(N^d)$ entries. Thus, an on-demand cross-producting scheme together with rule caching are proposed [4] for classifiers bigger than 50 rules in five dimensions. Incremental updates require reconstruction of the cross-product table, so it cannot support dynamic classifiers well.

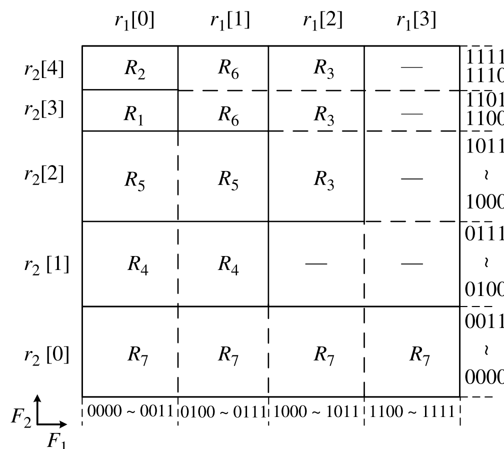


Figure 3.11 Geometric representation of the cross-producting algorithm.

3.3.3 Bitmap-Intersection

The bitmap-intersection scheme proposed by Lakshman et al. [8] applies to multi-dimensional packet classification with either type of specification in each field. This scheme is based on the observation that the set of rules, S, that matches a packet is the intersection of d sets, S_i, where S_i is the set of rules that matches the packet in the ith dimension alone.

Figure 3.12 contains an example to illustrate how the bitmap-intersection scheme works. Four rules of a 2D classifier are depicted as four rectangles in Figure 3.12 and projected on the two number lines. Two sets of intervals $\{X_1, \ldots, X_6\}$ and $\{Y_1, \ldots, Y_6\}$ are derived in each dimension by the rule projections. Each interval is associated with a precomputed 4-bit bitmap with each bit representing a rule. A '1' in the bitmap of X_k/Y_k denotes that the rule contains (matches) X_k/Y_k in the X/Y dimension. Given a packet $P(p_1, p_2)$, two range lookups (e.g., using a multiway search tree in Fig. 3.9) are performed in each interval set and two intervals, X_i and Y_j, which contain p_1 and p_2, are determined. Then the resulting bitmap, obtained by the intersection (a simple bitwise AND operation) of the bitmaps of X_i and Y_j, shows all matching rules for P. If the rules are ordered in decreasing order of priority, the first '1' in the bitmap denotes the highest priority rule. It is straightforward to expand the scheme to apply to a multi-dimensional classification.

Since each bitmap is N bits wide, and there are $O(N)$ ranges in each of the d dimensions, the storage space consumed is $O(dN^2)$. Query time is $O(d \cdot t_{RL} + dN/w)$, where t_{RL} is the time to perform one range lookup and w is the memory width. Time complexity can be reduced by a factor of d by looking up each dimension independently in parallel. Incremental updates are not well-supported.

It is reported that the scheme can support up to 512 rules with a 33-MHz FPGA and five 1-Mbyte SRAMs, classifying 1 mpps [8]. The scheme works well for a small number of rules in multiple dimensions, but suffers from a quadratic increase in storage and linear increase in classification time with the size of the classifier.

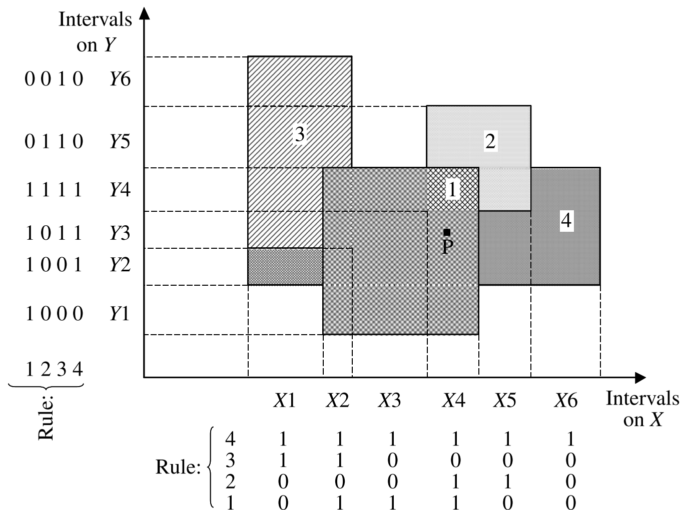


Figure 3.12 Geometric interpretation of the bitmap-insertion scheme for a 2D classifier.

3.3.4 Parallel Packet Classification (P^2C)

Lunteren et al. [9] proposed a fast multi-field classification scheme based on the independent searching and primitive range encoding. The idea is that the ranges in each dimension are first encoded into some code vectors. Then for a specific classification operation, P^2C performs irrespective parallel searching in each dimension (sub-range) to get the corresponding code vectors. And then the vectors found in all dimensions are combined together to carry out a ternary match in TCAM to finally gain the multi-field classification outcome. Ranges are defined as the intervals divided by the boundaries of all rules; as shown in Figure 3.13, for example, X_0–X_8 on axis X and Y_0–Y_6 on axis Y are all ranges. And it is straightforward that each range has a unique matching condition.

The ranges on each dimension divide the hyper-plane into several 'grids' (as depicted in Fig. 3.13); each of the grids can be represented by a d-tuple of the ranges; and the rule with the highest priority that covers one of the grids is defined as 'the rule corresponding to this grid'. For example, as the case shown in Figure 3.13, the grid determined by the 2-tuple (X_2, Y_2) corresponds to R_4, and the grid determined by (X_5, Y_4) corresponds to R_1, with assumption that R_1 is of higher priority.

According to this principle, if the correspondent relationships of the tuple of the ranges and the rules are pre-computed, the classification operation is then carried out to be: (1) to find the corresponding ranges of the given key on each dimension in parallel; (2) to combine the ranges into a key to find the corresponding rule.

In P^2C, the ranges are all encoded into code vectors; in the first step, the ranges are found in the form of code vectors and then the code vectors are combined together and put in a TCAM to gain the final multi-dimension classification outcome.

Encoding the Ranges. As shown in Figure 3.14, the layers (dashed lines) are defined according to the priority of the rules and the relation of overlap between them. The higher the priority of a rule is, the higher the layer its corresponding range belongs to, on their dimension; non-overlapping rules can be within the same layer.

'Layer' is the minimum unit in the assignment of bits of code. Binary digits above the lines are code vectors assigned to the range on this layer, representing the matching

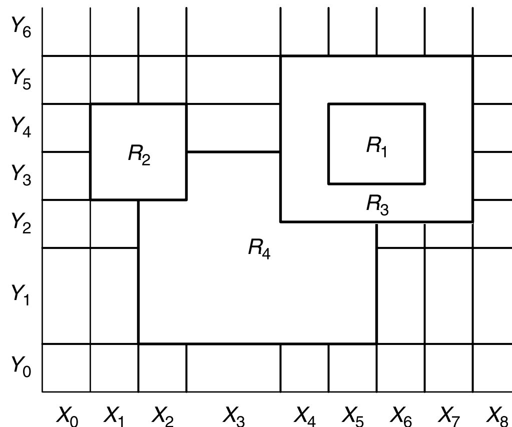


Figure 3.13 Rules and primitive ranges in both X and Y dimensions.

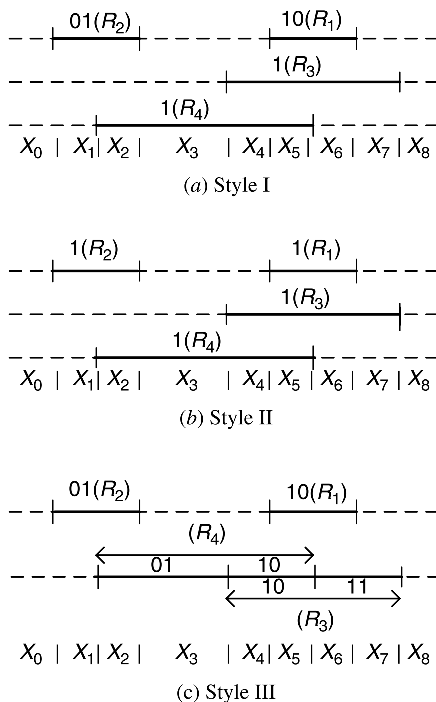


Figure 3.14 Three encoding styles.

conditions of the corresponding ranges, such as '01' and '10' in Figure 3.14*a*. Three range coding styles, as shown in Figure 3.14, have been proposed to fit for different environments, in order to get the optimal encoding result (with the fewest bits assigned).

