CHAPTER 4

Recursive Flow Classification: An Algorithm for Packet Classification on Multiple Fields

1 Introduction

Chapters 2 and 3 described algorithms for routing lookups. In this chapter and the next we consider algorithms for multi-field packet classification.[1]

This chapter presents an algorithm for fast packet classification on multiple header fields. The algorithm, though designed with a hardware realization in mind, is suitable for implementation in software as well. As we will see from the overview of previous work on packet classification algorithms in Section 2, the packet classification problem is expensive to solve in the worst-case — theoretical bounds state that solutions to multi-field classification either require storage that is geometric, or a number of memory accesses that is polylogarithmic, in the number of classification rules. Hence, most classi-

1. The packet classification problem was introduced in Chapter 1: its motivation described in Section 2.1, problem definition in Section 2.3 and the metrics for classification algorithms in Section 3.

fication algorithms proposed in the literature [7][23][96] are designed to work well for two dimensions (i.e., with two header fields), but do not perform as well in multiple dimensions. This is explained in detail in Section 2.

This chapter makes the observation that classifiers in real networks have considerable structure and redundancy that can be exploited by a practical algorithm. Hence, this chapter takes a pragmatic approach, and proposes a heuristic algorithm, called RFC[1] (Recursive Flow Classification), that seems to work well with a selection of classifiers in use today. With current technology, it appears practical to use the proposed classification algorithm for OC192c line rates in hardware and OC48c rates in software. However, the storage space and preprocessing time requirements become large for classifiers with more than approximately 6000 four-field rules. For this, an optimization of the basic RFC algorithm is described which decreases the storage requirements of a classifier containing 15,000 four-field rules to below 4 Mbytes.

1.1 Organization of the chapter

Section 2 overviews previous work on classification algorithms. Section 3 describes the proposed algorithm, RFC, and Section 4 discusses experimental results of RFC on the classifiers in our dataset. Section 5 describes variations of RFC to handle larger classifiers. Section 6 compares RFC with previous work described in Section 2, and finally, Section 7 concludes with a summary and contributions of this chapter.

2 Previous work on classification algorithms

Recall from Section 3 of Chapter 1 that a classification algorithm preprocesses a given classifier to build a data structure, that is then used to find the highest priority matching rule for every incoming packet. We will assume throughout this chapter that rules do not

1. This is not to be confused with “Request for Comments”.

carry an explicit priority field, and that the matching rule closest to the top of the list of rules in the classifier is the highest priority matching rule. We will work with the following example classifier in this section.

Example 4.1: The classifier C shown in Table 4.1 consists of six rules in two fields (dimensions) labeled $F1$ and $F2$. All field specifications are prefixes of maximum length 3 bits. As per convention, rule priorities are ordered in decreasing order from top to bottom of the classifier.

TABLE 4.1. An example classifier.

Rule	F1	F2
R1	00*	00*
R2	0*	01*
R3	1*	0*
R4	00*	0*
R5	0*	1*
R6	*	1*

2.1 Range lookups

Algorithms that perform classification in multiple dimensions often use a one-dimensional lookup algorithm as a primitive. If the field specifications in a particular dimension are all prefixes, a lookup in this dimension usually involves either finding all matching prefixes or the longest matching prefix — this could be performed using any of the algorithms discussed in Chapters 2 and 3. However, as we will see in Section 3.2, field specifications can be arbitrary ranges. Hence, it will be useful to define the following range lookup problem for a dimension of width W bits.

Definition 1.1: *Given a set of N disjoint ranges $G = \{G_i = [l_i, u_i]\}$ that form a partition of the number line $[0, 2^W - 1]$, i.e., l_i and u_i are such that $l_1 = 0, l_i \le u_i, l_{i+1} = u_i + 1, u_N = 2^W - 1$; the range lookup problem is to find* *the range G_P (and any associated information) that contains an incoming point P.*

We have already seen one algorithm to solve the range lookup problem — the binary search algorithm of Section 2.2.6 in Chapter 2 builds a binary search tree on the endpoints of the set of ranges. We could also solve the range lookup problem by first converting each range to a set of maximal prefixes, and then solving the prefix matching problem on the union of the prefixes thus created. The conversion of a range to prefixes uses the observation that a prefix of length s corresponds to a range $[l, u]$ where the $(W-s)$ least significant bits of l are all 0 and those of u are all 1. Hence, if we split a given range into the minimum number of subranges satisfying this property, we arrive at a set of maximal prefixes equivalent to the original range. Table 4.2 lists examples of some range to prefix conversions for 4-bit fields.

TABLE 4.2. Examples of range to prefix conversions for 4-bit fields.

Range	**Constituent maximal prefixes**
[4,7]	01**
[3,8]	0011, 01**, 1000
[1,14]	0001, 001*, 01**, 10**, 110*, 1110

It can be seen that a range on a W-bit dimension can be split into a maximum of $2W-2$ maximal prefixes.[1] Hence, the range lookup problem can be solved using a prefix matching algorithm, but with the storage complexity increased by a factor of $2W$. Feldmann and Muthukrishnan [23] show a reduction of the range lookup problem to the prefix matching problem with an increase in storage complexity by only a constant factor of 2. However, as we will see later, this reduction cannot be used in all multi-dimensional classification schemes.

1. For example, the range $[1, 2^W-2]$ is split into $2W-2$ prefixes. An example of this is the last row in Table 4.2 with $W = 4$.

2.2 Bounds from Computational Geometry

There is a simple geometric interpretation of the packet classification problem. We have seen that a prefix represents a contiguous interval on the number line. Similarly, a two-dimensional rule represents an axes-parallel rectangle in the two-dimensional euclidean space of size $2^{W_1} \times 2^{W_2}$, where W_1 and W_2 are the respective widths of the two dimensions. Generalizing, a rule in d dimensions represents a d-dimensional hyperrectangle in d-dimensional space. A classifier is therefore a collection of rectangles, each of which is labeled with a priority. An incoming packet header represents a point with coordinates equal to the values of the header fields corresponding to the d dimensions. For example, Figure 4.1 shows the geometric representation of the classifier in Table 4.1. Rules of higher priority overlay those of lower priority in the figure.

Given this geometric representation, classifying an arriving packet is equivalent to finding the highest priority rectangle among all rectangles that contain the point representing the packet. If higher priority rectangles are drawn on top of lower priority rectangles (as in Figure 4.1), this is equivalent to finding the topmost visible rectangle containing a given point. For example, the packet represented by the point P(011,110) in Figure 4.1 would be classified by rule R_5.

There are several standard problems in the field of computational geometry [4][79][84], such as ray-shooting, point location and rectangle enclosure, that resemble packet classification. Point location in a multi-dimensional space requires finding the enclosing region of a point, given a set of *non-overlapping* regions. Since the hyperrectangles in packet classification could be overlapping, packet classification is at least as hard as point location. The best bounds for point location in N rectangular regions and d dimensions in the worst-case, for $d > 3$, are $O(\log N)$ time with $O(N^d)$ space;[1] or $O((\log N)^{d-1})$ time and $O(N)$ space [73][79]. Clearly this is impracticably slow for classi-

1. The time bound for $d \le 3$ is $O(\log\log N)$ [73] but has large constant factors.

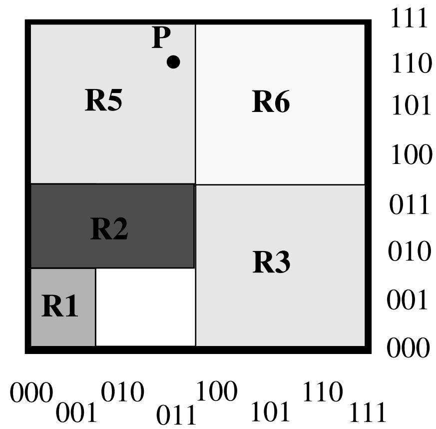


Figure 4.1 Geometric representation of the two-dimensional classifier of Table 4.1. An incoming packet represents a point in the two dimensional space, for instance, P(011,110). Note that R4 is completely hidden by R1 and R2.

fication in a high speed router — with just 100 rules and 4 fields, N^d space is about 100 Mbytes; and $(\log N)^{d-1}$ is about 350 memory accesses.

2.3 Linear search

As in the routing lookup problem, the simplest data structure is a linked-list of all the classification rules, possibly stored in sorted order of decreasing priorities. For every arriving packet, each rule is evaluated sequentially until a rule is found that matches all the relevant fields in the packet header. While simple and storage-efficient, this algorithm clearly has poor scaling properties: the time to classify a packet grows linearly with the number of rules.[1]

2.4 Ternary CAMs

We saw in Section 2.2.8 of Chapter 2 how ternary CAMs (TCAMs) can be used for performing longest prefix matching operations in dedicated hardware. TCAMs can similarly be used for multi-dimensional classification with the modification that each row of the TCAM memory array needs to be wider than 32 bits — the required width depends on

1. Practical evidence suggests that this data structure can support a performance between 10,000 and 30,000 packets per second using a 200 MHz CPU with a few hundred 4-dimensional classification rules.

the number of fields used for classification, and usually varies between 128 and 256 bits depending on the application. An increasing number of TCAMs are being used in the industry at the time of writing (for at least some applications) because of their simplicity, speed (the promise of classification in a single clock-cycle), improving density, and possibly absence of competitive algorithmic solutions. While the same advantages and disadvantages as discussed in Chapter 2 hold for a classification TCAM, we look again at a few issues specifically raised by classification.

- **Density**: The requirement of a wider TCAM further decreases its depth for a given density. Hence, for a 2 Mb 256-bit wide TCAM, at most 8K classification rules can be supported. As a TCAM row stores a (value, mask) pair, range specifications need to be split into mask specifications, further bringing down the number of usable TCAM entries by a factor of $(2W-2)^d$ in the worst case for d-dimensional classification. Even if only two 16-bit dimensions specify ranges (which is quiet common in practice with the transport-layer source and destination port number fields), this is a multiplicative factor of 900.
- **Power**: Power dissipated in one TCAM row increases proportionally to its width.

In summary, classification makes worse the disadvantages of existing TCAMs. Because of these reasons, TCAMs will probably still remain unsuitable in the near future for the following situations: (1) Large classifiers (256K-1M rules) used for microflow recognition at the edge of the network, (2) Large classifiers (128-256K rules) used at edge routers that manage thousands of subscribers (with a few rules per subscriber), (3) Extremely high speed (greater than 200-250 Mpps) classification, and (4) Software-based classification that may be required for a large number of dimensions, for instance, more than 8.

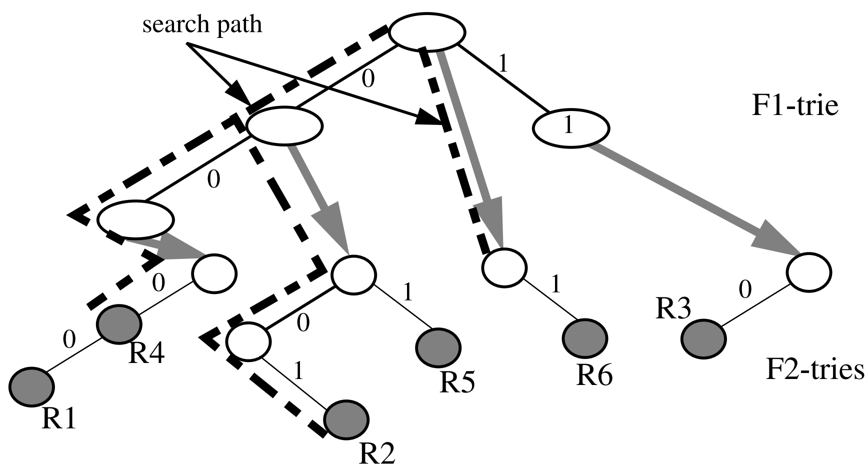


Figure 4.2 The hierarchical trie data structure built on the rules of the example classifier of Table 4.1. The gray pointers are the "next-trie" pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is also shown.

2.5 Hierarchical tries

A d-dimensional hierarchical radix trie is a simple extension of the radix trie data structure in one dimension (henceforth called a 1-dimensional trie), and is constructed recursively as follows. If d equals 1, the hierarchical trie is identical to the 1-dimensional radix trie studied before in Section 2.1.3 of Chapter 2. If d is greater than 1, we first construct a 1-dimensional trie on say dimension $F1$, called the $F1$-trie. Hence, the $F1$-trie is a 'trie' on the set of prefixes $\{R_{j1}\}$, belonging to dimension $F1$ of all rules in the classifier, $C = \{R_j\}$, where $R_j = \{R_{j1}, R_{j2}\}$. For each prefix, p, in the 1-dimensional $F1$-trie, we recursively construct a $(d-1)$-dimensional hierarchical trie, T_p, on those rules which exactly specify p in dimension $F1$, in other words, on the set of rules $\{R_j : R_{j1} = p\}$. Prefix p is linked to the trie T_p using another pointer called the next-trie pointer. For instance, the data structure in two dimensions is comprised of one $F1$-trie and several $F2$-tries linked to nodes in the $F1$-trie. The storage complexity of the data structure for an N-rule classifier is $O(NdW)$. The hierarchical trie data structure for the example classifier of

Table 4.1 is shown in Figure 4.2. Hierarchical tries are sometimes called "multi-level tries," "backtracking-search tries," or "trie-of-tries."

A classification query on an incoming packet $(v_1, v_2, \ldots, v_d)$ proceeds recursively on each dimension as follows. The query algorithm first traverses the 1-dimensional $F1$-trie based on the bits in v_1 in the usual manner. At each $F1$-trie node encountered during this traversal, the algorithm follows the next-trie pointer (if non-null) and recursively traverses the $(d-1)$-dimensional hierarchical trie stored at that node. Hence, this query algorithm encounters a rule in its traversal if and only if that rule matches the incoming packet, and it need only keep track of the highest priority rule encountered. Because of its recursive nature, the query algorithm is sometimes referred to as a backtracking search algorithm. The query time complexity for d-dimensions is $O(W^d)$. Incremental updates can be carried out in $O(d^2W)$ time since each of the d-prefix components of the updated rule is stored in exactly one location at maximum depth $O(dW)$ in the data structure. As an example, the path traversed by the classification query algorithm for an incoming packet (000,010) is also shown in Figure 4.2.

2.6 Set-pruning tries

A set-pruning trie [106] is similar to a hierarchical trie but with reduced data structure query time obtained by eliminating the need for doing recursive traversals. This is achieved by replicating rules at several nodes in the data structure as follows. Consider a d-dimensional hierarchical trie consisting of an $F1$-trie and several $(d-1)$-dimensional hierarchical tries. Let S be the set of nodes representing prefixes longer than a prefix p in the $F1$-trie. A set-pruning trie is similar to this hierarchical trie except that the rules in the $(d-1)$-dimensional hierarchical trie linked to a prefix p in the $F1$-trie are "pushed down," i.e., replicated in the $(d-1)$-dimensional hierarchical tries linked to all the nodes in S. This "pushing-down" of prefixes is carried out recursively (during preprocessing) on the remaining $(d-1)$ dimensions in the set-pruning trie data structure.

