# Algorithms for Packet Classification

**Pankaj Gupta and Nick McKeown, Stanford University**

**Abstract**

The process of categorizing packets into "flows" in an Internet router is called packet classification. All packets belonging to the same flow obey a predefined rule and are processed in a similar manner by the router. For example, all packets with the same source and destination IP addresses may be defined to form a flow. Packet classification is needed for non-best-effort services, such as firewalls and quality of service; services that require the capability to distinguish and isolate traffic in different flows for suitable processing. In general, packet classification on multiple fields is a difficult problem. Hence, researchers have proposed a variety of algorithms which, broadly speaking, can be categorized as basic search algorithms, geometric algorithms, heuristic algorithms, or hardware-specific search algorithms. In this tutorial we describe algorithms that are representative of each category, and discuss which type of algorithm might be suitable for different applications.

Until recently, Internet routers provided only best-effort service, servicing packets in a first-come-first-served manner. Routers are now called on to provide different qualities of service to different applications, which means routers need new mechanisms such as admission control, resource reservation, per-flow queuing, and fair scheduling. All of these mechanisms require the router to distinguish packets belonging to different flows.

Flows are specified by *rules* applied to incoming packets. We call a collection of rules a *classifier*. Each rule specifies a flow to which a packet may belong based on some criteria applied to the packet header (shown in Fig. 1). To illustrate the variety of classifiers, consider some examples of how packet classification can be used by an Internet service provider (ISP) to provide different services. Figure 2 shows $ISP_1$ connected to three different sites: enterprise networks $E_1$ and $E_2$, and a network access point[1] (NAP), which is in turn connected to $ISP_2$ and $ISP_3$. $ISP_1$ provides a number of different services to its customers, as shown in Table 1.

| Service | Example |
|---|---|
| Packet filtering | Deny all traffic from $ISP_3$ (on interface X) destined to $E_2$. |
| Policy routing | Send all voice-over-IP traffic arriving from $E_1$ (on interface Y) and destined to $E_2$ via a separate ATM network. |
| Accounting and | Treat all video traffic to $E_1$ (via interface Y) as highest priority and perform accounting for the traffic sent this way. |
| Traffic rate limiting | Ensure that $ISP_2$ does not inject more than 10 Mb/s of e-mail traffic and 50 Mb/s of total traffic on interface X. |
| Traffic shaping | Ensure that no more than 50 Mb/s of Web traffic is injected into $ISP_2$ on interface X. |

■ Table 1. *Customer services provided by $ISP_1$.*

Table 2 shows the flows into which an incoming packet must be classified by the router at interface *X*. Note that the flows specified may or may not be mutually exclusive. For example, the first and second flow in Table 2 overlap. This is common in practice, and when no explicit priorities are specified, we follow the convention that rules closer to the top of the list take priority.

## Problem Statement

Each rule of a classifier has *d* components. $R[i]$ is the $i$th component of rule *R*, and is a regular expression on the $i$th field of the packet header. A packet *P* is said to *match* rule *R*, if $\forall i$, the $i$th field of the header of *P*, satisfies the regular expression $R[i]$. In practice, a rule component is not a general regular expression but is often limited by syntax to a simple address/mask or operator/number(s) specification. In an address/mask specification, a 0 (1) at bit position *x* in the mask denotes that the corresponding bit in the address is a don't care (significant) bit. Examples of operator/number(s) specifications are *eq 1232* and *range 34-9339*. Note that a prefix can be specified as an address/mask pair where the mask is contiguous (i.e., all bits with value 1 appear to the left of bits with value 0 in the mask). It can also be specified as a range of width equal to $2^t$ where $t = 32 - prefixlength$. Most com-

[1] *A network access point is a network site which acts as an exchange point for Internet traffic. ISPs connect to the NAP to exchange traffic with other ISPs.*

| Flow | Relevant packet fields |
|---|---|
| Email and from ISP2 | Source link-layer address, source transport port number |
| From $ISP_2$ | Source link-layer address |
| From $ISP_3$ and going to $E_2$ | Source link-layer address, destination network-layer address |