The corresponding results produced by the three encoding styles are given by Table 3.4, according to the encoding styles in Figure 3.14. Then, according to the relationship of the rules and the range (as shown in Fig. 3.13), the matching conditions of the rule in dimension X is given by Table 3.5. The ternary form matching conditions for dimension Y can be obtained similarly.

When a packet arrives, it is mapped to a range independently. The range corresponds to a code, which is then combined with the codes of other dimensions' ranges. The combined range code is then used to lookup the TCAM to find the highest priority rule.

TABLE 3.4 Intermediate Result Vectors for the Range Hierarchies

	Ranges						
	X_1	X_2	X_3	X_4	X_5	X_6	X_7
Style I	0100	0101	0001	0011	1011	1010	0010
Style II	100	101	001	011	111	110	010
Style III	0100	0101	0001	0010	1010	1011	0011

TABLE 3.5 Ternary-Match Conditions for the Range Hierarchies

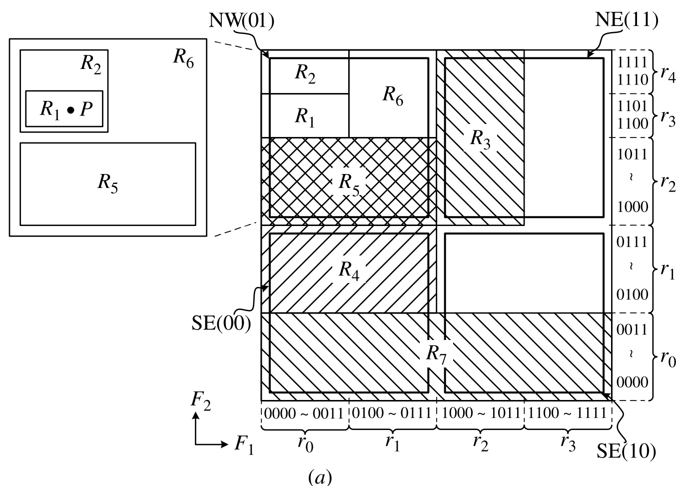
	R_4	R_3	R_2	R_1
Style I	xxx1	xx1x	01xx	10xx
Style II	xx1	x1x	10x	11x
Style III	xx01,xx10	xx10,xx11	01xx	10xx

Performance Comments. Adopting independent field search makes parallel search in each dimension viable. The time complexity of this algorithm lies on the most time-consuming one of those independent field searches. Every packet classification needs about five memory accesses (four for the longest field search, and one for TCAM access) [9]. The reported memory requirement is relatively good because the coding method is used. The algorithm has been designed such that any rule insertion or removal will not impact the codes for existing rules. Furthermore, compared to most other packet classification schemes, P^2C stores the data related to a single rule at very few locations, typically one location in each field search structure and at one location in the TCAM (e.g., a total of 6 locations for 5-tuple classification). As a result, updates require only incremental modification of very few locations in the data structure, enabling fast efficient updates with a rate $>$10,000 rules/second. Those updates can be performed without interrupting the classification operation.

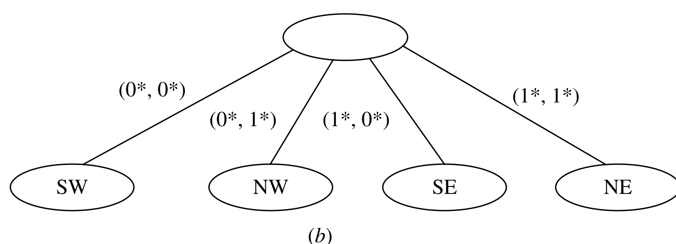
3.3.5 Area-Based Quadtree

Buddhikot et al. [10] proposed the area-based quadtree (AQT) structure for 2D classification with prefix specification in both fields as shown in Figure 3.15*a*. Compared with a binary tree, the node of quadtree may have up to four children and four pointers labeled with 00, 01, 10, and 11 as shown in Figure 3.15*b*. Each node in the AQT represents a 2D space that is evenly decomposed into four quadrants corresponding to its four children. For example, the root node of Figure 3.15*b* denotes the entire square in Figure 3.15*a* and thus the four children represent the four quadrants, SW (southwest), NW (northwest), SE (southeast), and NE (northeast), respectively. If the 2D space represented by the root node is expressed as a prefix pair (*, *), the four quadrants are therefore (0*, 0*), (0*, 1*), (1*, 0*), and (1*, 1*) with each prefix denoting a range in one dimension.

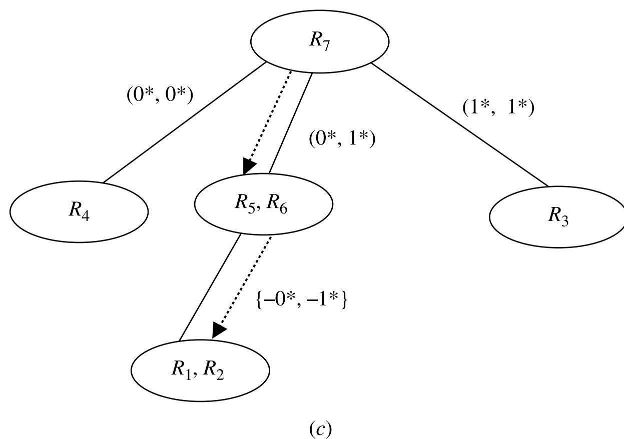
For a certain classifier C, the AQT is constructed as follows. A rule set $S(u)$ containing all rules of C is associated with the root node u. u is expanded with four children (decomposing space (*, *)) each associated with a rule set $S(v_m)$, where $1 \leq m \leq 4$. $S(v_m)$ contains the rules that are fully contained in the quadrant represented by v_m. However, there are some rules in $S(u)$ that are not fully contained in any of the four quadrants. Such rules are stored in another so-called crossing filter set (CFS) associated with node u. Each rule in the CFS is said to be a crossing node u because it completely spans at least one dimension of the 2D space represented by u. For instance, rule R_7 denoted by a bold rectangle in Figure 3.15*a* is said to cross space (*, *) since the rectangle completely spans x-dimension of space (*, *). Thus, R_7 is stored in the CFS of the root node in Figure 3.15*c*. Each child node v_m of u is expanded in the same way until $S(v_m)$ is empty. This procedure is carried out recursively and the construction terminates until the rule set S of each leaf node is empty. Note that only the CFS is indeed stored at each node after the construction. Therefore, the storage



(*a*)



(*b*)



(*c*)

Figure 3.15 Example of the area-based quadtree structure. (*a*) Cross-producting scheme; (*b*) Area-based quadtree with the four quadrants; and (*c*) Expressed as (F_1, F_2) pairs from each rule in Table 3.2.

complexity of AQT tree is $O(NW)$ since each rule is only stored once at a certain node v, where the rule crosses the 2D space of v.

Given the 2-tuple, $P(p_1, p_2)$, query of the AQT involves traversing the quadtree from root node to a leaf node. The branching decision at each node is made by two bits, each taken

from p_1 and p_2. P is compared with the rules in the CFS of each node encountered along the traversing path to find matches. The best matching rule is returned after traversing to the leaf node. The CFS at each node could be split into two sets, x-CFS and y-CFS. The former contains rules completely spanning in x dimension, while the latter spans in y dimension. Because of this spanning, only the $y(x)$ dimension (field) of each rule in $x(y)$-CFS needs to be stored. For instance, R_7 completely spans the space (*, *) in x dimension so it belongs to the x-CFS of the root node. In fact, only '00*', which is the y field of R_7 is kept and used for the range lookup at the root node. Now finding matches of P in the CFS of a node is transformed into two range lookups in x-CFS and y-CFS. Figure 3.15c shows the AQT for the classifier in Table 3.2 with the geometric interpretation shown in Figure 3.15a. The dashed line in Figure 3.15c indicates the traversing path when a 2-tuple (001, 110) (point P in Fig. 3.15a) is searched in the AQT.