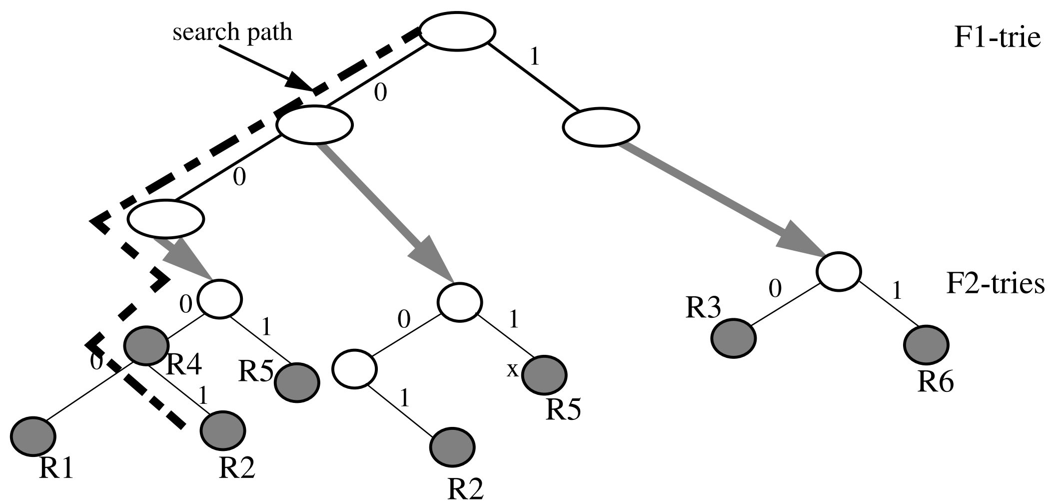


Figure 4.3 The set-pruning trie data structure built on the rules of example classifier of Table 4.1. The gray pointers are the "next-trie" pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is also shown.

The query algorithm for an incoming packet $(v_1, v_2, \ldots, v_d)$ now need only traverse the $F1$-trie to find the longest matching prefix of v_1, follow its next-trie pointer (if non-null), traverse the $F2$-trie to find the longest matching prefix of v_1, and so on for all dimensions. The manner of replication of rules ensures that every matching rule will be encountered in this path. The query time complexity reduces to $O(dW)$ at the expense of an increased storage complexity of $O(N^d dW)$ since a rule may need to be replicated $O(N^d)$ times — for every dimension k, the k^{th} prefix component of a rule may be longer than $O(N)$ other k^{th} prefix components of other rules in the classifier. Update complexity is $O(N^d)$, and hence, this data structure is, practically speaking, static.

The set-pruning trie for the example classifier of Table 4.1 is shown in Figure 4.3. The path traversed by the query algorithm on an incoming packet (000,010) is also shown. Note that replication may lead to prefix components of different rules being allocated to the same trie node. When this happens, only the highest priority rule need be stored at that

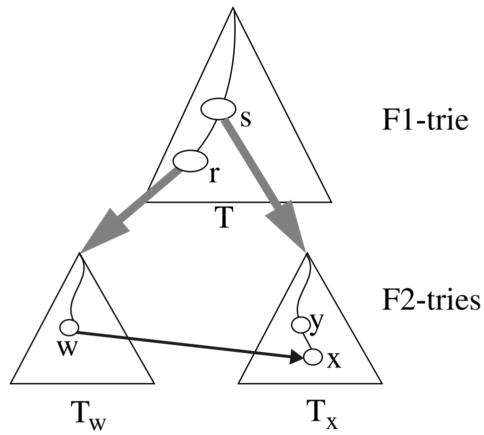


Figure 4.4 Showing the conditions under which a switch pointer is drawn from node w to node x. The pointers out of nodes s and r to tries T_x and T_w respectively are next-trie pointers.

node — for instance, both R5 and R6 are allocated to node x in the $F2$-trie of Figure 4.4, but the node x stores only the higher priority rule R5.

2.7 Grid-of-tries

The grid-of-tries data structure, proposed by Srinivasan et al [95], is an optimization of the hierarchical trie data structure for two dimensions. This data structure avoids the memory blowup of set-pruning tries by allocating a rule to only one trie node as in hierarchical tries. However, it still achieves $O(W)$ query time by using pre-computation and storing a *switch pointer* in some trie nodes. A switch pointer is labeled ‘0’ or ‘1’ and guides the search process in the manner described below. The conditions which must be satisfied for a switch pointer labeled b (b = ’0’ or ‘1’) to exist from a node w in the trie T_w to a node x of another trie T_x are (see Figure 4.4):

1. T_x and T_w are distinct tries built on the prefix components of dimension $F2$. Furthermore, T_x and T_w are respectively pointed to by the next-trie pointers of two distinct nodes, say r and s of the same trie, T, built on prefix components of dimension $F1$.

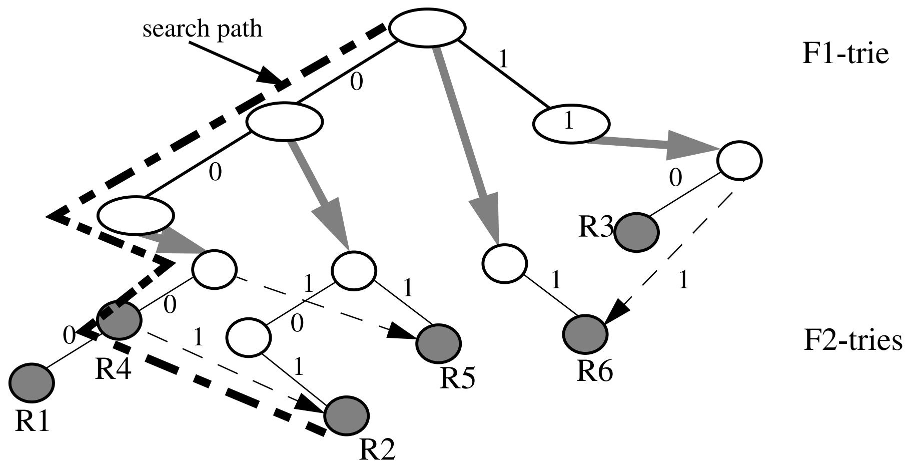


Figure 4.5 The grid-of-tries data structure built on the rules of example classifier in Table 4.1. The gray pointers are the "next-trie" pointers, and the dashed pointers are the switch pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is also shown.

2. The bit-string that denotes the path from the root node to node w in trie T_w concatenated with the bit b is identical to the bit-string that denotes the path from the root node to node x in the trie T_x.
3. Node w does not have a child pointer labeled b, and
4. Node s in trie T is the closest ancestor of node r that satisfies the above conditions.

If the query algorithm traverses paths $U1(s, root(T_x), y, x)$ and $U2(r, root(T_w), w)$ for an incoming packet on the hierarchical trie, the query algorithm need only traverse the path $U(s, r, root(T_w), w, x)$ on a grid-of-tries data structure. This is because paths $U1$ and $U2$ are identical (by condition 2 above) till $U1$ terminates at node w because w does not have a child branch labeled b (by condition 3). The use of another pointer, called a "switch pointer," from node w directly to node x allows the grid-of-tries query algorithm to traverse all branches that would have been traversed by the hierarchical trie query algorithm without the need to ever backtrack. This new algorithm examines each bit of the incoming packet header at most once. Hence, the time complexity reduces to $O(W)$, while storage complexity of $O(NW)$ remains identical to that of 2-dimensional hierarchical tries.

However, adding switch pointers to the hierarchical trie data structure makes incremental updates difficult to support, so the authors recommend rebuilding the data structure (in time $O(NW)$) in order to carry out updates [95]. The grid-of-tries data structure for the example classifier of Table 4.1 is shown in Figure 4.5, along with an example path traversed by the query algorithm.

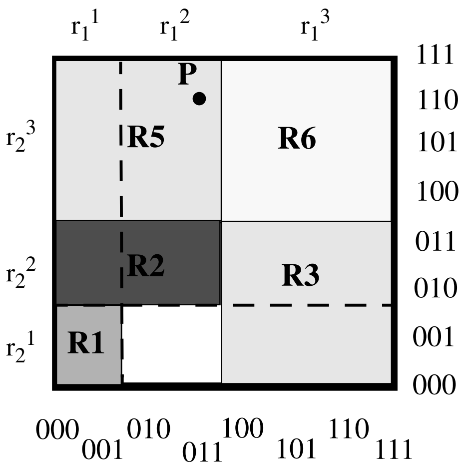
Reference [95] reports a memory usage of 2 Mbytes on a classifier containing 20,000 rules in two dimensions comprising destination and source IP prefixes, when the stride of the destination prefix trie is 8 bits and that of the source prefix tries is 5 bits. The worst case number of memory accesses is therefore 9. The classifier was constructed by using a publicly available routing table for the destination IP dimension and choosing prefixes from this routing table randomly to form the source IP dimension.

Grid-of-tries is a good data structure for two dimensional classification occupying reasonable amount of memory and requiring a few memory accesses. It can be used as an optimization for the last two dimensions of a multi-dimensional hierarchical trie, hence decreasing the classification time complexity by a factor of W to $O(NW^{d-1})$ in d dimensions, in the same amount of storage $O(NdW)$. As with hierarchical and set-pruning tries, grid-of-tries requires range specifications to be split into prefixes before the data structure is constructed.

2.8 Crossproducting

Crossproducting [95] is a packet classification solution suitable for an arbitrary number of dimensions. The idea is to classify an incoming packet in d dimensions by composing the results of separate 1-dimensional range lookups in each dimension as follows.

The preprocessing step to construct the data structure comprises computing the set of ranges, G_k, of size $s_k = |G_k|$, projected by rule specifications in each dimension k, $1 \le k \le d$. Let r_k^j, $1 \le j \le s_k$, denote the j^{th} range in G_k. A crossproduct table C_T of size



Crossproduct Table

(r_1^1, r_2^1)	**R1**
(r_1^1, r_2^2)	**R2**
(r_1^1, r_2^3)	**R5**
(r_1^2, r_2^1)	—
(r_1^2, r_2^2)	**R2**
(r_1^2, r_2^2)	**R5**
(r_1^3, r_2^1)	**R3**
(r_1^3, r_2^2)	**R3**
(r_1^3, r_2^3)	**R5**

Figure 4.6 The table produced by the crossproducting algorithm and its geometric representation of the two-dimensional classifier of Table 4.1.

$\prod_{k=1}^{d} s_k$ is then constructed, and the best matching rule for each entry $\left(r_1^{i_1}, r_2^{i_2}, \ldots, r_d^{i_d}\right)$, $1 \le i_k \le s_k$, $1 \le k \le d$ in this table is precomputed and stored.

Classification query on an incoming packet $(v_1, v_2, \ldots, v_d)$ first performs a range lookup in each dimension k to identify the range $r_k^{i_k}$ containing point v_k. The tuple $\langle r_1^{i_1}, r_2^{i_2}, \ldots, r_d^{i_d} \rangle$ is then directly looked up in the crossproduct table C_T to access the precomputed best matching rule.

Example 4.5: The crossproduct table for the example classifier of Table 4.1 is shown in Figure 4.6. The figure also illustrates the geometric interpretation of crossproducting. There is one entry in the crossproduct table for each rectangular cell in the grid created by extending the sides of each original rectangle representing a rule. The query algorithm for an example incoming packet P(011,110) accesses table entry with the address (r_1^2, r_2^3) accessing rule R5.

We have seen that N prefixes give rise to at most $2N$ ranges, hence, $s_k \le 2N$, and C_T is of size $O(N^d)$. The lookup time is $O(dt_{RL})$ where t_{RL} is the time complexity of doing a range lookup in one dimension. Crossproducting is a suitable solution for very small classifiers only because of its high worst case storage complexity. Reference [95] proposes using an on-demand crossproducting scheme together with caching for classifiers bigger than 50 rules in five dimensions. Crossproducting is a static solution since addition of a rule could change the set of projected ranges and necessitate re-computing the crossproduct table.

2.9 Bitmap-intersection

The bitmap-intersection classification scheme, proposed by Lakshman and Stiliadis [48], is based on the observation that the set of rules, S, that match a packet header, is the intersection of d sets, S_i, where S_i is the set of rules that match the packet in the i^{th} dimension alone. While crossproducting precomputes S and stores the best matching rule in S, this scheme computes S and the best matching rule on the fly, i.e., during each classification operation.

In order to compute intersection of sets efficiently in hardware, each set is encoded as an N-bit bitmap with one bit corresponding to each of the N rules. The set of matching rules is then the set of rules whose corresponding bits are ‘1’ in the bitmap. A classification query on a packet, P, proceeds in a fashion similar to crossproducting by first performing separate range lookups in each of the d dimensions. Each range lookup returns a bitmap encoding the set of matching rules (precomputed for each range) in that dimension. The d sets are intersected (by a simple hardware boolean AND operation) to give the set of rules that match P. The best matching rule is then computed from this set. See Figure 4.7 for the bitmaps corresponding to the example classifier of Table 4.1.

Since each bitmap is N bits wide, and there are $O(N)$ of ranges in each of the d dimensions, the total amount of storage space consumed is $O(dN^2)$. The classification time complexity is $O(dt_{RL} + dN/w)$ where t_{RL} is the time to do one range lookup and w is the memory width so that it takes N/w memory operations to access one bitmap. Time com-

Dimension 1

r_1^1	{R1,R2,R4,R5,R6}	110111
r_1^2	{R2,R5,R6}	010011
r_1^3	{R3,R6}	001001

Dimension 2

r_2^1	{R1,R3,R4}	101100
r_2^2	{R2,R3,R4}	011000
r_2^3	{R5,R6}	000111

Query on P(011,010):

010011 Dimension-1 bitmap
000111 Dimension-2 bitmap

000011 Intersected bitmap
R5 Best matching rule

Figure 4.7 The bitmap tables used in the "bitmap-intersection" classification scheme for the example classifier of Table 4.1. See Figure 4.6 for a description of the ranges. Also shown is classification query on an example packet P(011, 110).

plexity can be brought down by a factor of d by using parallelism in hardware to lookup each dimension independently in parallel. Incremental updates are not supported. The same scheme can be implemented in software, but the classification time is expected to be higher because of the unavailability of hardware-specific features, such as parallelism and bitmap-intersection.

Reference [48] reports that the scheme could support up to 512 rules with a 33 MHz FPGA device and five 1 Mbit SRAMs, classifying one million packets per second. The scheme works well for a small number of rules in multiple dimensions, but suffers from a quadratic increase in storage space and a linear increase in memory bandwidth requirements (and hence in classification time) with the size of the classifier. A variation is described in [48] that decreases the storage requirement at the expense of increased classification time.