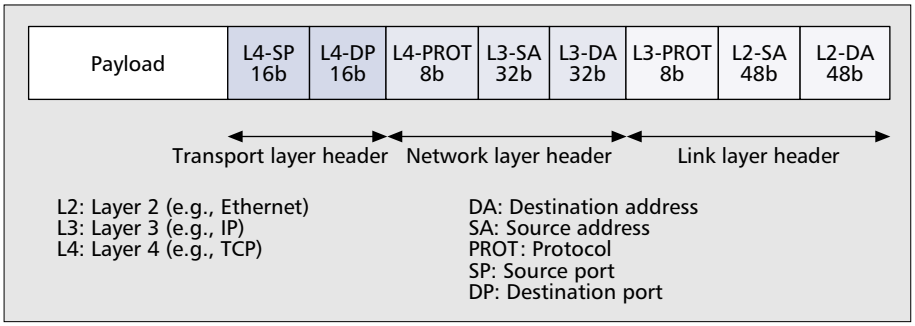
■ Table 2. *Flows into which an incoming packet must be classified by the router at interface X.*


0890-8044/01/$10.00 © 2001 IEEE

monly occurring specifications can be represented by ranges.

An example real-life classifier in four dimensions is shown in Table 3. By convention, the first rule $R1$ is of highest priority, and rule $R7$ is of lowest priority. Some example classification results are shown in Table 4.

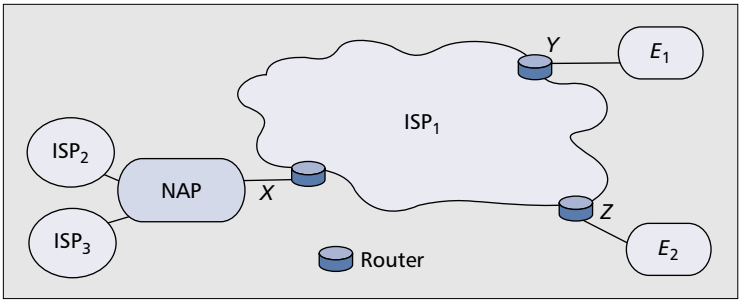
Longest prefix matching for routing lookups is a special case of one-dimensional packet classification. All packets destined to the set of addresses described by a common prefix may be considered to be part of the same flow. The address of the next hop to which the packet should be forwarded is the associated action. The length of the prefix defines the priority of the rule.




■ Figure 1. *This figure shows some of the header fields (and their widths) that might be used for classifying the packet. Although not shown in this figure, higher-layer (e.g., application-level) headers may also be used.*

## Performance Metrics for Classification Algorithms

- *Search speed* — Faster links require faster classification. For example, links running at 10 Gb/s can bring 31.25 million packets/s (assuming minimum-sized 40-byte TCP/IP packets).
- *Low storage requirements* — Small storage requirements enable the use of fast memory technologies like static random access memory (SRAM). SRAM can be used as an on-chip cache by a software algorithm and as on-chip SRAM for a hardware algorithm.
- *Ability to handle large real-life classifiers*.
- *Fast updates* — As the classifier changes, the data structure needs to be updated. We can categorize data structures into those which can add or delete entries incrementally, and those which need to be reconstructed from scratch each time the classifier changes. When the data structure is reconstructed from scratch, we call it *preprocessing*. The update rate differs among different applications: a very low update rate may be sufficient in firewalls where entries are added manually or infrequently, whereas a router with per-flow queues may require very frequent updates.
- *Scalability in the number of header fields used for classification*.




■ Figure 2. *An example network of an ISP ($ISP_1$) connected to two enterprise networks ($E_1$ and $E_2$) and to two other ISP networks across a network access point (NAP).*

- *Flexibility in specification* — A classification algorithm should support general rules, including prefixes, operators (range, less than, greater than, equal to, etc.), and wildcards. In some applications, noncontiguous masks may be required.