An efficient update algorithm for the AQT is proposed in [10]. It has $O(NW)$ space complexity, $O(\alpha W)$ search time, and $O(\alpha^{\alpha} \cdot \sqrt{N})$ update time, where α is a tunable integer parameter.

3.3.6 Hierarchical Intelligent Cuttings

Hierarchical intelligent cuttings (HiCuts), proposed by Gupta and McKeown [11], partitions the multidimensional search space guided by heuristics that exploit the structure of the classifier.

Rule Storing Organization. The HiCuts algorithm builds a decision tree data structure by carefully preprocessing the classifier. Each internal node v of the decision tree built on a d-dimensional classifier is associated with:

1. A box $B(v)$, which is a d-tuple of intervals or ranges: $([l_1 : r_1], [l_2 : r_2], \ldots, [l_d : r_d])$.
2. A cut $C(v)$, defined by a dimension i, and $np(C)$, the number of times $B(v)$ is cut (partitioned) in dimension i (i.e., the number of cuts in the interval $[l_i : r_i]$). The cut thus evenly divides $B(v)$ into smaller boxes, which is associated with the children of v.
3. A set of rules $S(v)$. The tree's root has all the rules associated with it. If u is a child of v, then $S(u)$ is defined as the subset of $S(v)$ that collides with $B(u)$. That is, every rule in $S(v)$ that spans, cuts, or is contained in $B(u)$ is also a member of $S(u)$. $S(u)$ is called the colliding rule set of u.

As an example, consider the case of two W-bit-wide dimensions. The root node, v, represents a box of size $2^W \times 2^W$. We make the cuttings using axis-parallel hyperplanes, which are just lines in two dimensions. Cut $C(v)$ is described by the number of equal intervals we cut in a particular dimension of box $B(v)$. If we decide to cut the root node along the first dimension into D intervals, the root node will have D children, each with an associated box of size $(2^W/D) \times 2^W$.

We perform cutting on each level and recursively on the children of the nodes at that level until the number of rules in the box associated with each node falls below a threshold called *binth*. In other words, the number of rules in each leaf node is limited to at most binth to speed up the linear search in the node. A node with fewer than binth rules is not partitioned further and becomes a leaf of the tree.

TABLE 3.6 Rule Set Example with Two Dimensions in Ranges

Rule	X Range	Y Range
R_1	0–31	0–255
R_2	0–255	128–131
R_3	64–71	128–255
R_4	67–67	0–127
R_5	64–71	0–15
R_6	128–191	4–131
R_7	192–192	0–255

To illustrate this process, Table 3.6 shows an example classifier. Figure 3.16 illustrates this classifier geometrically and Figure 3.17 shows a possible decision tree. Each ellipse denotes an internal node v with a triplet $(B(v), \dim(C(v)), np(C(v)))$ and each square is a leaf node containing rules. The root node u denotes the entire space with the box $B(u) = 256 \times 256$. $B(u)$ is evenly cut into four small boxes in dimension X shown in Figure 3.17. In this example, binth $= 2$. Therefore, the set with R_2, R_3, R_4, and R_5 is further cut in Y dimension.

Classification Scheme. Each time a packet arrives, the classification algorithm traverses the decision tree to find a leaf node, which stores a small number of rules. A linear search of these rules yields the desired matching.

Performance Comments. The characteristics of the decision tree (its depth, degree of each node, and the local branching decision to be made at each node) can be tuned to trade off query time against storage requirements. They are chosen while preprocessing the classifier based on its characteristics. Four heuristics have been proposed when performing the cuts on node v [11].

For 40 real-life 4D classifiers and some of them with up to 1700 rules, HiCuts requires less than 1 Mbyte of storage, has a worst-case query time of 20 memory accesses, and supports fast updates.

3.3.7 HyperCuts

Singh et al. [12] have proposed a classification algorithm based on a decision tree, called HyperCuts, the idea of which is somewhat similar with that of the HiCuts algorithm (Section 3.3.6) in that they both allow the leaf nodes to store more than one rule and that linear search is performed at the leaf nodes. HyperCuts makes two improvements over HiCuts:

1. At each node of the decision tree, the rule is cut in multi-dimension at one time and stored in a multi-dimensional array. In other words, it uses HyperCut to cut rules, while HiCuts only considers one-dimensional cutting.
2. If all the children of a node in the tree contain the same subset of rules, the subset is lifted up to be stored in the node to reduce storage space. For instance, R2 in Figure 3.17 can be moved up to the root node and eliminated from the four children nodes.

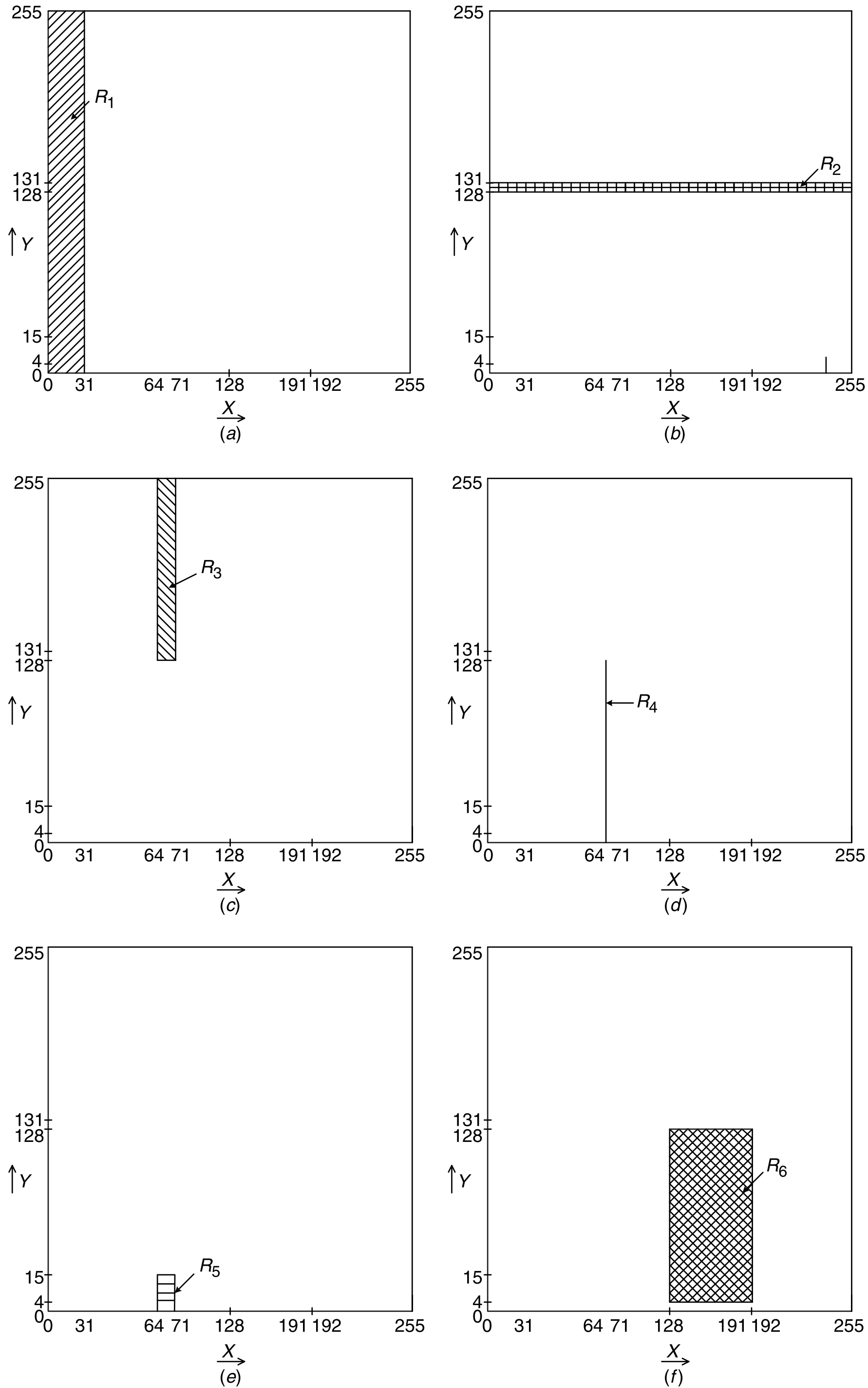


Figure 3.16 Geometrical representation of the seven rules in Table 3.6. (*a*) R_1; (*b*) R_2; (*c*) R_3; (*d*) R_4; (*e*) R_5; (*f*) R_6; (*g*) R_7; (*h*) All seven rules.