2.10 Tuple space search

The idea of the basic tuple space search algorithm (Suri et al [96]) is to decompose a classification query into a number of exact match queries. The algorithm first maps each d-dimensional rule into a d-tuple whose i^{th} component stores the length of the prefix

Rule	Specification	Tuple
R1	(00*,00*)	(2,2)
R2	(0**,01*)	(1,2)
R3	(1**,0**)	(1,1)
R4	(00*,0**)	(2,1)
R5	(0**,1**)	(1,1)
R6	(***,1**)	(0,1)

Tuple	Hash Table Entries
(0,1)	{R6}
(1,1)	{R3,R5}
(1,2)	{R2}
(2,1)	{R4}
(2,2)	{R1}

Figure 4.8 The tuples and associated hash tables in the tuple space search scheme for the example classifier of Table 4.1.

specified in the i^{th} dimension (the scheme supports only prefix specifications). Hence, the set of rules mapped to the same tuple are of a fixed and known length, and thus stored in a hash table for exact match query operations. A classification query is carried out by performing exact match operations on each of the hash tables corresponding to all possible tuples in the classifier. The tuples and their corresponding hash tables for the example classifier of Table 4.1 are shown in Figure 4.8. A variation of the basic algorithm uses heuristics to avoid searching all hash tables using ideas similar to those used in the “binary search on prefix lengths” lookup scheme mentioned in Section 2.2.5 of Chapter 2 (see [96] for details).

Classification time in the tuple space search scheme is equal to the time needed for M hashed memory accesses, where M is the number of tuples in the classifier. The scheme uses $O(N)$ storage since each rule is stored in exactly one hash table. Incremental updates are supported and require just one hashed memory access to the hash table associated with the tuple of the modified rule. In summary, the tuple space search algorithm performs well for multiple dimensions in the average case if the number of tuples is small. However, the use of hashing makes the time complexity of searches and updates non-deterministic. Also, the number of tuples could be very large, up to $O(W^d)$, in the worst case. Furthermore, since the scheme supports only prefixes, the storage complexity increases by a factor of $O(W^d)$ for generic rules as each range could be split into $O(W)$ prefixes in the

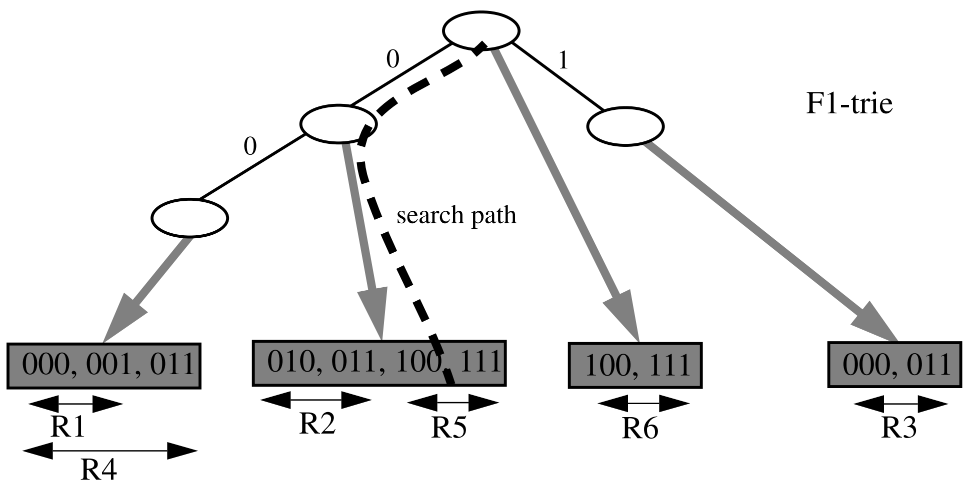


Figure 4.9 The data structure of Section 2.11 for the example classifier of Table 4.1 The search path for example packet P(011, 110) resulting in R5 is also shown.

manner explained in Section 2.1. This is one example where the range-to-prefix transformation technique of [23] cannot be applied because all fields are looked up simultaneously.

2.11 A 2-dimensional classification scheme from Lakshman and Stiliadis [48]

Lakshman and Stiliadis [48] propose a 2-dimensional classification algorithm where one dimension, say $F1$, is restricted to having prefix specifications, while the second dimension, $F2$, is allowed to have arbitrary range specifications. The data structure first builds an $F1$-trie on the prefixes of dimension $F1$, and then associates a set G_w of non-overlapping ranges to each trie node, w, that represents prefix p. These ranges are created by the end-points of possibly overlapping projections on dimension $F2$ of those rules, S_w, that specify exactly p in dimension $F1$. A range lookup data structure (e.g., an array or a binary search tree) is then constructed on G_w and associated with trie node w. The data structure for the example classifier of Table 4.1 is shown in Figure 4.9.

Given a point $P(v_1,v_2)$, the query algorithm proceeds downwards from the root of the trie according to the bits of v_1 in the usual manner. At every trie node, w, encountered during this traversal, a range lookup is performed on the associated data structure G_w. The range lookup operation returns the range in G_w containing v_2, and hence the best matching rule, say R_w, within the set S_w that matches point P. The highest priority rule among the rules $\{R_w\}$ for all trie nodes w encountered during the traversal is the desired highest priority matching rule in the classifier.

The query algorithm takes time $O(W \log N)$ because a range lookup needs to be performed (in $O(\log N)$ time) at every trie node in the path from the root to a null node in the $F1$-trie. This can be improved to $O(W + \log N)$ using a technique called *fractional cascading* borrowed from Computational Geometry [4]. This technique augments the data structure such that the problem of searching for the same point in several sorted lists is reduced to searching in only one sorted list plus accessing a constant number of elements in the remaining lists. The storage complexity is $O(NW)$ because each rule is stored only once in the data structure. However, the use of fractional cascading renders the data structure static.

2.12 Area-based quadtree

The Area-based Quadtree (AQT) data structure proposed by Buddhikot et al [7] for classification in two dimensions supports incremental updates that can be traded off with classification time by a tunable parameter. The preprocessing algorithm first builds a quadtree [4], a tree in which each internal node has four children. The parent node of a quadtree represents a two dimensional space that is decomposed into four equal sized quadrants, each of which is represented by a child of that node. The original two dimensional space is thus recursively decomposed into four equal-sized quadrants till each quadrant has less than or equal to one rule in it (see Figure 4.10 for an example of the

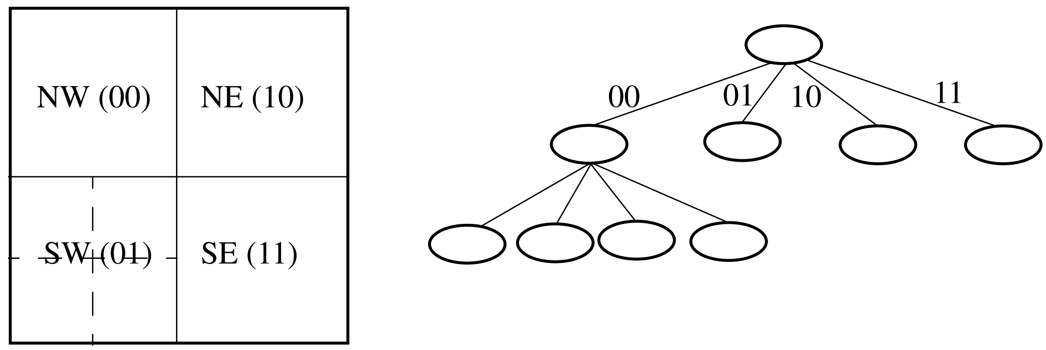


Figure 4.10 An example quadtree constructed by spatial decomposition of two-dimensional space. Each decomposition results in four quadrants.

decomposition process). A set of rules is then allocated to each node of the quadtree in the manner described next.

A rule is said to cross a quadrant in dimension j if it completely spans the dimension-j of the area represented by that quadrant. For instance, rule R6 spans in both dimensions the quadrant represented by the root node (the complete 2-dimensional space) of Figure 4.11, while rule R5 does not. If we divide the 2-dimensional space into four quadrants, rule R5 crosses the north-west quadrant in both dimensions while rule R2 crosses the south-west quadrant in dimension-$F1$. The set of rules crossing the quadrant represented by a node in dimension k is called the "k-crossing filter set (k-CFS)" of that node.

Two instances of the same data structure are associated with each quadtree node — one each for storing the rules in k-CFS ($k = 1, 2$). Since rules in crossing filter sets span at least one of the two dimensions, only the range specified in the other dimension need be stored in the data structure. The classification query proceeds by traversing the quadtree according to the bits in the given packet — looking at two bits at a time, formed by transposing one bit from each dimension. The query algorithm does two 1-dimensional look-

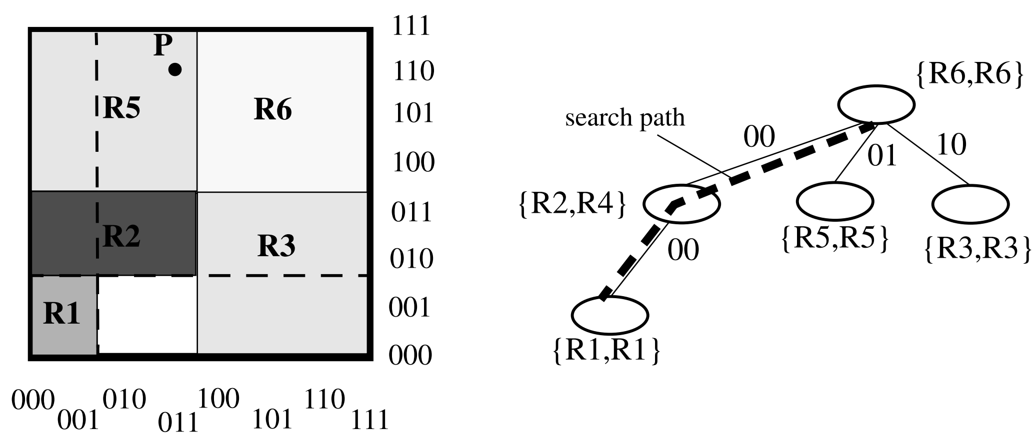


Figure 4.11 The AQT data structure for the classifier of Table 4.1. The label of each node denotes {1-CFS, 2-CFS}. Also shown is the path traversed by the query algorithm for an incoming packet P(001, 010), yielding R1 as the best matching rule.

ups (one for each dimension on k-CFS) at each quadtree node traversed. Figure 4.11 shows the AQT data structure for the example classifier of Table 4.1.

Reference [7] also proposes an efficient incremental update algorithm that enables AQT to achieve the following bounds for N two-dimensional rules: $O(NW)$ space complexity, $O(\alpha W)$ search time and $O(\alpha \sqrt[\alpha]{N})$ update time for a tunable integral parameter α.

2.13 Fat Inverted Segment Tree (FIS-tree)

Feldmann and Muthukrishnan [23] propose the FIS-tree data structure for two dimensional classification as a modification of the segment tree data structure. We first describe the segment tree data structure, and then the FIS-tree data structure.

A segment tree [4] stores a set S of line segments (possibly overlapping) to answer queries such as finding the highest priority line segment containing a given point efficiently. It consists of a balanced binary search tree on the end points of the line segments in S. Each node, w, of a segment tree represents a range G_w — leaves represent the original line segments in S, and parent nodes represent the union of the ranges represented by

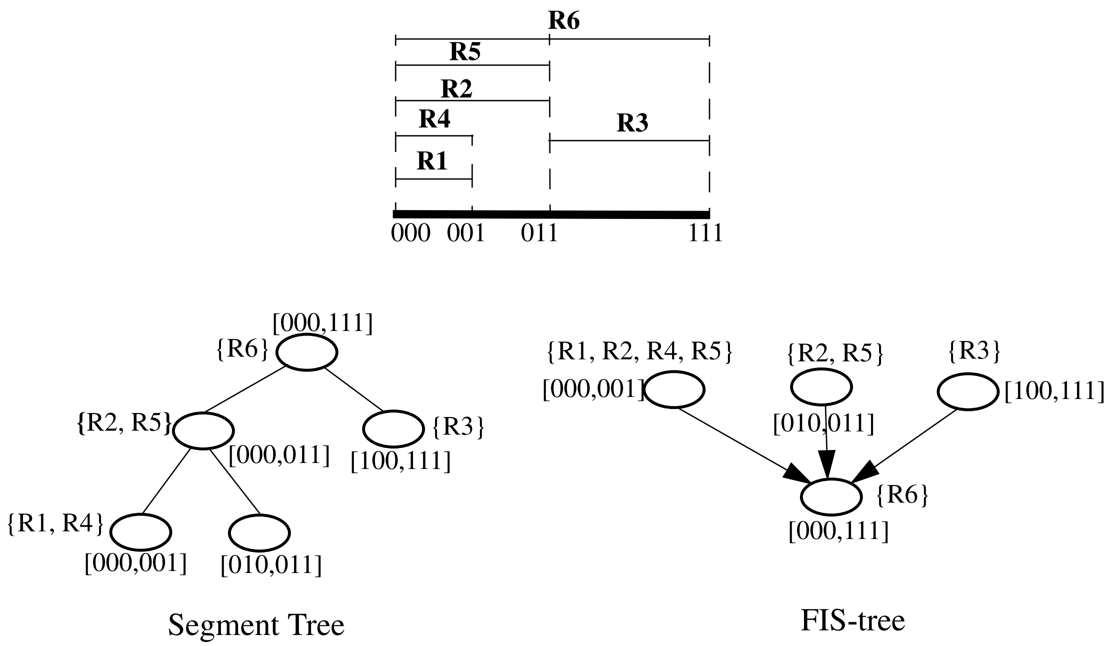


Figure 4.12 The segment tree and the 2-level FIS-tree for the classifier of Table 4.1.

their children. A line segment is allocated to a node w if it contains G_w but does not contain $G_{parent(w)}$. The highest priority line segment among all the line segments allocated to a node is precomputed and stored at the node. The search algorithm for finding the highest priority line segment containing a given point traverses the segment tree downwards from the root, and calculates the highest priority of all the precomputed segments encountered at each node during its traversal. Figure 4.12 shows the segment tree for the line segments created by the $F1$-projections of the rules of classifier in Table 4.1.

An FIS-tree is a segment tree with two modifications: (1) The segment tree is compressed (made "fat" by increasing the degree to more than two) in order to decrease its depth so that it occupies a given number of levels l. (2) Pointers are set up inverted, i.e., go from child nodes to the parent to help the search process described below. The classification data structure for 2-dimensional classifiers consists of an FIS-tree on dimension $F1$, and a range lookup data structure associated with each node of the FIS-tree. An instance of the range lookup data structure associated with node w of the FIS-tree stores the ranges formed by the $F2$-projections of those classifier rules whose $F1$-projections were allocated to w.

A classification query on a given point $P(v_1,v_2)$ first solves the range lookup problem in dimension $F1$. This returns a leaf node of the FIS-tree representing the range containing the point v_1. The query algorithm then follows the parent pointers from this leaf node up towards the root node, carrying out 1-dimensional range lookups in the associated range lookup data structures at each node traversed. The algorithm finally computes the highest priority rule containing the given point at the end of the traversal.

The search time complexity for an l-level FIS-tree is $O((l+1)\,t_{RL})$ with a storage space complexity of $O(ln^{1+1/l})$, where t_{RL} is the time taken to carry out a 1-dimensional range lookup. Storage space can be traded off with search time by suitably tuning the parameter l. Several variations to the FIS-tree are needed in order to support incremental updates — even then, it is easier to support inserts than deletes [23]. The static FIS-tree can be extended to multiple dimensions by building hierarchical FIS-trees, but the bounds obtained are similar to other data structures studied earlier. (Please see [23] for details on supporting updates in FIS trees and multi-dimensional static FIS trees).