## Classification Algorithms

### Background

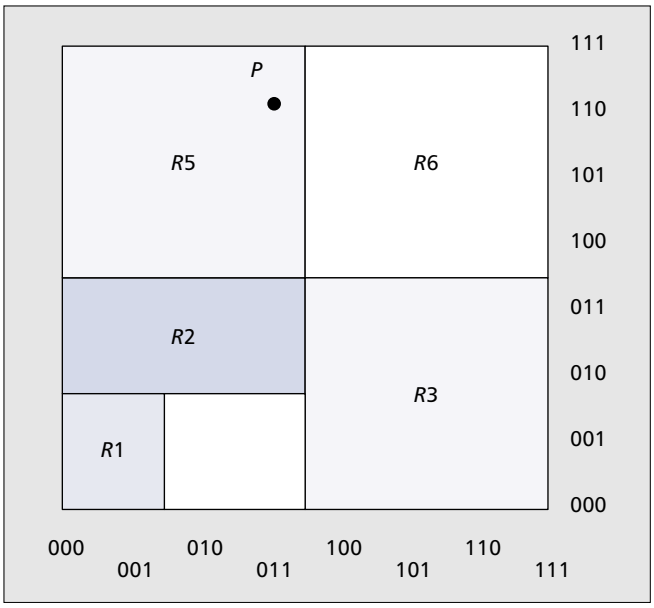
For the next few sections, we will use the example classifier in Table 5 repeatedly. The classifier has six rules in two fields labeled $F1$ and $F2$; each specification is a prefix of maximum length 3 bits. We will refer to the classifier as $C = \{R_j\}$ and each rule $R_j$ as a 2-tuple: $<R_{j1}, R_{j2}>$.

| Rule | Network-layer destination (address/mask) | Network-layer source (address/mask) | Transport-layer destination | Transport-layer protocol | Action |
|---|---|---|---|---|---|
| *R*1 | 152.163.190.69/255.255.255.255 | 152.163.80.11/255.255.255.255 | * | * | Deny |
| *R*2 | 152.168.3.0/255.255.255.0 | 152.163.200.157/255.255.255.255 | eq www | udp | Deny |
| *R*5 | 152.163.198.4/255.255.255.255 | 152.163.160.0/255.255.252.0 | gt 1023 | tcp | Permit |
| *R*6 | 0.0.0.0/0.0.0.0 | 0.0.0.0/0.0.0.0 | * | * | Permit |

■ Table 3. *A real-life classifier in four dimensions.*

| Packet header | Network-layer destination | Network-layer source | Transport-layer destination | Transport-layer protocol | Best matching rule, action |
|---|---|---|---|---|---|
| *P*1 | 152.163.190.69 | 152.163.80.11 | www | tcp | *R*1, deny |
| *P*2 | 152.168.3.21 | 152.163.200.157 | www | udp | *R*2, deny |
| *P*3 | 152.168.198.4 | 152.163.160.10 | 1024 | tcp | *R*5, permit |

■ Table 4. *Example classification results.*




■ Figure 3. *A geometric representation of the classifier in Table 5. A packet represents a point, for instance* P*(011,110), in two-dimensional space. Note that* R*4 is hidden by* R*1 and* R*2.*

*Bounds from Computational Geometry* — There is a simple geometric interpretation of packet classification. While a prefix represents a contiguous interval on the number line, a two-dimensional rule represents a rectangle in two-dimensional Euclidean space, and a rule in $d$ dimensions represents a $d$-dimensional hyper-rectangle. A classifier is therefore a collection of prioritized hyper-rectangles, and a packet header represents a point in $d$ dimensions. For example, Fig. 3 shows the classifier in Table 5 geometrically in which high-priority rules overlay lower-priority rules. Classifying a packet is equivalent to finding the highest-priority rectangle that contains the point representing the packet. For example, point P(011,110) in Fig. 3 would be classified by rule $R_5$.

There are several standard geometry problems (e.g., ray shooting, point location, and rectangle enclosure) that resemble packet classification. Point location involves finding the enclosing region of a point, given a set of nonoverlapping regions. The best bounds for point location in $N$ rectangular regions and $d > 3$ dimensions are $O(\log N)$ time with $O(N^d)$ space;[2] or $O(\log N)^{d-1}$ time with $O(N)$ space [1, 2]. In packet classification, hyper-rectangles can overlap, making classification at least as hard as point location. Hence, a solution is either impracticably large (with 100 rules and 4 fields, $N^d$ space is about 100 MBytes) or too slow ($(\log N)^{d-1}$ is about 350 memory accesses).

We can conclude that:

- Multifield classification is considerably more complex than one-dimensional longest prefix matching.
- Complexity may require that practical solutions use heuristics.