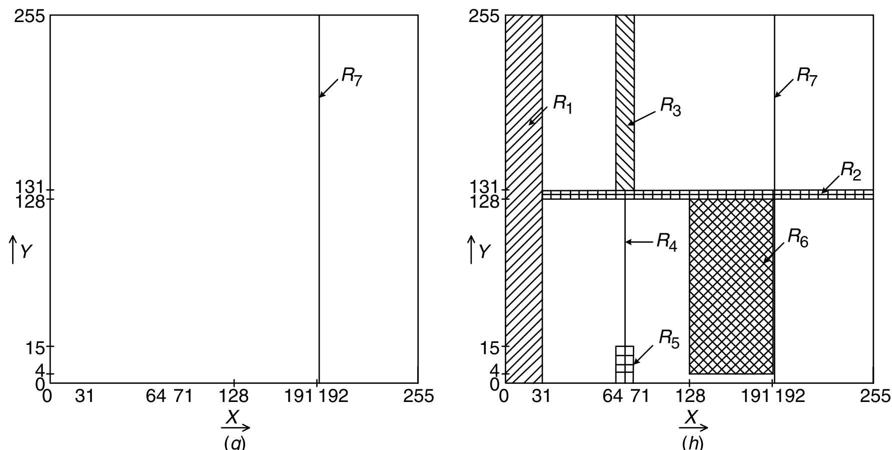


Figure 3.16 Continued.

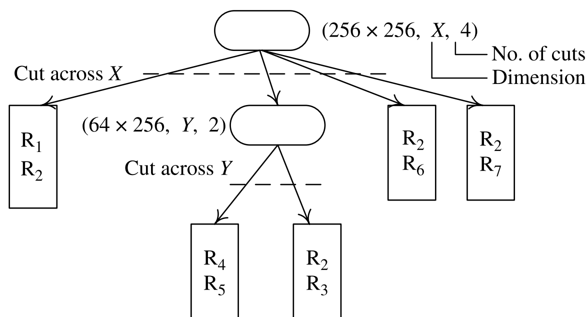


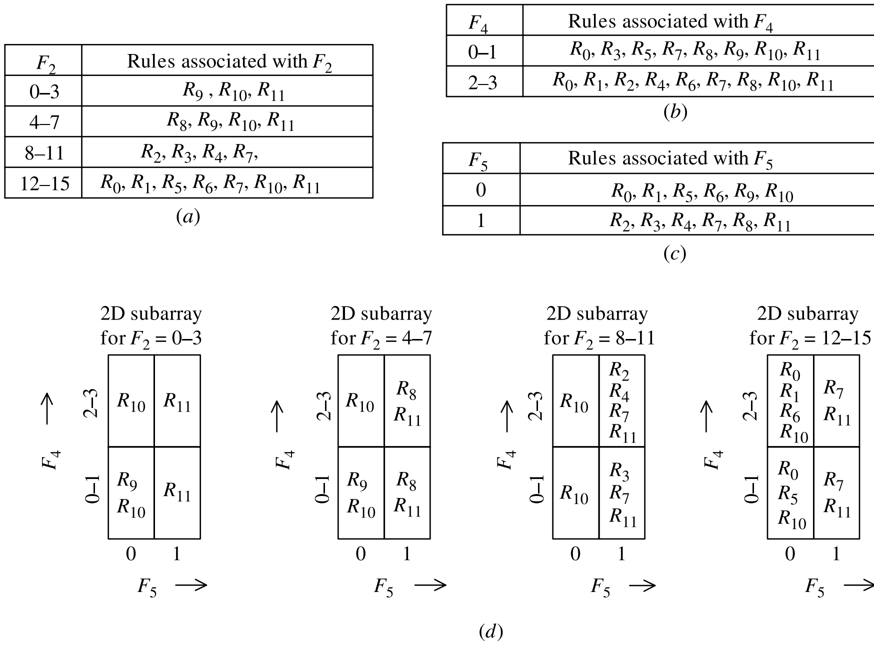
Figure 3.17 Possible decision tree for classifier in Table 3.6 ($binth = 2$).

Each node in the decision tree has associated with:

- A region $R(v)$ that is covered.
- A number of cuts (NC) and a corresponding array of NC pointers.
- A list of rules that may match.

Figure 3.18 shows HyperCuts in action for classifier in Table 3.7. The tree consists of a single root node that covers the region [0–15, 0–15, 0–3, 0–3, 0–1], which is split into sub-regions with 16 cuts. Note that each dimension is evenly cut into multiple regions.

Rule Storing Organization. The decision-tree-building algorithm starts with a set of N rules, each of the rules containing d dimensions. Each node identifies a region and has associated with it a set of rules S that match the region. If the size of the set of rules at the current node is larger than the acceptable bucket size (i.e., binth in HiCuts), the node is split in a number (NC) of child nodes, where each child node identifies a sub-region of the region associated with the current node. Identifying the number of child nodes as well as the sub-region associated with each of the child nodes is a two-step process, which tries to



F_2	Rules associated with F_2
0–3	R_9 , R_{10}, R_{11}
4–7	R_8, R_9, R_{10}, R_{11}
8–11	R_2, R_3, R_4, R_7,
12–15	R_0, R_1, R_5, R_6, R_7, R_{10}, R_{11}

(*a*)

F_4	Rules associated with F_4
0–1	R_0, R_3, R_5, R_7, R_8, R_9, R_{10}, R_{11}
2–3	R_0, R_1, R_2, R_4, R_6, R_7, R_8, R_{10}, R_{11}

(*b*)

F_5	Rules associated with F_5
0	R_0, R_1, R_5, R_6, R_9, R_{10}
1	R_2, R_3, R_4, R_7, R_8, R_{11}

(*c*)

Figure 3.18 HyperCuts decision tree based on the classifier in Table 3.7. (*a*) Rules corresponding to fixed values in F_2; (*b*) Rules corresponding to fixed values in F_4; (*c*) Rules corresponding to fixed values in F_5; (*d*) A search through the HyperCuts decision tree (evenly cut in each dimension).

locally optimize the split(s) such that the distribution of the rules among the child nodes is optimal. The detailed cutting process can be found in the work of Singh et al. [12].

Classification Scheme. The search algorithm for a packet with an i-dimensional header starts with an initialization phase of setting the current node for searching as the root node

TABLE 3.7 Range-based Representation of a Classifier with 12 Rules on Five Fields (F_1–F_5)

Rule	F_1	F_2	F_3	F_4	F_5	Action
R_0	0–1	14–15	2	0–3	0	act_0
R_1	0–1	14–15	1	2	0	act_0
R_2	0–1	8–11	0–3	2	1	act_1
R_3	0–1	8–11	0–3	1	1	act_2
R_4	0–1	8–11	2	3	1	act_1
R_5	0–7	14–15	2	1	0	act_0
R_6	0–7	14–15	2	2	0	act_0
R_7	0–7	8–15	0–3	0–3	1	act_2
R_8	0–15	4–7	0–3	0–3	1	act_2
R_9	0–15	0–7	0–3	1	0	act_0
R_{10}	0–15	0–15	0–3	0–3	0	act_3
R_{11}	0–15	0–15	0–3	0–3	1	act_4

of the decision tree structure and setting the regions that cover the packet header to the maximum value of the regions for each of the dimensions. Then the decision tree is traversed until either a leaf node or a NULL node is found, with the hyper-regions that cover the values in the packet header being updated at each node traversed. Once a leaf node is found, the list of rules associated with this node is fully searched and the first matching rule is returned. If there is no match, a NULL is returned.