Extensive measurements on real-life 2-dimensional classifiers are reported in [23] using the static FIS-tree data structure. These measurements indicate that two levels suffice in the FIS tree for 4-60K rules with a storage consumption of less than 5 Mbytes. One classification operation requires fewer than 15 memory accesses. For larger classifiers containing up to one million 2-dimensional rules, at least 3 levels are required with a storage consumption of approximately 100 Mbytes, while one classification operation requires fewer than 18 memory accesses.

2.14 Summary of previous work

Table 4.3 gives a summary of the complexities of the multi-dimensional classification algorithms reviewed in this chapter. Most proposed algorithms work well for two dimensions, but do not extend to multiple dimensions. Others have either non-deterministic search time (e.g., tuple space search), or do not scale to classifiers larger than a few hundred rules (e.g., crossproducting or bitmap-intersection). This is not surprising since theoretical bounds tell us that multi-dimensional classification has poor worst-case performance, in either storage or time complexity.

TABLE 4.3. Comparison of the complexities of previously proposed multi-dimensional classification algorithms on a classifier with N rules and d W-bit wide dimensions. The results assume that each rule is stored in $O(1)$ space and takes $O(1)$ time to determine whether it matches a packet. This table ignores the multiplicative factor of $(2W-2)^d$ in the storage complexity caused by splitting of ranges to prefixes.

Algorithm	Worst-case time complexity	Worst-case storage complexity
Linear Search	N	N
Hierarchical tries	W^d	NdW
Set-pruning tries	dW	$N^d dW$
Grid-of-tries	W^{d-1}	NdW
Crossproducting	dW	N^d
Bitmap-intersection	$(W + N/memwidth)d$	dN^2
Tuple space search	N	N
FIS-tree	$(l+1)W$	$l \times N^{1+1/l}$
Ternary CAM	1	N

3 Proposed algorithm RFC (Recursive Flow Classification)

3.1 Background

The RFC algorithm is motivated by the observation that real-life classifiers contain a large amount of structure and redundancy that can be exploited by a pragmatic classification algorithm. RFC works well for a selection of multi-dimensional real-life classifiers available to us. We proceed to describe the observed characteristics of these real-life classifiers and a description of the structure present in them.

3.2 Characteristics of real-life classifiers

We collected 793 packet classifiers from 101 different ISP and enterprise networks with a total of 41,505 rules. For privacy reasons, sensitive information such as IP addresses were sanitized while preserving the relative structure in the classifiers.[1] Each network provided up to ten separate classifiers for different services.[2] We found the classifiers to have the following characteristics:

1. The classifiers do not contain a large number of rules. Only 0.7% of the classifiers contain more than 1000 rules, with a mean of 50 rules. The distribution of the number of rules in a classifier is shown in Figure 4.13. The relatively small number of rules per classifier should not come as a surprise: in most networks today, rules are configured manually by network operators, and it is a non-trivial task to ensure correct behavior if the classifier becomes large.
2. The syntax of these classifiers allows a maximum of 8 header fields to be specified: source/destination network-layer address (32-bits), source/destination transport-layer port numbers (16-bits for TCP and UDP), type-of-service (TOS) field

1. We wanted to preserve the properties of set relationship, e.g. inclusion, among the rules, or their fields. A 32-bit IP address *p0.p1.p2.p3* is sanitized as follows: (a) A random 32-bit number *c0.c1.c2.c3* is first chosen, (b) a random permutation of the 256 numbers 0...255 is then generated to get *perm[0..255]* (c) Another random number *S* between 0 and 255 is generated: these randomly generated numbers are common for all the rules in the classifier, (d) The IP address with bytes: *perm[(p0 ^ c0 + 0 * s) % 256]*, *perm[(p1 ^ c1 + 1 * s) % 256]*, *perm[(p2 ^ c2 + 2 * s) % 256]* and *perm[(p3 ^ c3 + 3 * s) % 256]* is then returned as the sanitized transformation of the original IP address, where ^ denotes the exclusive-or operation. This transformation preserves set relationship across bytes but not necessarily within a byte. Hence, some structure present in the original classifier may be lost. However, we have since had access to some of the original classifiers, with results similar to those shown in this chapter.

2. In the collected dataset, classifiers for different services are made up of one or more ACLs (access control lists). An ACL rule can have one of two actions, “deny” or “permit”. In this discussion, we will assume that each ACL is a separate classifier, a common case in practice.

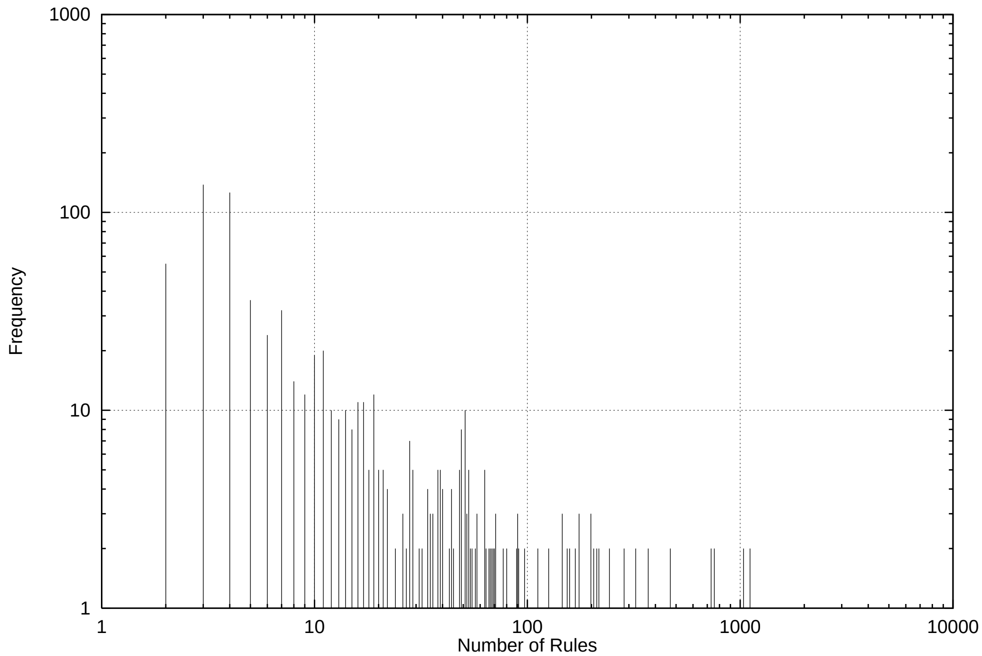


Figure 4.13 The distribution of the total number of rules per classifier. Note the logarithmic scale on both axes.

(8-bits), protocol field (8-bits), and transport-layer protocol flags (8-bits) with a total of 120 bits. 17% of all rules in the dataset have 1 field specified, 23% have 3 fields specified and 60% have 4 fields specified.[1]

3. The transport-layer protocol field is restricted to a small set of values: in our dataset, this field contained only the following values: TCP, UDP, ICMP, IGMP, (E)IGRP, GRE and IPINIP, or the wildcard '*' (i.e., the set of all transport protocols).

4. The transport-layer address fields have a wide variety of specifications. Many (10.2%) of them are *range* specifications — such as 'range 20-24' or 'gt 1023,' which means all values greater than 1023. In particular, the specification 'gt 1023' occurs in about 9% of the rules. Splitting this range into prefixes results in six constituent maximal prefixes: 1024-2047, 2048-4095, 4096-8191, 8192-16383, 16384-32767, 32768-65535. Thus, converting all range specifications to prefix specifications could result in a large increase in the size of a classifier.

1. If a field is not specified, the wildcard specification is assumed. Note that this is determined by the syntax of the rule specification language.

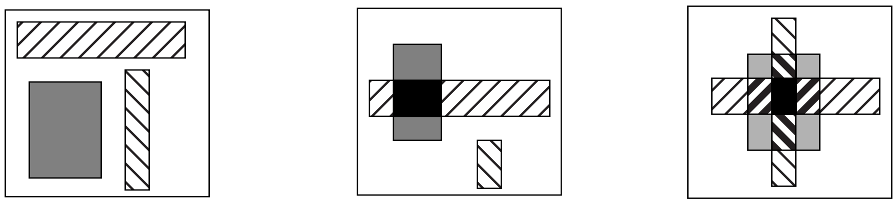
5. Approximately 14% of all classifiers had at least one rule with a non-contiguous mask, and 10.2% of all rules had non-contiguous masks. A non-contiguous mask means that the bits that are ‘1’ in the mask are not contiguous. For example, a specification of 137.98.217.0/8.22.160.80 has a non-contiguous mask, which is surprising. One suggested reason for this is that some network operators choose a specific numbering/addressing scheme for their routers. This observation indicates that a packet classification algorithm cannot always rely on a network-layer address specification to be a prefix.

6. It is common for different rules in the same classifier to share a number of field specifications. Sharing occurs because a network operator frequently wants to specify the same policy for a pair of communicating groups of hosts or subnetworks — for instance, the network operator may want to prevent every host in one group of IP addresses from accessing any host in another group of IP addresses. Given the limitations of a simple address/mask syntax specification, a separate rule must be written for each pair in the two (or more) groups. This observation is used in an optimization of the basic algorithm, described later in Section 5.1.

7. We found that 15% of the rules were redundant. A rule R is said to be redundant if one of the following conditions hold (here, we think of a rule R as the set of all packet headers which could match R): (a) There exists a rule T appearing earlier than R in the classifier such that R is a subset of T. Thus, no packet will ever match R, i.e., R is redundant. We call this *backward redundancy* — 7.8% of the rules were found to be backward redundant. (b) There exists a rule T appearing after R in the classifier such that (i) R is a subset of T, (ii) R and T have the same actions, and (iii) For each rule V appearing in between R and T in the classifier, either V is disjoint from R, or V has the same action as R. We call this *forward redundancy* — 7.2% of the rules were forward redundant. In this case, R can be eliminated to obtain a new smaller classifier. A packet matching R in the original classifier will match T in the new classifier, but will yield the same action.

3.3 Observations about the structure of the classifiers

To illustrate the structure we found in our dataset, we start with an example 2-dimensional classifier containing three rules. Figure 4.14(a) shows three such rectangles, where each rectangle represents a rule with a range of values in each dimension. The classifier contains three explicitly defined rules, and the default rule (represented by the background



(a) 4 regions (b) 5 regions (c) 7 regions

Figure 4.14 Some possible arrangements of three rectangles (2-dimensional rules). Each differently shaded rectangle comprises one region. The total number of regions indicated includes the white background region.

rectangle). The arrangement of the three rules in Figure 4.14(a) is such that four distinct regions, differently shaded, are created (including the white background region). A different arrangement could create five regions, as in Figure 4.14(b), or seven regions, as in Figure 4.14(c). A classification algorithm must keep a record of each region and be able to determine the region to which each newly arriving packet belongs. Intuitively, the larger the number of regions that the classifier contains, the more storage is required, and the longer it takes to classify a packet.

Even though the number of rules is the same in each of the three cases in Figure 4.14, the task of the classification algorithm becomes progressively harder as it needs to distinguish more regions. In general, it can be shown that the number of regions created by N rules in d dimensions can be $O(N^d)$. Such a worst case example for two dimensions is shown in Figure 4.15.

We analyzed the structure in our dataset and found that the number of overlapping regions is considerably smaller than the worst case. Specifically, for the biggest classifier with 1733 rules, the number of distinct overlapping regions in four dimensions was found to be 4316, compared to approximately 10^{11} regions for the worst possible combination of rules. Similarly, the number of overlapping regions was found to be relatively small in each of the classifiers in the dataset. This is because rules originate from specific policies

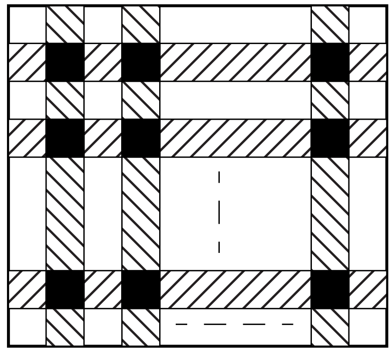


Figure 4.15 A worst case arrangement of N rectangles. $N/2$ rectangles span the first dimension, and the remaining $N/2$ rectangles span the second dimension. Each of the black squares is a distinct region. The total number of distinct regions is therefore $N^2/4 + N + 1 = O(N^2)$.

of network operators and agreements between different networks. For example, the operators of two different networks may specify several policies relating to the interaction of the hosts in one network with the hosts in the other. This implies that rules tend to be clustered in small groups instead of being randomly distributed. As we will see, the proposed algorithm exploits this structure to simplify its task.

3.4 The RFC algorithm

Classifying a packet can be viewed as mapping S bits in the packet header to T bits of *classID* (an identifier denoting the rule, or action), where $T = \log N$, $T \ll S$, in a manner dictated by the N classifier rules. A simple and fast, but unrealistic, way of doing this mapping might be to precompute the value of classID for each of the 2^S different packet header values. This would yield the answer in one step (i.e., one memory access) but would require too much memory. The main aim of RFC is to perform the same mapping but over several stages. As shown in Figure 4.16, RFC performs this mapping recursively — in each stage the algorithm performs a *reduction*, mapping one set of values to a smaller set.

The RFC algorithm has P *phases*, where each phase consists of a set of parallel memory lookups. Each lookup is a reduction in the sense that the value returned by the memory

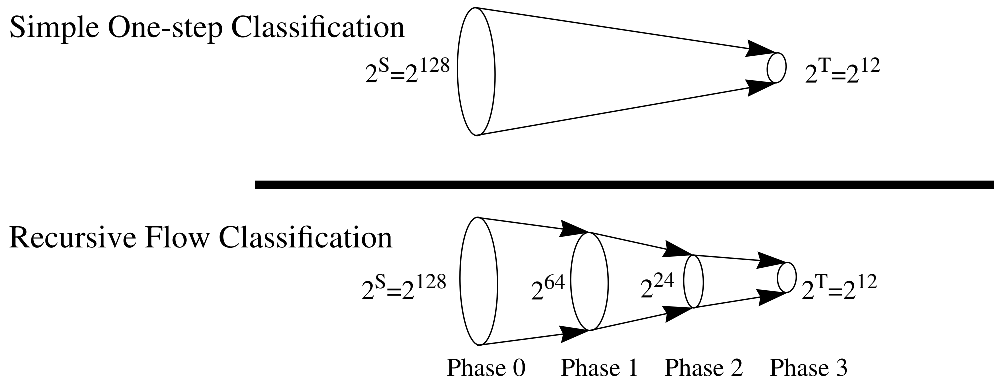


Figure 4.16 Showing the basic idea of Recursive Flow Classification. The reduction is carried out in multiple phases, with a reduction in phase *I* being carried out recursively on the image of the phase *I-1*. The example shows the mapping of 2^S bits to 2^T bits in 4 phases.

lookup is shorter (is expressed in fewer bits) than the index of the memory access. The algorithm, as illustrated in Figure 4.17, operates as follows:

1. In the first phase (phase 0), d fields of the packet header are split up into multiple chunks that are used to index into multiple memories in parallel. For example, the number of chunks equals 8 in Figure 4.17. Figure 4.18 shows an example of how the fields of a packet may be split into chunks. Each of the parallel lookups yields an output value that we will call *eqID*. (The reason for calling this identifier *eqID* will become clear shortly). The contents of each memory are chosen so that the result of the lookup is narrower than the index, i.e., requires fewer bits.
2. In subsequent phases, the index into each memory is formed by combining the results of the lookups from earlier phases. For example, the results from the lookups may be concatenated to form a wider index — we will consider another way to combine them later.
3. After successive combination and reduction, we are left with one result from the memory lookup in the final phase. Because of the way the memory contents have been precomputed, this value corresponds to the classID of the packet.