*Range Lookups* — Packet classification is made yet more complex by the need to match on ranges as well as prefixes. A range lookup for a dimension of $W$ width bits can be defined as:

Definition 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.*

To assess the increased complexity of ranges, we can convert each range to a set of prefixes (a prefix of length corresponds to a range where the least significant bits of $l$ are all 0 and those of $u$ are all 1) and use a longest prefix matching algorithm. Table 6 shows some examples of range-to-prefix conversions for $W = 4$.

A $W$-bit range can be represented by at most $2W - 2$ prefixes (see the last row of Table 6 as an example), which means a prefix matching algorithm can find ranges with times as much storage. Feldman and Muthukrishnan [3] show a reduction of ranges to prefix lookup with a twofold storage increase that can be used in some specific multidimensional classification schemes.

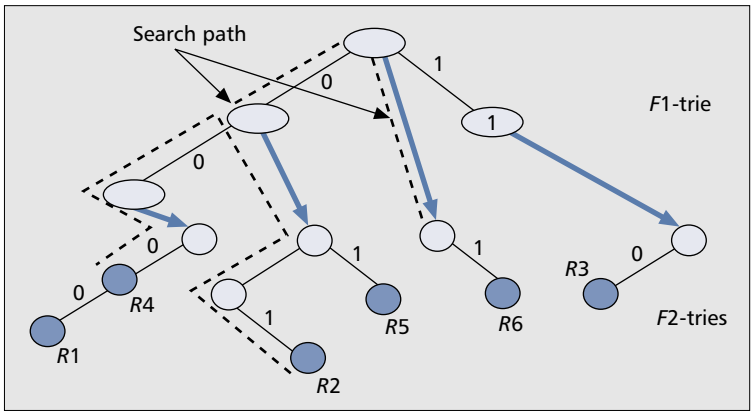
## Taxonomy of Classification Algorithms

The classification algorithms we will describe here can be categorized into the four classes shown in Table 7.

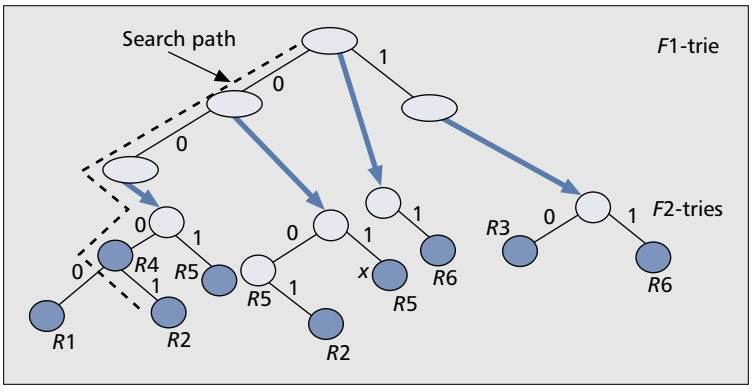
We now proceed to describe representative algorithms from each class.

## Basic Data Structures

*Linear Search* — The simplest data structure is a linked list of rules stored in order of decreasing priority. A packet is compared with each rule sequentially until a rule is found that




■ Figure 4. *A hierarchical trie data structure. The gray pointers are the next-trie pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is shown.*




■ Figure 5. *A set-pruning trie data structure. The gray pointers are the "next-trie" pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is shown.*

[2] *The time bound for* d ″ *3 is* O*(loglog*N*) [1] but has large constant factors.*

matches all relevant fields. 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.

*Hierarchical Tries* — A $d$-dimensional hierarchical radix trie is a simple extension of the one dimensional radix trie data structure, and is constructed recursively as follows. If $d$ is greater than 1, we first construct a 1-dimensional trie, called the $F1$-trie, on the set of prefixes $\{R_{j1}\}$, belonging to dimension $F1$ of all rules in the classifier, $C = \{R_j\}$. For each prefix, $p$, in the $F1$-trie, we recursively construct a $(d - 1)$ -dimensional hierarchical trie, $T_p$, on those rules which specify exactly $p$ in dimension $F1$, that is, the set of rules $\{R_j{:}R_{j1} = p\}$. Prefix $p$ is linked to the trie $T_p$ using a *next-trie* pointer. The storage complexity of the data structure for an $N$-rule classifier is $O(NdW)$. The data structure for the classifier in Table 5 is shown in Fig. 4. Hierarchical tries are sometimes called *multilevel tries*, *backtracking-search tries*, or *trie-of-tries*.