This is further explained by going through an example. Figure 3.19 shows that a packet arrives at a node A that covers the regions 200–239 in the X dimension and 80–159 in the Y dimension. The packet header has the value 215 in the X dimension and 111 in the Y dimension. During the search, the packet header is escorted by a set of registers carrying information regarding the hyper-region to which the packet header belongs at the current stage. In this example, the current hyper-region is $\{[200–239], [80–159], \ldots\}$.

Node A has 16 cuts, with four cuts for each of the dimensions X and Y. To identify the child node, which must be followed for this packet header, the index in each dimension is determined as follow: first, $X_{\text{index}} = \lfloor 215 - 200/10 \rfloor = 1$. This is because each cut in the X dimension is of size $(239 - 200 + 1)/4 = 10$. Similarly, $Y_{\text{index}} = \lfloor 111 - 80/20 \rfloor = 1$. This is because each cut in the Y dimension is of size $(159 - 80 + 1)/4 = 20$. As a result the child node B is picked and the set of registers is updated with the new values describing the hyper-region covering the packet header at this stage. This hyper-region is now: $\{[210–219], [100–119], \ldots\}$. The search ends when a leaf node is reached in which case the packet header is checked against the fields in the list of rules associated with the node.

Performance Comments. By contrast, HyperCuts allows the cutting of both source and destination IP fields in a single node, which not only disperses the rules among the child nodes in a single step but also reduces the effect of rule replication. Furthermore, HyperCuts pushing up common sets of rules reduces the damage due to replication. It is reported in [12] that for real-life firewall databases, this optimization resulted in a memory

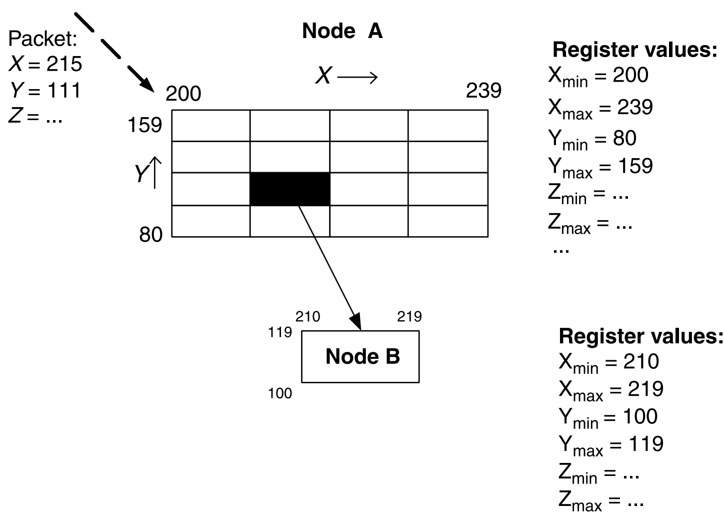


Figure 3.19 Search through the HyperCuts decision tree.

reduction of 10 percent. Overall, for firewall databases, HyperCuts uses an amount of memory similar to EGT-PC (Section 3.2.4) while its search time is up to five times better than HiCuts optimized for speed. For synthetic core-router style databases of 20,000 rules size, HyperCuts requires only 11 memory accesses for search in the worst case; for edge-router style databases, HyperCuts requires 35 memory accesses for a database of 25,000 rules. In the case of firewall-like databases, the presence of ~10 percent wildcards in either of the source and destination IP field contributes to a steep memory increase. This is possibly because of a large number of rules replicated in leaf nodes.

3.4 HEURISTIC ALGORITHMS

3.4.1 Recursive Flow Classification

Gupta and McKeown [3, 7] have proposed a heuristic scheme called Recursive Flow Classification (RFC) and have applied it to packet classification on multiple fields. Classifying a packet involves mapping the H-bit packet header to a T-bit action identifier (where $T = \log N$, $T \ll H$) for a N-rule classifier. A simple, but impractical, method is to precompute the action for each of the 2^H different packet headers and store the results in a $2^H \times 2^T$ array. Therefore, only one memory access is needed to yield the corresponding action. But this would require too much memory. The main aim of RFC is to perform the same mapping, but over several stages (phases), as shown in Figure 3.20. The mapping is performed recursively; at each stage, the algorithm performs a reduction, mapping one set of values to a smaller set. In each phase, a set of memories returns a value shorter, that is, expressed in fewer bits, than the index of the memory access. Figure 3.21 illustrates the structure and packet flow of the RFC scheme.

Rule Storing Organization. The construction of the preprocessed tables is explained with a sample rule set C [7] shown in Table 3.8. Note that the address information has been sanitized, which makes it different from what we used in reality.

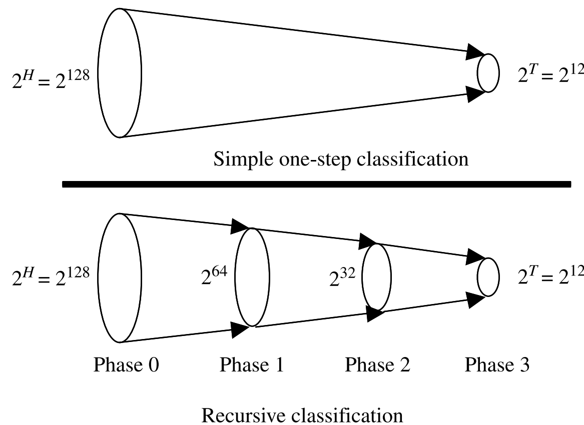


Figure 3.20 Basic idea of the recursive flow classification (RFC).

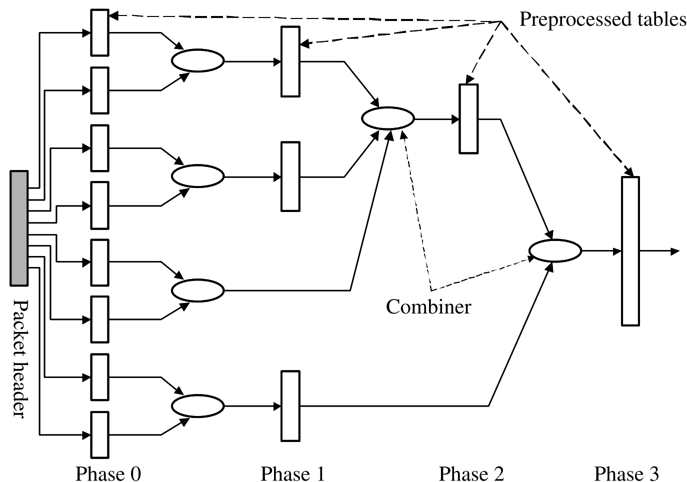


Figure 3.21 Packet flow in the recursive flow classification.