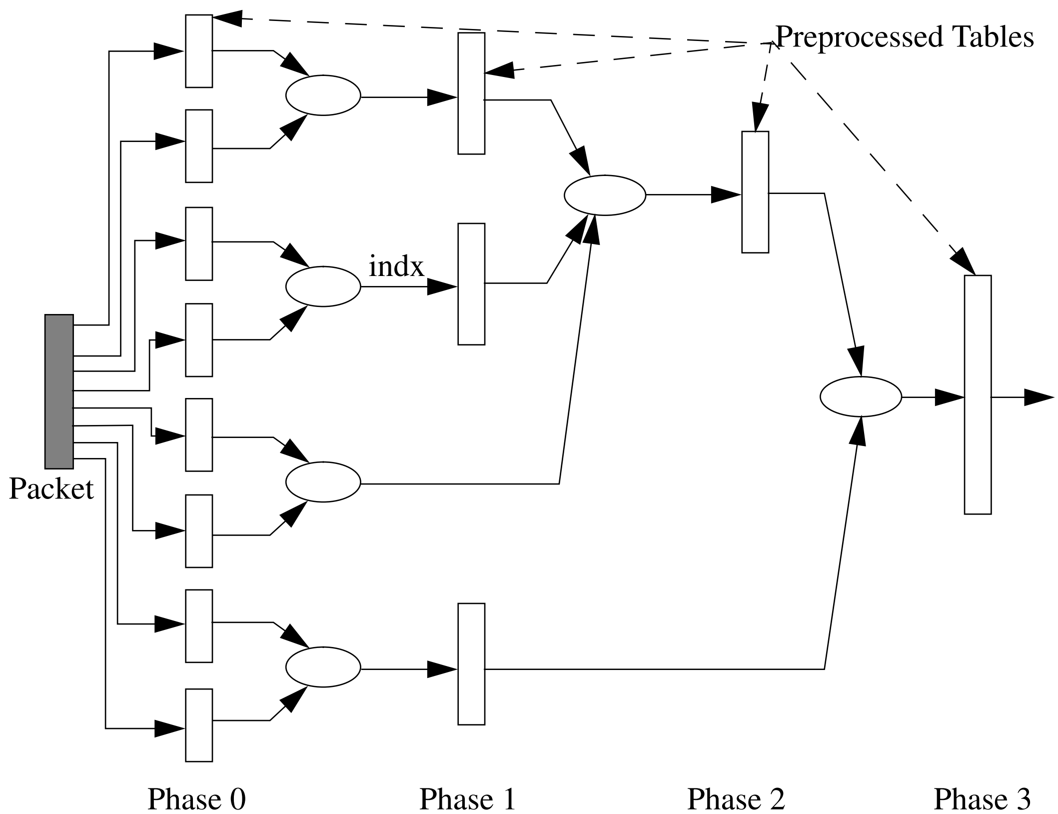


Figure 4.17 Packet flow in RFC.

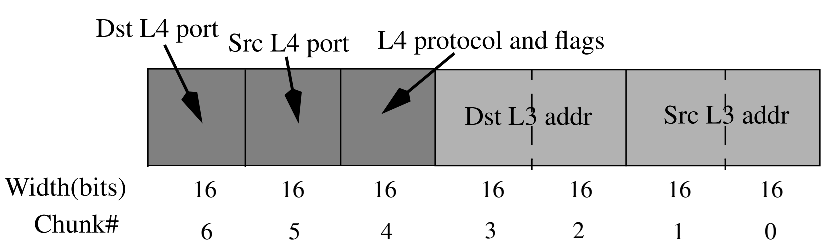


Figure 4.18 Example chopping of the packet header into chunks for the first RFC phase. L3 and L4 refer to the network-layer and transport-layer fields respectively.

For the above scheme to work, the contents of each memory are filled after suitably preprocessing the classifier. To illustrate how the memories are populated, we consider a simple example based on the classifier in Table 4.4.

TABLE 4.4. An example 4-dimensional classifier.

Dst L3 (value/mask)	Src L3 (value/mask)	Dst L4	L4 protocol
152.163.190.69/ 255.255.255.0	152.163.80.1/ 255.255.255.255	*	*
152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	udp
152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	range 20-21	udp
152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	tcp
152.168.3.198.4/ 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/ 255.255.255.0	gt 1023	tcp

We will see how the 24 bits used to express the two chunks: chunk #4 (L4, i.e., transport-layer protocol) and chunk #6 (Dst L4, i.e, transport-layer destination) are reduced to just three bits by Phases 0 and 1 of the RFC algorithm. We start with chunk #6, which contains the 16-bit transport-layer destination address. The column corresponding to the transport-layer field in Table 4.4 partitions the set of all possible chunk values into four sets: (a) {20, 21} (b) {http (=80)} (c) {>1023} (d) {all remaining numbers in the range 0-65535}. The four sets can be encoded using two bits 00 through 11. We call these two bit values the *equivalence class IDs* (*eqIDs*) of the respective sets. The memory corresponding to chunk #6, in Phase 0, is indexed using the 2^{16} different values of 16-bit wide chunk #6. In each memory location m, we place the *eqID* for the set containing the value m. For example, the value in the memory location 20 is 00, denoting the set {20,21}. In this way, a 16-bit to 2-bit reduction is obtained for chunk #6 in Phase 0. Similarly, the column corresponding to 8-bit transport-layer protocol in Table 4.4 consists of three sets: (a) {tcp} (b) {udp} (c) {all remaining protocol values in the range 0-255} — which can be encoded using two-bit *eqID*s. Hence, chunk #4 undergoes an eight-bit to two-bit reduction in Phase 0.

In the second phase (Phase 1), we consider the combination of the transport-layer Destination and protocol chunks. Table 4.4 shows that the five sets corresponding to the combination of these chunks are: (a) {({80}, {udp})} (b) {({20-21}, {udp})} (c) {({80}, {tcp})} (d) {({gt 1023}, {tcp})} (e) {all remaining crossproducts of the two columns}. The five sets can be represented using 3-bit *eqID*s. The index into the memory in Phase 1 is constructed by concatenating the two 2-bit *eqIDs* from Phase 0. Hence, Phase 1 reduces the number of bits from four to three. If we now consider the combination of both Phase 0 and Phase 1, we find that 24 bits have been reduced to just 3 bits. Hence, the RFC algorithm uses successive combination and reduction to map the long original packet header to a short classID.

We will now see how a classifier is preprocessed to generate the values to be stored in the memory tables at each phase. In what follows, we will use the term *Chunk Equivalence Set* (CES) to denote a set mentioned in the example above, e.g., each of the three sets: (a) {tcp} (b) {udp} (c) {all remaining protocol values in the range 0-255} is said to be a Chunk Equivalence Set because if there are two packets with different protocol values lying in the same set (and having otherwise identical headers), the rules of the classifier do not distinguish between them. Each CES can be constructed in the following manner.

First phase (Phase 0): The process of constructing a CES in a single dimension is similar to the procedure mentioned earlier for constructing non-overlapping basic intervals from the projections of the rules onto this dimension. The difference lies in that two non-contiguous ranges may now form a part of the same CES. Consider a fixed chunk of size b bits, and those component(s) of the rules in the classifier corresponding to this

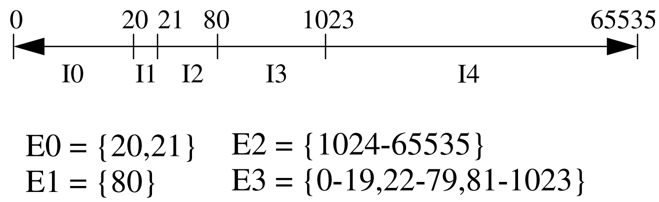


Figure 4.19 An example of computing the four equivalence classes E0...E3 for chunk #6 (corresponding to the 16-bit transport-layer destination port number) in the classifier of Table 4.4.

chunk. Project the rules in the classifier on 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. Also, two intervals are in the same equivalence set if exactly the same rules project onto them. As an example, consider chunk #6 (destination L4 port) of the classifier in Table 4.4. The intervals, $I0\ldots I4$, and the constructed equivalence sets, $E0\ldots E3$ are shown in Figure 4.19. The RFC table kept in the memory for this chunk is filled with the corresponding *eqID*s. Thus, in this example, *table[20]* $= 00$, *table[23]* $= 11$, etc. The pseudocode for computing the *eqID*s in Phase 0 is shown in Figure 4.20.

To facilitate the calculation of *eqIDs* for subsequent RFC phases, we assign a *class bitmap* (CBM) for each CES. The CBM has one bit for each rule in the classifier, and indicates those rules that contain the corresponding CES. For example, E0 in Figure 4.19 will have the CBM 101000, indicating that the first and the third rules of the classifier in Table 4.4 contain E0 in chunk #6. Note that the class bitmap is *not* physically stored in the RFC table: it is just used to facilitate the calculation of the stored *eqIDs* by the preprocessing algorithm.

Subsequent phases: A chunk in a subsequent phase is formed by a combination of two (or more) chunks obtained from memory lookups in previous phases. If, for example, the resulting chunk is of width b bits, we again create equivalence sets such that two b-bit

```
/* Phase 0, Chunk j of width b bits*/
for each rule rl in the classifier
begin
   project the i^th component of rl onto the number line [0,2^b – 1] , marking the start and end points of
   each of its constituent intervals.
endfor
/* Now scan through the number line looking for distinct equivalence classes */
bmp := 0; /* all bits of bmp are initialised to '0' */
for n in 0..2^b-1
   begin
   if (any rule starts or ends at n)
   begin
      update bmp;
      if (bmp not seen earlier)
      begin
         eq := new_equivalence_class();
         eq->cbm := bmp;
      endif
   endif
   else eq := the equivalence class whose cbm is bmp;
table_0_j[n] = eq->ID; /* fill ID in the rfc table*/
endfor
```

Figure 4.20 Pseudocode for RFC preprocessing for chunk j of Phase 0.

packet header values that are not distinguished by the rules of the classifier belong to the same CES. Hence, (20,udp) and (21,udp) will be in the same CES in the classifier of Table 4.4 in Phase 1. The new equivalence sets for a phase are determined by computing all possible intersections of equivalence sets from the previous phases being combined. Each distinct intersection is an equivalence set for the newly created chunk. The pseudocode for this preprocessing is shown in Figure 4.20.

3.5 A simple complete example of RFC

Realizing that the preprocessing steps are involved, we present a complete example of RFC operation on a classifier, showing how preprocessing is performed to determine the

```
    /* Assume that chunk i is formed by combining m distinct chunks c1, c2, ..., cm of phases p1,p2, ...,
pm where p1, p2, ..., pm < j */
    indx := 0; /* indx runs through all the entries of the RFC table, table_j_i */
    listEqs := nil;
    for each CES, c1eq, of chunk c1
    for each CES, c2eq, of chunk c2
    ........
    for each CES, cmeq of chunk cm
    begin
     intersectedBmp := c1eq->cbm & c2eq->cbm & ... & cmeq->cbm;/* bitwise ANDing */
     neweq := searchList(listEqs, intersectedBmp);
     if (not found in listEqs)
     begin
      /* create a new equivalence class */
      neweq := new_Equivalence_Class();
      neweq->cbm := bmp;
      add neweq to listEqs;
     endif
     /* Fill up the relevant RFC table contents.*/
     table_j_i[indx] := neweq->ID;
     indx++;
    endfor
```

Figure 4.21 Pseudocode for RFC preprocessing for chunk i of Phase j .

contents of the memories, and how a packet is looked up. The example is based on a 4-field classifier of Table 4.5 and is shown in Figure 4.22.

TABLE 4.5. The 4-dimensional classifier used in Figure 4.22.

Rule#	Chunk#0 (Src L3 bits 31..16)	Chunk#1 (Src L3 bits 15..0)	Chunk#2 (Dst L3 bits 31..16)	Chunk#3 (Dst L3 bits 15..0)	Chunk#4 (L4 protocol) [8 bits]	Chunk#5 (Dstn L4) [16 bits]
R0	0.83/0.0	0.77/0.0	0.0/0.0	4.6/0.0	udp (17)	*
R1	0.83/0.0	1.0/0.255	0.0/0.0	4.6/0.0	udp	range 20 30
R2	0.83/0.0	0.77/0.0	0.0/255.255	0.0/255.255	*	21
R3	0.0/255.255	0.0/255.255	0.0/255.255	0.0/255.255	*	21
R4	0.0/255.255	0.0/255.255	0.0/255.255	0.0/255.255	*	*

4 Performance of RFC

In this section, we look at the performance obtained by the RFC algorithm on the classifiers in our dataset. First, we consider the storage requirements of RFC. Then we consider its performance to determine the rate at which packets can be classified.

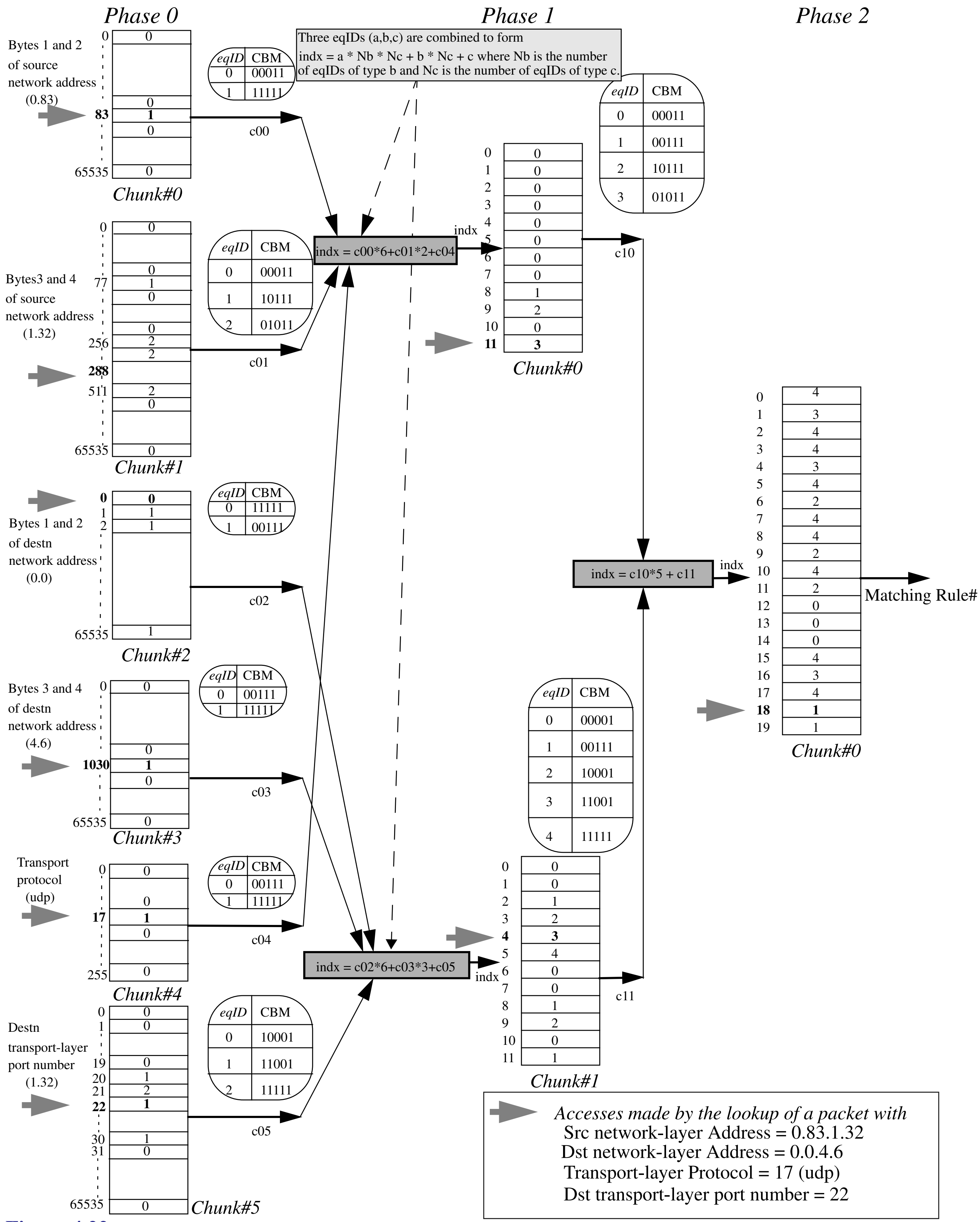


Figure 4.22 This figure shows the contents of RFC tables for the example classifier of Table 4.5. The sequence of accesses made by the example packet have also been shown using big gray arrows. The memory locations accessed in this sequence have been marked in bold.