TABLE 3.8 Rule Set Example

Destination IP (addr/mask)	Source IP (addr/mask)	Port Number	Protocol
152.163.190.69/255.255.255.255	152.163.80.11/0.0.0.0	*	*
152.168.3.0/255.255.255.0	152.163.200.157/0.0.0.0	eq www	UDP
152.168.3.0/255.255.255.0	152.163.200.157/0.0.0.0	range 20–21	UDP
152.168.3.0/255.255.255.0	152.163.200.157/0.0.0.0	eq www	TCP
152.163.198.4/255.255.255.255	152.163.160.0/255.255.252.0	gt 1023	TCP
152.163.198.4/255.255.255.255	152.163.36.0/0.0.0.255	gt 1023	TCP

1. The first step of constructing the preprocessed table is to split the d fields of the packet header into multiple chunks that are used to index multiple memories in parallel. For example, the number of chunks equals eight in Figure 3.21. One possible way of chopping the packet header for rule set C is shown in Figure 3.22.
2. Each of the parallel lookups will map the chunk to an *eqID* according to the rules. Consider a chunk of size b bits. Its mapping table is of 2^b entries and each entry contains an *eqID* for that chunk value. The *eqID* is determined by those component(s) of the rules in the classifier corresponding to this chunk. The term 'Chunk Equivalence Set' (CES) is used to denote a set of chunk values that have the same *eqID*. To further understand the meaning of 'CES', consider chunk 3 for the classifier in Table 3.8. If there are two packets with protocol values lying in the same set and have otherwise

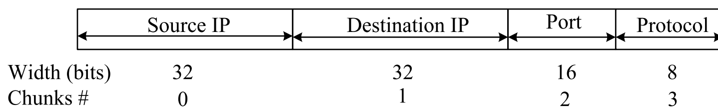


Figure 3.22 Chopping of packet header into chunks for rule set C in the first phase of RFC.

identical headers, the rules of the classifier do not distinguish between them. Then the 'CES' for chunk 3 will be:

(a) {TCP}

(b) {UDP}

(c) {all remaining numbers in the range 0–255}

Each CES can be constructed in the following manner. For a *b*-bit chunk, project the rules in the classifier on to the number line $[0, 2^b - 1]$. Each component projects to a set of (not necessarily contiguous) intervals on the number line. The end points of all the intervals projected by these components form a set of non-overlapping intervals. Two points in the same interval always belong to the same equivalence set. Two intervals are also in the same equivalence set if exactly the same rules project onto them. An example of chunk 2 of the classifier in Table 3.8 with the end-points of the intervals ($I_0 \ldots I_4$) and the constructed equivalence sets ($E_0 \ldots E_3$) are shown in Figure 3.23. The four CESs can be decoded using two bits. For example, we can assign '00' to E_1, '01' to E_0, '10' to E_2 and '11' to E_3. Then the RFC table for this chunk is filled with the corresponding *eqID*s, such as *table*(20) ='01', *table*(23) ='11', etc.

3. A chunk in the following steps is formed by a combination of two (or more) chunks obtained from memory lookups in previous steps, with a corresponding CES. If, for example, the resulting chunk is of width *b* bits, we again create equivalence sets such that two *b*-bit numbers that are not distinguished by the rules of the classifier belonging to the same CES. Thus, (20, UDP) and (21, UDP) will be in the same CES in the classifier of Table 3.8 in the second step. To determine the new equivalence sets for this phase, we compute all possible intersections of the equivalence sets from the previous steps being combined. Each distinct intersection is an equivalence set for the newly created chunk. For example, if we combine chunk 2 (port number) and 3 (protocol), then five CESs can be obtained:

(a) $\{(\{80\}, \{UDP\})\}$

(b) $\{(\{20 - 21\}, \{UDP\})\}$

(c) $\{(\{80\}, \{TCP\})\}$

(d) $\{(\{gt1023\}, \{TCP\})\}$

(e) {all the remaining crossproducts}.

These can be expressed in a three-bit *eqID*, as shown in Figure 3.24*e*. From this example, we can see that the number of bits has been reduced from four to three during step one (two bits for chunk 2 and 3, respectively) and step two. For the combination of the two steps, this number has dropped from 24 to 3.

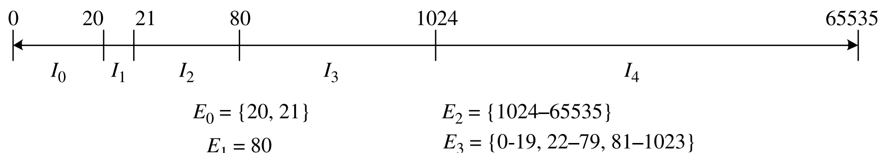


Figure 3.23 Example of computing the four equivalence classes $E_0 \ldots E_3$ for chunk 2 (corresponding to the 16-bit transport-layer destination port number) in the rule set of Table 3.8.

Destination IP (addr/mask)	Chunk Number	eqID (only 2 bits required)
152.163.190.69/255.255.255.255	0	00
152.168.3.0/255.255.255.0	1	01
152.163.198.4/255.255.255.255	2	10

(*a*)

Source IP (addr/mask)	Chunk Number	eqID (only 2 bits required)
152.163.80.11/0.0.0.0	0	00
152.163.200.157/0.0.0.0	1	01
152.163.160.0/255.255.252.0	2	10
152.163.36.0/0.0.0.255	3	11

(*b*)

Port Number	Chunk Number	eqID (only 2 bits required)
Range 20-21	0	00
eq www	1	01
gt 1023	2	10
0-19,22-79,81-1023	3	11

(*c*)

Protocol	Chunk Number	eqID (only 2 bits required)
tcp	0	00
udp	1	01
all remaining protocols	2	10

(*d*)

Port Number and Protocol	Chunk Number	eqID (only 3 bits required)
eq www & udp	0	000
Range 20-21 & udp	1	001
eq www & tcp	2	010
gt 1023 & tcp	3	011
all remaining crossproducts	4	100

(*e*)

Figure 3.24 Rule storing organization for RFC for the rule set in Table 3.8. (*a*) Destination IP field made into chunks and epIDs. (*b*) Source IP field made into chunks and eqIDs. (*c*) Port number field made into chunks and eqIDs. (*d*) Protocol field made into chunks and eqIDs. (*e*) Port number and protocol fields combined and made into chunks and eqIDs.

Classification Scheme. The classification of a packet is first split into several chunks to be used as an index; then the required *eqID*s are combined into chunks of the second phase; this procedure goes on until the final phase is reached when all the remaining *eqID*s have been combined into only one chunk. The corresponding table will hold the actions for that packet.

Performance Comments. The contents of each memory are chosen so that the result of the lookup is narrower than the index. Different combinations of the chunks can yield different storage requirements. It is reported [7] that with real-life 4D classifiers of up to 1700 rules, RFC appears practical for 10 Gbps line rates in hardware and 2.5 Gbps rates in

software. However, the storage space and preprocessing time grow rapidly for classifiers larger than 6000 rules. An optimization described in Ref. [7] reduces the storage requirement of a 15,000 four-field classifier to below 4 Mbytes.

3.4.2 Tuple Space Search

Srinivansan et al. [13] have proposed a tuple space search scheme for multi-dimensional packet classification with prefix specification. The basic tuple space search algorithm decomposes a classification query into several exact match queries in hash tables. This algorithm is motivated by the observation that while classifiers contain many different prefixes, the number of distinct prefix lengths tends to be small. Thus, the number of distinct combinations of prefix lengths is also small. Then a tuple for each combination of field length can be defined and by concatenating the known set of bits for each field in order, a hash key can be created to map the rules of that tuple into a hash table. Suppose a classifier C with N rules that results in M distinct tuples. Since M tends to be much smaller than N in practice, even a linear search through the tuple set is likely to greatly outperform the linear search through the classifier.

Rule Storing Organization. Each rule R in a classifier can be mapped into a d-tuple whose ith component specifies the length of the prefix in the ith field of R. A tuple space is defined as the set of all such tuples of a classifier. For each tuple in the tuple space, a hash table is created storing all rules mapped in the tuple. As an example, Table 3.9 shows the tuples and associated hash tables for the classifier in Table 3.2. For instance, (1, 2) means the length of the first prefix is one and the length of the second prefix is two.

Rules always specify IP addresses using prefixes, so the number of bits specified is clear. For port numbers that are often specified using ranges, the length of a port range is defined to be its nesting level. For instance, the full port number range [0, 65,535] has nesting level and length 0. The ranges [0, 1023] and [1024, 65,535] are considered to be nesting level 1, and so on. If we had additional ranges [30,000, 34,000] and [31,000, 32,000], then the former will have nesting level 2 and the latter 3 (this algorithm assumes that port number ranges specified in a database are non-overlapping).

While the nesting level of a range helps define the tuple it will be placed in, a key to identify the rule within the tuple is also needed. Thus a RangeID is used, which is a unique ID given to each range in any particular nesting level. So the full range always has the ID 0. The two ranges at depth 1, namely, ≤ 1023 and >1024, receive the IDs 0 and 1, respectively. Suppose we had ranges 200 . . . 333, 32,000 . . . 34,230, and 60,000 . . . 65,500 at level 2, then they would be given IDs 0, 1 and 2, respectively.

TABLE 3.9 Example of the Tuple Space Search Scheme

Tuple	Hash Table Entries
(0, 2)	$\{R_7\}$
(1, 1)	$\{R_6\}$
(1, 2)	$\{R_4, R_5\}$
(2, 1)	$\{R_2, R_3\}$
(2, 2)	$\{R_1\}$

Notice that a given port in a packet header can be mapped to a different ID at each nesting level. For example, with the above ranges, a port number 33,000 will map on to three RangeID values, one for each nesting depth: ID 0 for nesting depth 0, ID 1 for nesting depth 1, and ID 1 for nesting depth 2. Thus, a port number field in the packet header must be translated to its corresponding RangeID values before the tuple search is performed. In summary, the nesting level is used to determine the tuple, and the RangeID for each nesting level is used to form the hash key, the input to the hash function.

Classification Scheme. All rules that map to a particular tuple have the same mask: some number of bits in the IP source and destination fields, either a wild card in the protocol field or a specific protocol ID, and port number fields that contain either a wild card or a RangeID. Thus, we can concatenate the required number of bits from each field to construct a hash key for that rule. All rules mapped to a tuple U are stored in a hash table Hashtable(U). For instance, rules R_4 and R_5 are stored in the same hash table, say Hashtable2. A probe in a tuple U involves concatenating the required number of bits from the packet as specified by U (after converting port numbers to RangeID), and then doing a hash in Hashtable(U). The key will be hashed in each table to find the matched rule. For instance, the key is hashed and matched with R_4 in Hashtable2. Thus, given a packet P, we can linearly probe all the tuples in the tuple set, and determine the highest priority filter matching P.

Performance Comments. The query time complexity of the basic tuple space search scheme is $O(M)$, where M is the number of tuples in the tuple space. Perfect hashing is assumed here, which is chosen to avoid hash collisions.

M is still large for many real cases. Thus, a simple but efficient optimization called tuple pruning is proposed in [13] to improve the query speed and update performance. When a packet header presents to be classified, longest prefix matching is first performed in each dimension. The best matching prefix P_i in each dimension i returns a tuple list tl_i that is precomputed and associated with the prefix. Each tuple in the tuple list from P_i contains at least one rule whose ith field equals P_i or is a prefix of P_i. Another tuple list, the intersection list, is derived from the intersection of all tl_i. For a given query, only the tuples contained in the intersection list need to be searched. It will benefit if the reduction in the tuple space afforded by pruning offsets the extra individual prefix (or range) matches on each field. Reference [13] reports that by having the tuple pruning only in two fields, for example, the source and destination IP address, the number of tuples that needs to be searched is greatly reduced.

The storage complexity for tuple space search is $O(NdW)$ since each rule is stored in exactly one hash table. Incremental updates are supported and require just one hash memory access to the hash table associated with the tuple of the modified rule. However, the use of hashing may have hash collision and cause the search/update nondeterministic.

3.5 TCAM-BASED ALGORITHMS

3.5.1 Range Matching in TCAM-Based Packet Classification

Ternary content addressable memory (TCAM)-based algorithms are gaining increasing popularity for fast packet classification. In general, a TCAM coprocessor works as a look-aside processor for packet classification on behalf of a network processing unit (NPU) or

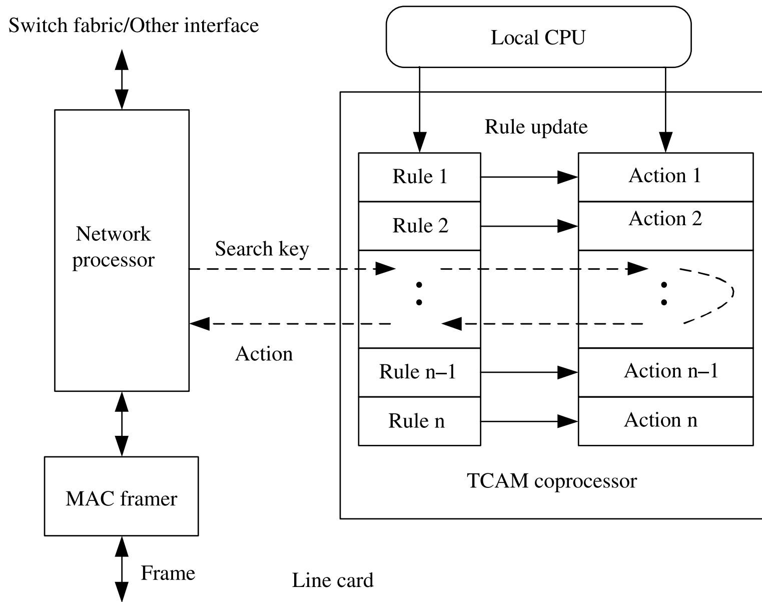


Figure 3.25 Network processor and its TCAM coprocessor.

network processor. When a packet is to be classified, an NPU generates a search key based on the information extracted from the packet header and passes it to the TCAM coprocessor for classification. A TCAM coprocessor finds a matched rule in $O(1)$ clock cycles and, therefore, offers the highest possible lookup/matching performance. Figure 3.25 is a logic diagram showing how an NPU works with its TCAM coprocessor. When a packet arrives, the NPU generates a search key based on the packet header information and passes it to the TCAM coprocessor to be classified, via a NPU/TCAM coprocessor interface. A local CPU is in charge of rule table update through a separate CPU-TCAM coprocessor interface.

However, despite its fast lookup performance, one of the most critical resource management issues in the use of TCAM for packet classification/filtering is how to effectively support filtering rules with ranges, known as range matching. The difficulty lies in the fact that multiple TCAM entries have to be allocated to represent a rule with ranges. A range is said to be exactly implemented in a TCAM if it is expressed in a TCAM without being encoded. For example, six rule entries are needed to express a range {>1023} in a TCAM, if the range is exactly implemented, as shown in Figure 3.26.

Today's real-world policy/firewall filtering (PF) tables were reported [7, 14–16] to involve significant amounts of rules with ranges. In particular, the statistical analysis of real-world rule databases in [17] shows that TCAM storage efficiency can be as low as

0000 01** **** ****   0000 1*** **** ****
0001 **** **** ****   001* **** **** ****
01** **** **** ****   1*** **** **** ****

Figure 3.26 Range >1023 is expressed in terms of 6 sub-ranges in a TCAM.

16 percent due to the existence of a significant number of rules with port ranges. Apparently, the reduced TCAM memory efficiency due to range matching makes TCAM power consumption, footprint, and cost an even more serious concern.

A widely adopted solution to deal with range matching is to perform range preprocessing/encoding by mapping ranges to a short sequence of encoded bits, known as bit-mapping. The idea is to view a d-tuple rule as a region in a d-dimensional rule space and encode any distinct overlapped regions among all the rules so that each rule can be translated into a sequence of encoded bits, known as rule encoding. Accordingly, a search key, which is based on the information extracted from the packet header, is preprocessed to generate an encoded search key, called search key encoding. Then, the encoded search key is matched against all the encoded rules to find the best matched rule. Unlike rule encoding, which can be pre-processed in software, search key encoding is performed on a per packet basis and must be done in hardware at wire-speed.

3.5.2 Range Mapping in TCAMs

As stated above, when the range specification is used for the field without range encoding, range splitting must be performed to convert the ranges into prefix formats to fit the bit boundary. This increases the number of entries and could make TCAMs infeasible for some classifiers that use the range specification. Liu [18] has suggested an efficient encoding scheme of range classifier into TCAM. The basic algorithm expands TCAM horizontally (using more bits per entry), and for a width limited application, an algorithm that allows both horizontal and vertical expansion is proposed.