4.1 RFC preprocessing

As our dataset has a maximum of four fields, the chunks for Phase 0 are created as shown in Table 4.6.

TABLE 4.6. Packet header fields corresponding to chunks for RFC Phase 0.

Chunk#	Field (subfield)
0	Source L3 address (most significant 16-bits)
1	Source L3 address (least significant 16-bits)
2	Destination L3 address (most significant 16-bits)
3	Destination L3 address (most significant 16-bits)
4	L4 protocol and flags
5	L4 destination port number

The performance of RFC (storage requirements and classification time) can be tuned with two parameters: (i) The number of phases, P, and (ii) The reduction tree used for a given P. For instance, two of the several possible reduction trees for $P = 3$ and $P = 4$ are shown in Figure 4.23 and Figure 4.24 respectively. (For $P = 2$, there is only one reduction tree possible.) When there is more than one reduction tree possible for a given value of P, the algorithm chooses a tree based on two heuristics: (i) Given a classifier, the maximum amount of pruning of the search space is likely to be obtained by combining those chunks together which have the most “correlation.” As an example, the combination of chunk 0 (most significant 16 bits of the source network address) and chunk 1 (least significant 16 bits of the source network address) in the toy example of Figure 4.22 would result in only 3 *eqID*s, while the combination of chunk 0 and chunk 4 (destination transport port number) would result in 5 *eqID*s. (ii) The algorithm combines as many chunks as it can without causing unreasonable memory consumption. Following these heuristics, we find that the “best” reduction tree for $P = 3$ is *tree_B* in Figure 4.23, and the “best” reduction tree for $P = 4$ is *tree_A* in Figure 4.24.[1]

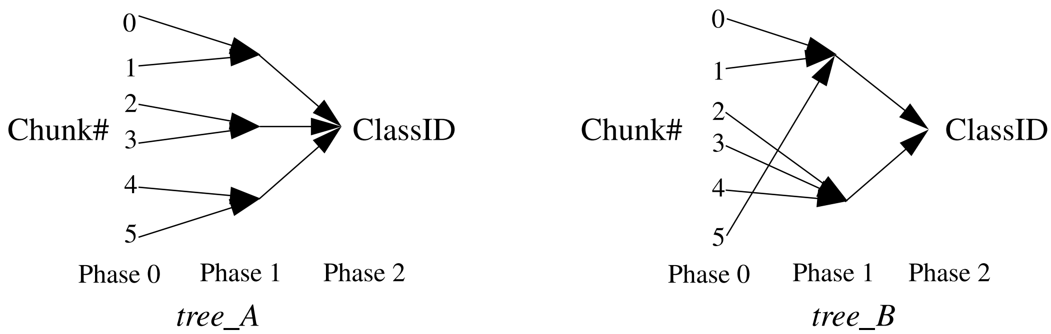


Figure 4.23 Two example reduction trees for three phases in RFC.

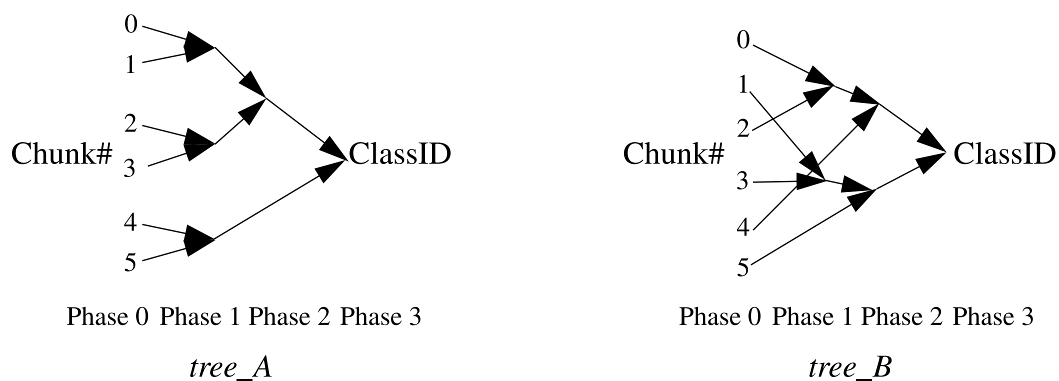


Figure 4.24 Two example reduction trees for four phases in RFC.

We now look at the performance of RFC on our dataset. Our first goal is to keep the total storage consumption small. The storage requirements for each of our classifiers is plotted in Figure 4.25, Figure 4.26 and Figure 4.27 for 2, 3 and 4 phases respectively.The graphs show how memory usage increases with the number of rules in each classifier. For practical purposes, it is assumed that memory is only available in widths of 8, 12 or 16 bits. Hence, an *eqID* requiring 13 bits is assumed to occupy 16 bits in the RFC table.

As we might expect, the graphs show that storage requirements decrease with an increase in the number of phases from three to four. However, this comes at the expense of

1. These reduction trees gave better performance results over other trees for a vast majority of the classifiers in our experiments.

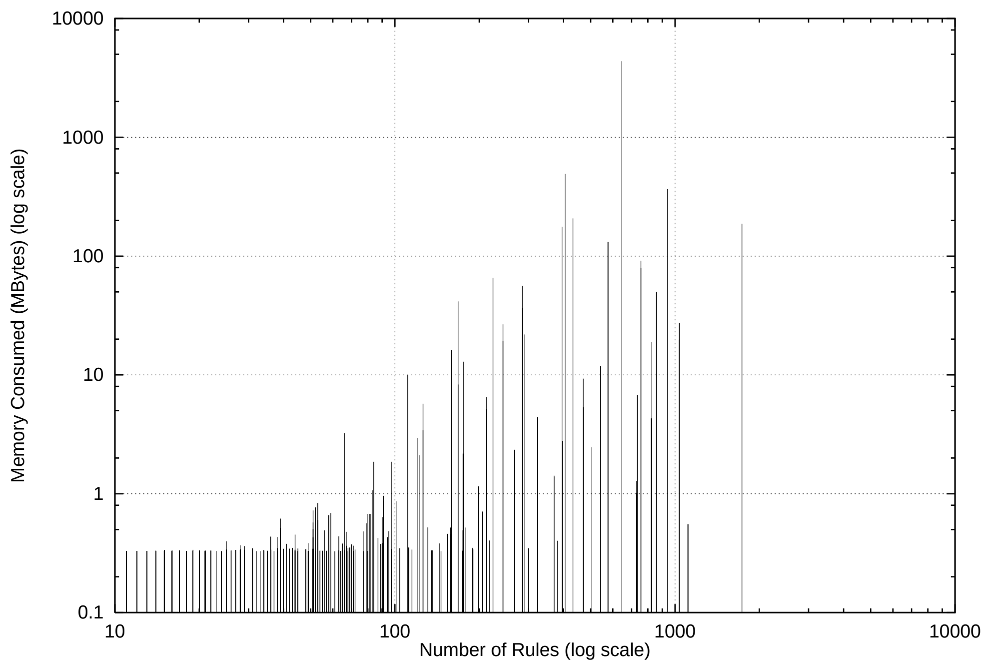


Figure 4.25 The RFC storage requirement in Megabytes for two phases using the dataset. This special case of RFC with two phases is identical to the Crossproducting method of [95].

two additional memory accesses, illustrating the trade-off between memory consumption and lookup time in RFC.

Like most algorithms in the literature, RFC does not support quick incremental updates, and may require rebuilding the data structure in the worst case. It turns out, however, that rebuilding is only necessary in the case of the addition of a new rule. Deletion of existing rules can be simply handled by changing the chunk equivalence sets of *eqID*s in the final phase. The performance of an implementation of such an incremental delete algorithm on random deletes is shown in Figure 4.28.

Our second goal is to keep the preprocessing time small — this is useful when updates necessitate rebuilding the data structure. Figure 4.29 plots the preprocessing time required for both three and four phases of RFC.[1] These graphs indicate that, if the data structure is

1. The case P=2 is not plotted: it was found to take hours of preprocessing time because of the unwieldy size of the RFC tables.

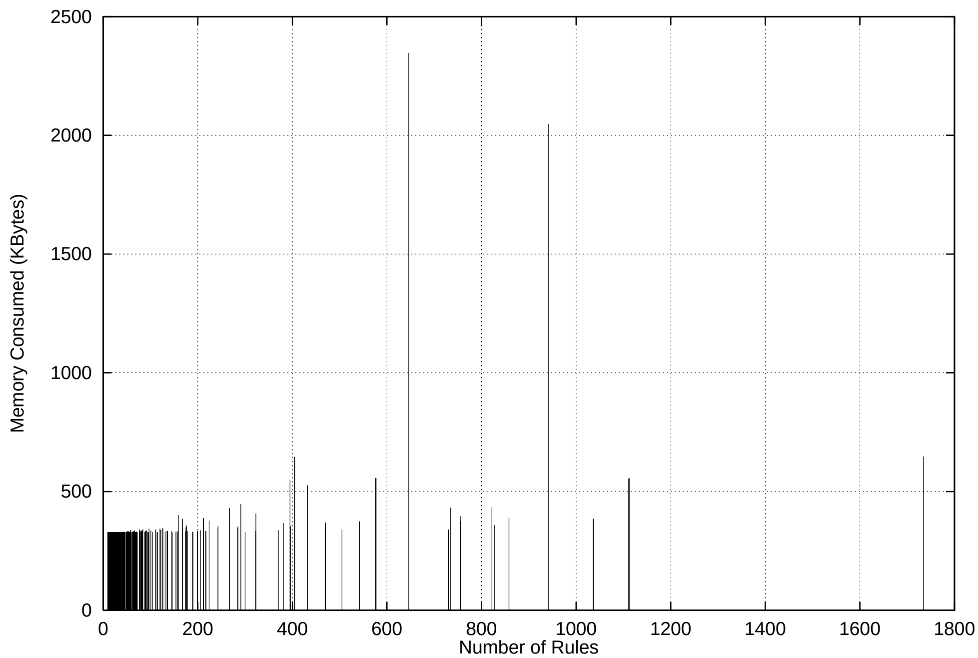


Figure 4.26 The RFC storage requirement in Kilobytes for three phases using the dataset. The reduction tree used is *tree_B* in Figure 4.23.

rebuilt on the addition of every rule, RFC may be suitable if (and only if) the rules change relatively slowly — for example, not more than once every few seconds. Thus, RFC may be suitable in environments where rules are changed infrequently; for example, if they are added manually, or on a router reboot.

Finally, note that there are some similarities between the RFC algorithm and the bitmap-intersection scheme of [48]; each distinct bitmap in [48] corresponds to a CES in the RFC algorithm. Also, note that when there are just two phases, RFC corresponds to the crossproducting method described in [95]. RFC is different from both these schemes in that it generalizes the concept of crossproducting to make storage requirements feasible for larger classifiers, along with a lookup time that scales better than that of the bitmap-intersection approach.

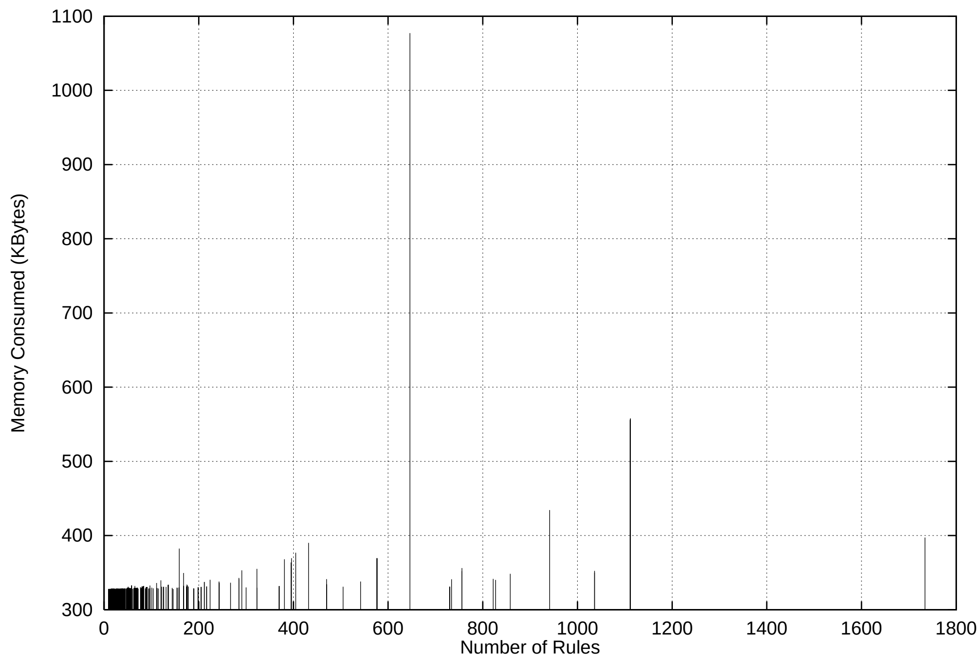


Figure 4.27 The RFC storage requirement in Kilobytes for four phases using the dataset. The reduction tree used is *tree_A* in Figure 4.24.

4.2 RFC lookup performance

The RFC lookup operation can be performed in hardware or in software.[1] We will discuss the lookup performance in each case separately.