Rule Storing Organization. For each range field, an n bits vector $B = b_1, b_2, \ldots, b_n$ is used to represent it, where n is the number of distinct ranges specified for this field. The B vector for a range E_i has 1 at bit position i, that is, $b_i = 1$ and all other bits are set to don't care. This is based on the observation that even though the number of rules in a classifier could be large, the number of distinct ranges specified for any range field is very limited and exact match specification also happens frequently for a range field. The bit vector representation for Table 3.10 is shown in Table 3.11 ($n = 5$ in this case). For example, the range of greater than 1023 in R1 is represented by 'xxxx1'.

Classification Scheme. A lookup key $v \in [0, 2k]$ is translated into an n bit vector $V = v_1, v_2, \ldots, v_n$. Bit v_i is set to 1 if the key v falls into the corresponding range E_i, otherwise it will be set to 0. Lookup key translation could be implemented as a direct memory lookup since most of the range fields are less than 16-bit wide. A complete lookup key translation

TABLE 3.10 Simple Example Classifier for TCAM Implementation

R_i	Dest IP Addr (IP/mask)	Dest Port Range	Action
1	10.0.0.0/255.0.0.0	>1023	Act_0
2	192.168.0.0/255.255.0.0	50–2000	Act_1
3	192.169.0.0/255.255.0.0	80(http)	Act_2
4	172.16.0.0/255.255.0.0	23(telnet)	Act_3
5	172.16.0.0/255.255.0.0	21(ftp)	Act_4

TABLE 3.11 Rules Stored in TCAM

R_i	TCAM Rules	
1	10.x.x.x	xxxx1
2	192.168.x.x	xxx1x
3	192.169.x.x	xx1xx
4	172.16.x.x	x1xxx
5	172.16.x.x	1xxxx

table for the example classifier for each possible lookup key value is shown in Figure 3.27. For example, the right most bit vector is set to 1 for all the locations above 1023. Furthermore, the second bit from the right between the locations 50 and 20 (including them) is also set to 1. Next paragraph, some optimization can be further made for exact match. It is possible to reduce the number of bits used to $\log_2(m + 1)$, where m is the number of exact matches.

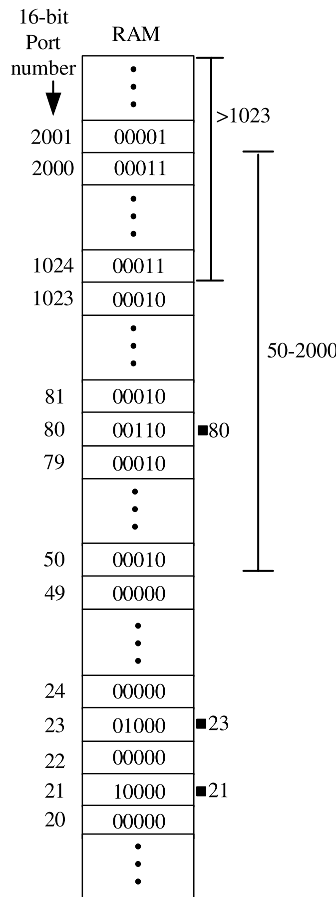


Figure 3.27 Lookup key translation table for the example classifier.

TABLE 3.12 Rules Stored in TCAM with Exact Match Optimization

R_i	TCAM Rules	
1	10.x.x.x	xxxx1
2	192.168.x.x	xxx1x
3	192.169.x.x	x01xx
4	172.16.x.x	x10xx
5	172.16.x.x	x11xx

The bit representation will contain two parts $\langle B_e, B \rangle$, B_e for exact matches, and B for all others. $B_e = b_1, b_2, \ldots, b_t$ is a t bit vector, where $t \geq \log_2(m+1)$.

For a normal range, $\langle B_e = 0, B \rangle$ and its B portion is the same as before. For an exact match, $\langle B_e = i, B = 0 \rangle$, if it is the ith exact match. In the previous example, if we use bit 2 and 3 as B_e (where bit 1 is the left-most significant bit), the example classifier stored in TCAM is shown in Table 3.12 and now only two bits are needed to represent the three distinct exact matches. The lookup key translation table needs to be changed accordingly. The lookup key also contains two parts $\langle V_e; V \rangle$, V_e corresponding to all exact matches and V to the rest. $V_e = \langle v_1, v_2, \ldots, v_t \rangle$ is a t bit vector (e.g., b_2 and b_3 of the right column of Table 3.12), and $V_e = i$ if the lookup key v equals to the ith exact match, otherwise $V_e = 0$. Assume that the classifier is stored in TCAM as shown in Table 3.12. A new packet arrives with destination IP address 192.169.10.1 and port number 80. First, the port number is indexed into the lookup key translation table (Figure 3.27). The resulting $V = 10$ because 80 falls in range 50–2000 but not range >1023. The resulting $V_e = 01$ because 01 is the value assigned to exactly match the value of 80. Together with destination IP address, the final result is rule 3.

Performance Comments. This scheme requires less TCAM storage space. Thus it can accommodate a much larger number of rules in a single TCAM table and reduce system cost and power consumption. Meanwhile, it also keeps the advantage of deterministic execution time of TCAM devices. However, adding/deleting rules causes the change of bit vectors and may require re-computation of the entire translation table, which is a time consuming process.

REFERENCES

[1] S. Lyer, R. R. Kompella, and A. Shelat, "ClassiPI: an architecture for fast and flexible packet classification," *IEEE Network*, vol. 15, no. 2, pp. 33–41 (Mar. 2001).

[2] P. Gupta, "Routing lookups and packet classifications: theory and practice," in *Proc. HOT Interconnects*, Stanford, California (Aug. 2000).

[3] P. Gupta and N. Mckeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32 (Mar. 2001).

[4] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvagel, "Fast and scalable layer four switching," in *Proc. ACM SIGCOMM*, Vancouver, Canada, pp. 191–202 (Aug. 1998).

[5] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?" in *Proc. IEEE INFOCOM'03*, San Francisco, California, vol. 1, pp. 53–63 (2003).

[6] G. Zhang and H. J. Chao, "Fast packet classification using field-level trie," in *Proc. IEEE GLOBECOM'03*, San Francisco, California, vol. 6, pp. 3201–3205 (Dec. 2003).

[7] P. Gupta and N. Mckeown, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM'99*, Harvard University, vol. 29, no. 4, pp. 147–160 (Aug. 1999).

[8] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM*, Vancouver, Canada, pp. 203–214 (Sep. 1998).

[9] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 560–571 (May 2003).

[10] M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space decomposition techniques for fast layer-4 switching," in *Conf. Protocols for High Speed Networks*, Holmdel, New Jersey, vol. 66, no. 6, pp. 277–283 (Aug. 1999).

[11] P. Gupta and N. McKeown, "Classification using hierarchical intelligent cuttings," in *Proc. HOT Interconnects VII*, Stanford, California (Aug. 1999).

[12] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, Karlsruhe, Germany, pp. 213–224 (Aug. 2003).

[13] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM*, Cambridge, Massachusetts, pp. 135–146 (Aug. 1999).

[14] F. Baboesu and G. Varghese, "Scalable packet classification," in *Proc. ACM SIGCOMM*, San Diego, California, pp. 199–210 (Aug. 2001).

[15] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Directions in packet classification for network processors," in *Proc. Second Workshop on Network Processors (NP2)*, Anahein, California (Feb. 2003).

[16] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *Proc. International Conference of Network Protocol (ICNP)*, Atlanta, Georgia, pp. 120–131 (Nov. 2003).

[17] H. Che, Z. Wang, K. Zheng, and B. Liu, "Dynamic range encoding scheme for TCAM coprocessors," Technical report. [Online]. Available at: http://crystal.uta.edu/hche/dres.pdf.

[18] H. Liu, "Efficient Mapping of range classifier into Ternary-CAM," in *Proc. 10th Hot Interconnects*, Stanford, California, pp. 95–100 (Aug. 2002).