4.2.1 Lookups in hardware

An example hardware implementation for the tree *tree_B* in Figure 4.23 (three phases) is illustrated in Figure 4.30 for four fields (six chunks in Phase 0). This design is suitable for all the classifiers in our dataset, and uses two 4 Mbit SRAMs and two 4-bank 64 Mbit SDRAMs clocked at 125 MHz.[2] The design is pipelined such that a new lookup may begin every four clock cycles.

1. Note that preprocessing is always performed in software.
2. These devices are in production in industry at the time of writing. In fact, even bigger and faster devices are available at the time of writing — see for example, reference [137].

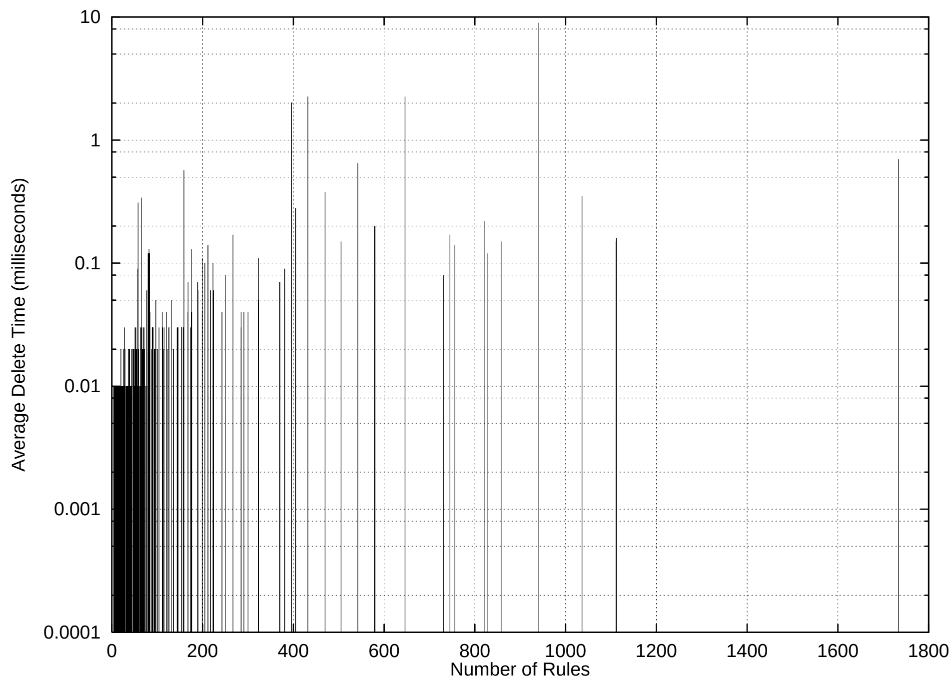


Figure 4.28 This graph shows the average amount of time taken by the incremental delete algorithm in milliseconds on the classifiers available to us. Rules deleted were chosen randomly from the classifier. The average is taken over 10,000 delete operations, and although not shown, variance was found to be less than 1% for all experiments. This data is taken on a 333 MHz Pentium-II PC running the Linux operating system.

The pipelined RFC lookup proceeds as follows:

1. **Pipeline Stage 0: Phase 0** *(Clock cycles 1-4)*: In the first three clock cycles, three accesses are made to the two SRAM devices in parallel to yield the six *eqID*s of Phase 0. In the fourth clock cycle, the *eqID*s from Phase 0 are combined to compute the two indices for the next phase.
2. **Pipeline Stage 1: Phase 1***(Clock cycles 5-8)*: The SDRAM devices can be accessed every two clock cycles, but we assume that a given bank can be accessed again only after eight clock cycles. By keeping the two memories for Phase 1 in different banks of the SDRAM, we can perform the Phase 1 lookups in four clock cycles. The data is replicated in the other two banks (i.e. two banks of memory hold a fully redundant copy of the lookup tables for Phase 1). This allows Phase 1 lookups to be performed on the next packet as soon as the current packet has completed. In this way, any given bank is accessed once every eight clock cycles.

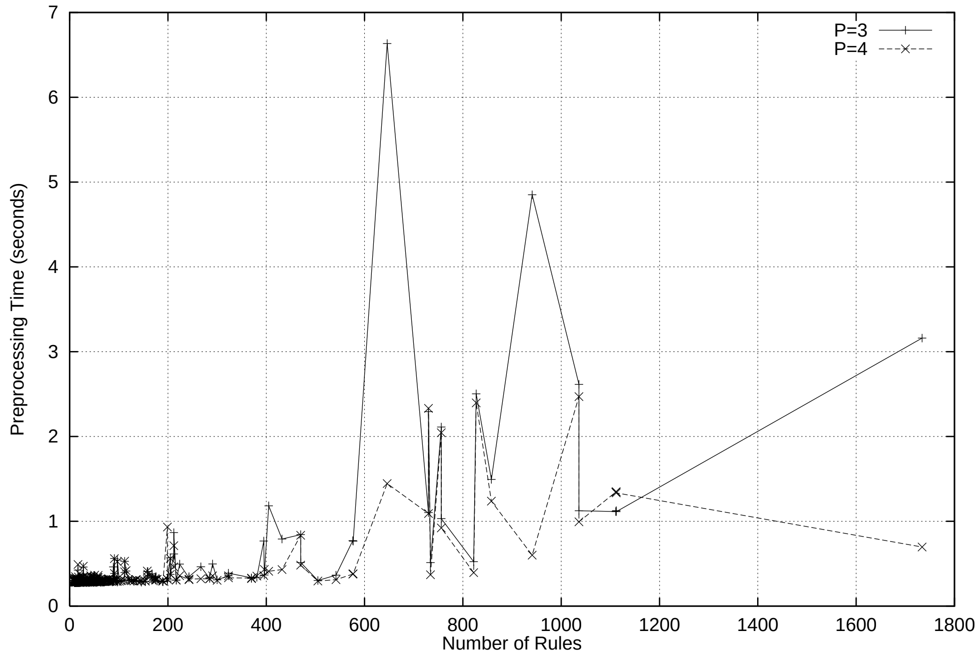


Figure 4.29 The preprocessing times for three and four phases in seconds, using the set of classifiers available to us. This data is taken by running the RFC preprocessing code on a 333 MHz Pentium-II PC running the Linux operating system.

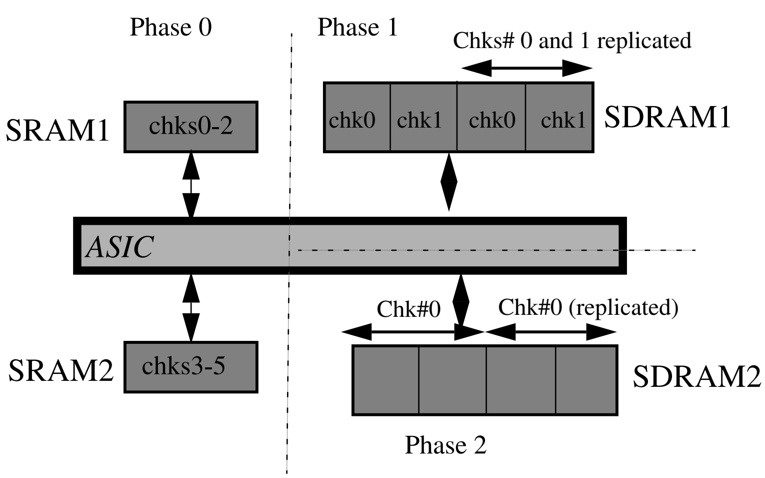


Figure 4.30 An example hardware design for RFC with three phases. The registers for holding data in the pipeline and the on-chip control logic are not shown. This design achieves OC192c rates in the worst case for 40 byte packets. The phases are pipelined with 4 clock cycles (at 125 MHz clock rate) per pipeline stage.

3. **Pipeline Stage 2: Phase 2** *(Clock cycles 9-12)*: Only one lookup is to be made. The operation is otherwise identical to Phase 1.

This design classifies approximately 30 million packets per second (to be exact, 31.25 million packets per second with a 125 MHz clock) with a total memory cost of approximately \$40.[1] This is fast enough to process minimum length TCP/IP packets at OC192 rates.

Discussion of how RFC exploits the structure in real-life classifiers

We saw in Section 3.3 that rules in real-life classifiers form a small number of overlapping regions and tend to cluster in small groups. The idea behind the *reduction* steps used in the RFC algorithm is to quickly narrow down the large search space to smaller subspaces containing these clusters. In order to do this without consuming too much storage, the reduction is carried out on small-sized chunks. However, the whole packet header needs to be looked at in order to prune the search space completely to arrive at the best matching rule — this is the purpose of the *combination* steps used in the RFC algorithm that incrementally combine a few chunks at a time till the whole packet header has been considered. Because the rules form a small number of overlapping regions, combining results of the reduction steps creates chunks that are still small enough to keep the total storage requirements reasonable.

Discussion of hardware implementation of RFC

We have seen that lower bounds to the multi-field packet classification problem imply that any solution will be either too slow, or will consume a large amount of storage in the worst case. Given that it is difficult to design hardware around an engine with unpredictable speed, RFC takes the approach of ensuring bounded worst-case classification time.

1. At the time of writing, SDRAMs are available at approximately \$1.0 per megabyte, and SRAMs at \$12 for a 4 Mbit device running at 125 MHz [119][129].

```
/* pktFields[i] stores the value of field i in the packet header */
for (each chunk numbered chkNum of phase 0)
  eqNums[0][chkNum] = contents of appropriate rfc table at memory location pktFields[chkNum];
for (phaseNum = 1..numPhases-1)
for (each chunk numbered chkNum in Phase phaseNum)
begin
 /* chd stores the number and description about this chunk's parents chkParents[0..numChkParents-1]*/
 chd = parent descriptor of (phaseNum, chkNum);
 indx = eqNums[phaseNum of chkParents[0]][chkNum of chkParents[0]];
 for (i=1..chd->numChkParents-1)
   begin
     indx = indx * (total #eqIDs of chd->chkParents[i]) + eqNums[phaseNum of chd->chkParents[i]][chkNum of chd->chkParents[i]];
     /*** Alternatively: indx = (indx << (#bits of equivID of chd->chkParents[i])) ^ (eqNums[phaseNum of chkParents[i]][chkNum of chkParents[i]] ***/
   endfor
 eqNums[phaseNum][chkNum] = contents of appropriate rfc table at address indx
endfor
endfor
return (eqNums[numPhases-1][0]); /* this contains the desired classID */
```

Figure 4.31 Pseudocode for the RFC lookup operation.

This has the side-effect of making it difficult to accurately predict the storage requirements of RFC as a function of the size of the classifier — the performance of RFC is determined by the structure present in the classifier. Even though pathological sets of rules do not seem to appear in practice, RFC storage requirements could scale geometrically with the number of rules in the worst case. This lack of characterization of the precise storage requirements of RFC as a function of only the number of rules in a classifier is a disadvantage to designers implementing RFC in hardware.

4.2.2 Lookups in software

Figure 4.31 shows the pseudocode to perform RFC lookups. When written in 'C,' approximately 30 lines of code are required to implement RFC. When compiled on a 333 MHz Pentium-II PC running Windows NT, we found that the worst case path for the code took $(140clk + 9t_m)$ time for three phases, and $(146clk + 11t_m)$ for four phases, where

t_m is the memory access time, and *clk* equals 3 ns.[1] With $t_m = 60ns$, this corresponds to $0.96\mu s$ and $1.1\mu s$ for three and four phases respectively. This implies that RFC can classify close to one million packets per second in the worst case for this dataset. The average lookup time was found to be approximately 50% faster than the worst case — Table 4.7 shows the average time taken per packet classification for 100,000 randomly generated packets for some classifiers in the dataset.

TABLE 4.7. Average time to classify a packet using a software implementation of RFC.

Number of rules in classifier	Average time per classification (ns)
39	587
113	582
646	668
827	611
1112	733
1733	621

The pseudocode in Figure 4.31 calculates the indices into each memory using multiplication/addition operations on *eqIDs* from previous phases. Alternatively, the indices can be computed by simple concatenation. This has the effect of increasing the memory consumed because the tables do not remain as tightly packed.[2] Given the simpler processing, we might expect the classification time to decrease at the expense of increased memory usage. Indeed the memory consumed grows approximately by a factor of two for the classifiers we have considered. Surprisingly, we saw no significant reduction in classification times. We believe that this is because the processing time is dominated by memory access time as opposed to the CPU cycle time.

1. The performance of the lookup code was analyzed using VTune [138], an Intel performance analyzer for processors of the Pentium family.
2. Not packing rfc tables in memories may in fact be desirable to accomodate newly added rules in the classifier.

4.3 Larger classifiers

To estimate how RFC might perform with future, larger classifiers, we synthesized large artificial classifiers. We used two different ways to create large classifiers (given the importance of the structure, it did not seem meaningful to generate rules randomly):

1. A large classifier can be created by concatenating classifiers for different services, but belonging to the same network, into a single classifier. This is actually desirable in scenarios where only one set of RFC tables is desired for the whole network. In such cases, the classID obtained would have to be combined with some other information (such as the classifier ID) to obtain the correct intended action. By only concatenating classifiers from the same network, we were able to create classifiers such that the biggest classifier had 3896 rules. For each classifier created, we measured the storage requirements of RFC with both three and four phases. This is shown in Figure 4.32.
2. To create even larger classifiers, we concatenated all the classifiers of a few (up to ten) different networks. The performance of RFC with four phases is plotted as the 'Basic RFC' curve in Figure 4.35. We found that RFC frequently runs into storage problems for classifiers with more than 6000 rules. Employing more phases does not help as we must combine at least two chunks in every phase, and finish with one chunk in the final phase.[1] An alternative way to process large classifiers would be to split them into two (or more) parts and construct separate RFC tables for each part. This would of course come at the expense of doubling the number of memory accesses.[2]

5 Variations

Several variations and improvements of RFC are possible. First, it is easy to see how RFC can be extended to process a larger number of fields in each packet header.

Second, we can possibly speed up RFC by taking advantage of fast lookup algorithms that find longest matching prefixes in one field. Note that in our examples, we use three

1. With six chunks in Phase 0, we could have increased the number of phases to a maximum of six. However we found no appreciable improvement by doing so.
2. For Phase 0, we need not lookup memory twice for the same chunk if we use wide memories. This would help us access the contents of both the RFC tables in one memory access.

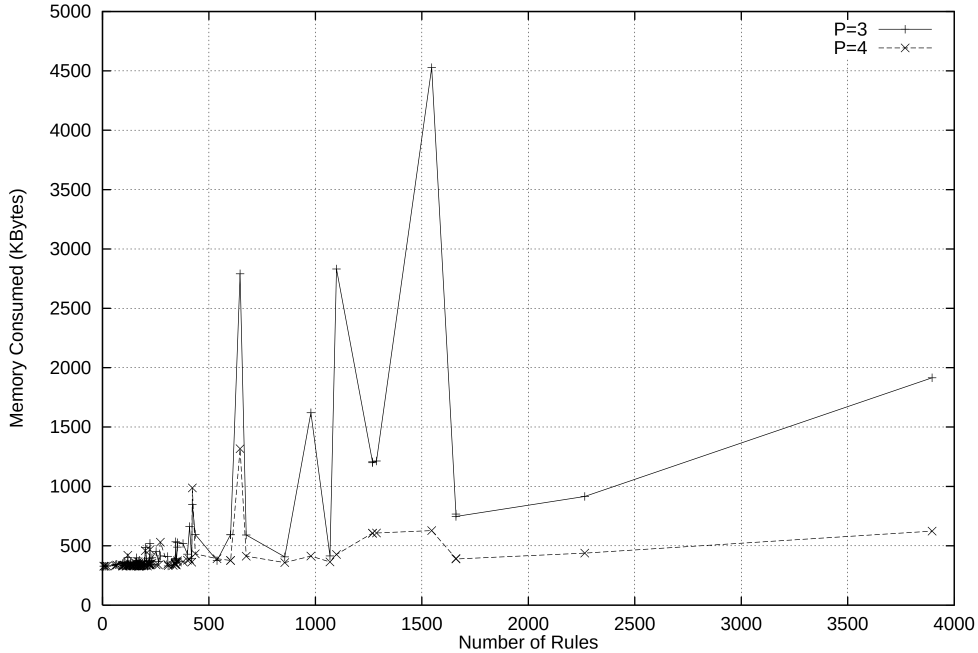


Figure 4.32 The memory consumed by RFC for three and four phases on classifiers created by merging all the classifiers of one network.

memory accesses each for the source and destination network-layer address lookups during the first two phases of RFC. This is necessary because of the large number of non-contiguous address/mask specifications. If only prefixes are allowed in the specification, one can use a more sophisticated and faster algorithm for looking up in one dimension, for instance, one of the algorithms described in Chapter 2.

Third, we can employ the technique described below to decrease the storage requirements for large classifiers.

5.1 Adjacency groups

Since the size of RFC tables depends on the number of chunk equivalence classes, we try to reduce this number by merging two or more rules of the original classifier as explained below. We find that each additional phase of RFC further increases the amount of compaction possible on the original classifier.

First we define some notation. We call two distinct rules R and S, with R appearing first in the classifier, to be *adjacent in dimension* i if all of the following three conditions are satisfied: (1) Both rules have the same action, (2) All but the i^{th} field have the exact same specification in the two rules, and (3) All rules appearing in between R and S in the classifier have either the same action or are disjoint from R (i.e., do not overlap with R). Two rules are simply said to be *adjacent* if they are adjacent in some dimension. Adjacency can also be viewed in the following way: Treat each rule with d fields as a boolean expression of d (multi-valued) variables. Each rule is a conjunction (logical-AND) of these variables. Two rules are now defined to be adjacent if they are adjacent vertices in the d-dimensional hypercube created by the symbolic representation of the d fields.

Example 4.3: For the example classifier of Table 4.8, R2 and R3 are adjacent in the dimension corresponding to the transport-layer Destination field. Similarly R5 is adjacent to R6 (in the dimension network-layer Source), but not to R4 (different actions), or to R7.

TABLE 4.8. An example classifier in four dimensions. The column headings indicate the names of the corresponding fields in the packet header. "*gt N*" in a field specification specifies a value strictly greater than *N*.

Rule	Network-layer destination (address/ mask)	Network-layer source (address/ mask)	Transport-layer destination	Transport -layer protocol	Action
R1	152.163.190.69/ 255.255.255.255	152.163.80.11/ 255.255.255.255	*	*	Deny
R2	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	udp	Permit
R3	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	range 20-21	udp	Permit
R4	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	tcp	Deny
R5	152.163.198.4/ 255.255.255.255	152.163.161.0/ 255.255.252.0	gt 1023	tcp	Permit
R6	152.163.198.4/ 255.255.255.255	152.163.0.0/ 255.255.252.0	gt 1023	tcp	Permit
R7	0.0.0.0/0.0.0.0	0.0.0.0/0.0.0.0	*	*	Permit

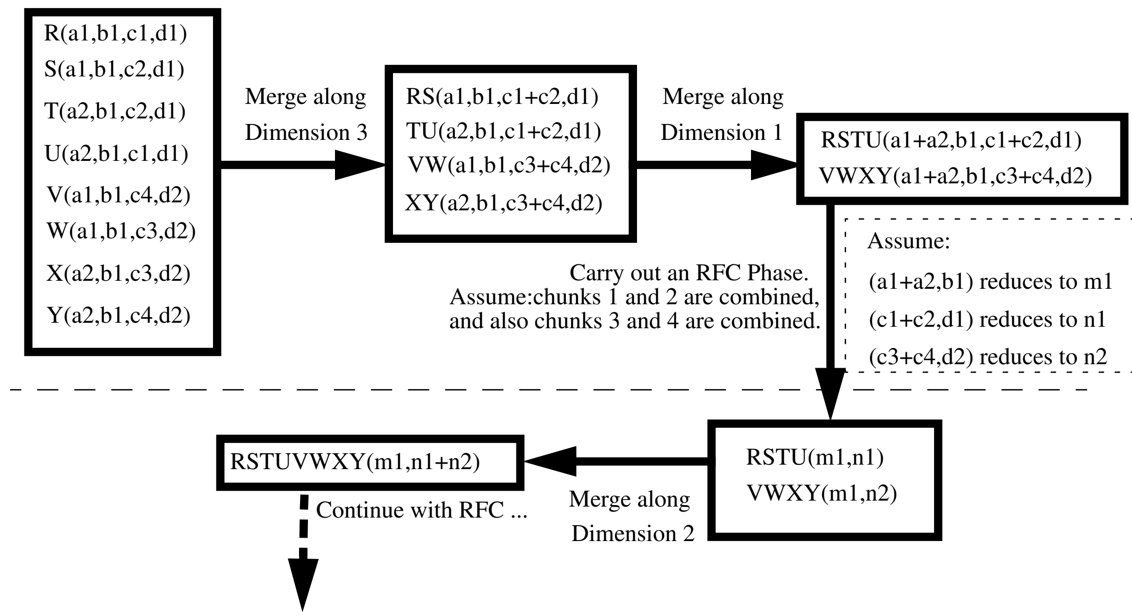


Figure 4.33 This example shows how adjacency groups are formed on a classifier. Each rule is denoted symbolically by *RuleName(value-of-field1, value-of-field2,...)*. All rules shown are assumed to have the same action. '+' denotes a logical OR.

Two rules R and S that are adjacent in dimension i are merged to form a new rule T with the same action as R (or S). T has the same specifications as that of R (or S) for all fields except that of the i^{th}, which is simply the *logical-OR* of the i^{th} field specifications in R and S. The third condition above ensures that the relative priority of the rules in between R and S will not be affected by this merging.

An *adjacency group* is defined recursively as: (1) Every rule in the original classifier is an adjacency group, and (2) Every merged rule that is created by merging two or more adjacency groups is an adjacency group.

The classifier is compacted as follows. Initially, every rule is in its own adjacency group. Next, adjacent rules are combined to create a new smaller classifier. This is implemented by iterating over all fields in turn, checking for adjacency in each dimension. After

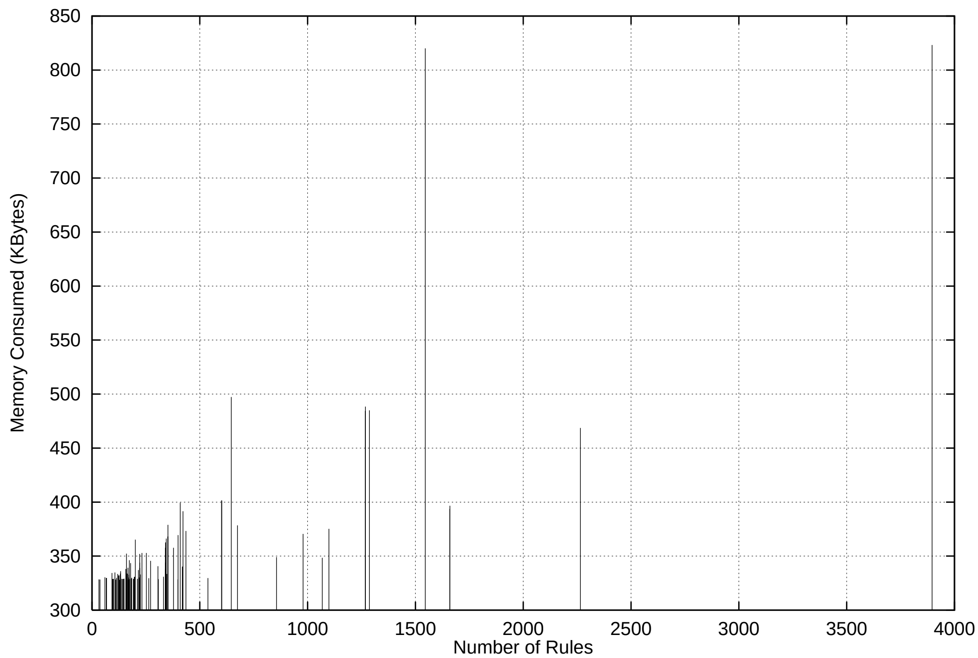


Figure 4.34 The memory consumed by RFC for three phases with the adjacency group optimization enabled on classifiers created by merging all the classifiers of one network. The memory consumed by the basic RFC scheme for the same set of classifiers is plotted in Figure 4.35.

these iterations are completed, the resulting classifier will not have any more adjacent rules. As each RFC phase collapses some dimensions, groups which were not adjacent in earlier phases may become so in later stages. In this way, the number of adjacency groups, and hence the size of the compacted classifier, keeps on decreasing with every phase. An example of this operation is shown in Figure 4.33.

Note that there is no change in the actual lookup operation: the equivalence class identifiers now represent bitmaps which keep track of adjacency groups rather than the original rules. The benefits of the adjacency group optimization are demonstrated in Figure 4.34 (using 3 RFC phases on 101 large classifiers created by concatenating all the classifiers belonging to one network) and in Figure 4.35 (using 4 RFC phases on even larger classifiers created by concatenating all the classifiers of a few different networks together) respectively. With this optimization, the storage requirements of RFC for a 15,000 rule

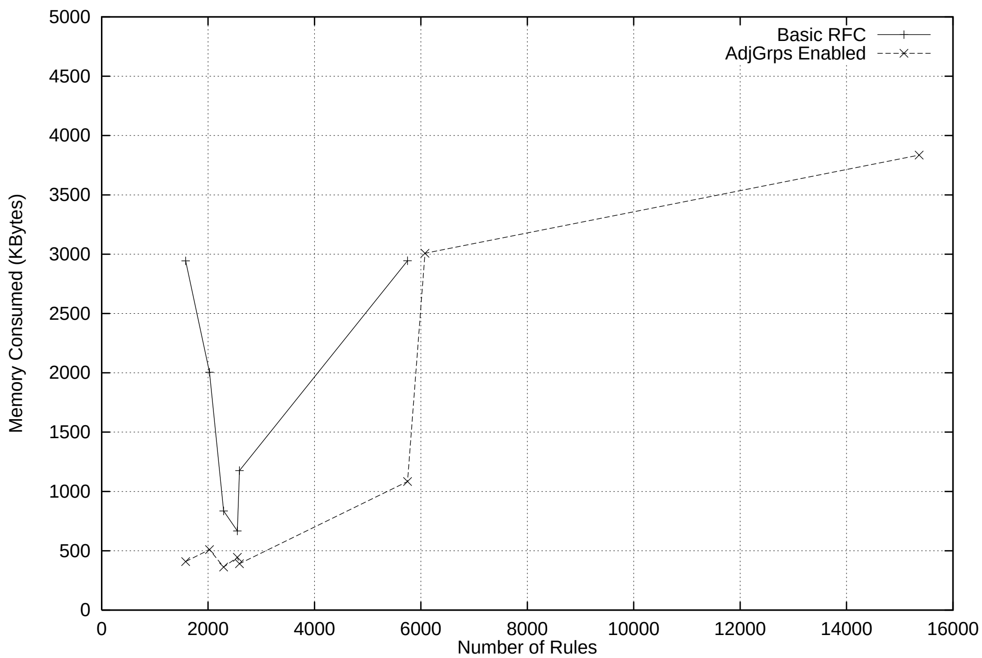


Figure 4.35 The memory consumed with four phases with the adjacency group optimization enabled on the large classifiers created by concatenating all the classifiers of a few different networks. Also shown is the memory consumed when the optimization is not enabled (i.e. the basic RFC scheme). Notice the absence of some points in the "Basic RFC" curve. For those classifiers, the basic RFC scheme takes too much memory/preprocessing time.

classifier decreases to only 3.85 MB. The intuitive reason for the reduction in storage is that several rules in the same classifier commonly share a number of specifications for many fields (an observation mentioned in Section 3.2).

However, the storage space savings come at a cost. Although the classifier will correctly identify the action for each arriving packet, it cannot tell which rule in the original classifier it matched — as the rules have been merged to form adjacency groups, the distinction between each rule has been lost. This may be undesirable in applications that wish to maintain matching statistics for each rule.

TABLE 4.9. A qualitative comparison of some multi-dimensional classification algorithms.

Scheme	Pros	Cons
Sequential evaluation	Good storage and update requirements. Suitable for multiple fields.	High classification time.
Grid-of-tries and FIS-tree	Good storage requirements and fast lookup rates for two fields. Suitable for big 2-dimensional classifiers.	Results in two dimensions do not extend as well to more than two fields. Not suitable for non-contiguous masks.
Crossproducting	Fast accesses. Suitable for multiple fields. Can be adapted to rules with non-contiguous masks.	Large memory requirements. Suitable without caching for small classifiers up to 50 rules.
Bitmap-intersection	Suitable for multiple fields. Can be adapted to rules with non-contiguous masks.	Large memory size and memory bandwidth required. Comparatively slow lookup rate. Hardware only.
Tuple space search	Suitable for multiple fields. Fast average classification and update time.	Non-deterministic and high classification time.
Recursive flow classification	Suitable for multiple fields. Supports rules with non-contiguous masks. Reasonable storage requirements for real-life classifiers. Fast classification.	High preprocessing time and memory requirements for large classifiers (i.e. having more than 6000 rules without adjacency group optimization).

6 Comparison with related work

Table 4.9 shows a qualitative comparison of RFC with previously proposed schemes for doing packet classification.

7 Conclusions and summary of contributions

It is relatively simple to perform packet classification at high speeds using excessively large amounts of storage, or at low speeds with small amounts of storage. When matching multiple fields simultaneously, theoretical bounds show that it is difficult to achieve both high classification rate and modest storage in the worst case. This chapter shows that real classifiers exhibit a considerable amount of structure and redundancy, and introduces for the first time the idea of using simple heuristic algorithms to solve the multi-dimensional packet classification problem.

The contribution of this chapter is the first proposed algorithm, RFC, that deliberately attempts to exploit this structure. RFC appears to perform well with the selection of real-life classifiers available to us. A hardware implementation of RFC can classify minimum-sized IP packets at OC192c rates with commercial memories commonly available today, while a software implementation can classify at OC48c rates. This chapter also shows that while the basic RFC scheme may consume a large amount of storage for large four-field classifiers (with more than 6000 rules), the structure and redundancy in the classifiers can be further exploited with an optimization of the basic RFC scheme. This optimization makes RFC practical for classifiers containing up to approximately 15,000 rules.