| Rule | *F1* | *F2* |
|---|---|---|
| $R_1$ | 00* | 00* |
| $R_2$ | 0* | 01* |
| $R_3$ | 1* | 0* |
| $R_4$ | 00* | 0* |
| $R_5$ | 0* | 1* |
| $R_6$ | * | 1* |

Table 5. *An example classifier.*

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

Table 6. *Examples of range-to-prefix conversions for* W = *4.*

Classification of an incoming packet $(v_1, v_2, \ldots, v_d)$ proceeds as follows. The query algorithm first traverses the $F1$-trie based on the bits in $v_1$. At each $F1$-trie node encountered, the algorithm follows the next-trie pointer (if present) and traverses the $(d - 1)$-dimensional trie. The query time complexity for $d$ dimensions is therefore $O(W^d)$. Incremental updates can be carried out similarly in $O(d^2W)$ time since each component of the updated rule is stored in exactly one location at maximum depth $O(dW)$.

*Set-Pruning Tries* — A set-pruning trie data structure [4] is similar, but with reduced query time obtained by replicating rules to eliminate recursive traversals. The data structure for the classifier in Table 5 is shown in Fig. 5. The query algorithm for an incoming packet $(v_1, v_2, \ldots, v_d)$ need only traverse the $F1$-trie to find the longest matching prefix of v1, follow its next-trie pointer (if present), traverse the $F2$-trie to find the longest matching prefix of v1, and so on for all dimensions. The rules are replicated to ensure that every matching rule will be encountered in the path. The query time is reduced to $O(dW)$ at the expense of increased storage of $O(N^d dW)$ since a rule may need to be replicated $O(N^d)$ times. Update complexity is $O(N^d)$; hence, this data structure works only for relatively static classifiers.

## Geometric Algorithms

*Grid-of-tries* — The grid-of-tries data structure, proposed by Srinivasan *et al.* [5] for 2D classification, reduces storages space by allocating a rule to only one trie node as in a hierarchical trie, and yet achieves $O(W)$ query time by precomputing and storing a *switch pointer* in some trie nodes. A switch pointer is labeled 0 or 1 and guides the search process. The conditions that 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 (Fig. 6):

- $T_x$ and $T_w$ are distinct tries built on the prefix components of dimension $F2$. $T_x$ and $T_w$ are pointed to by two distinct nodes, say $r$ and $s$ respectively of the same trie, $T$, built on prefix components of dimension $F1$.
- 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 $T_x$.
- Node $w$ does not have a child pointer labeled $b$.

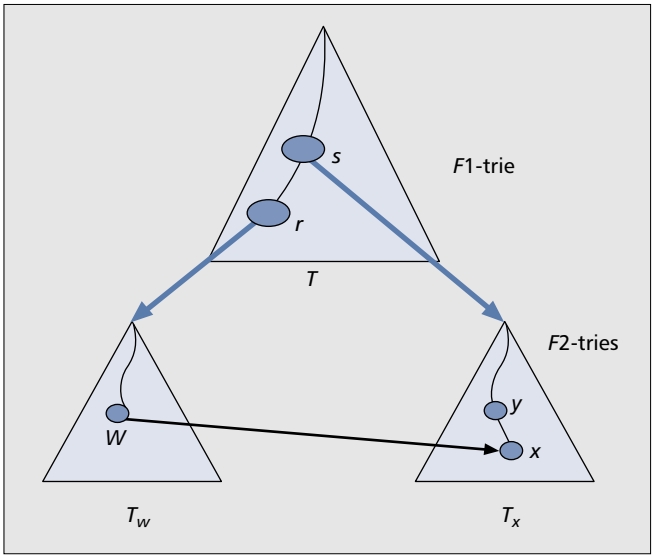



Figure 6. *The conditions under which a switch pointer exists from node* w *to* x.

| Category | Algorithms |
|---|---|
| Basic data structures | Linear search, caching, hierarchical tries, set-pruning tries |
| Geometry-based | Grid-of-tries, AQT, FIS |
| Heuristic | RFC, hierarchical cuttings, tuple-space search |
| Hardware only | Ternary CAM, bitmap-intersection |

Table 7. *Four classes of classification algorithms.*

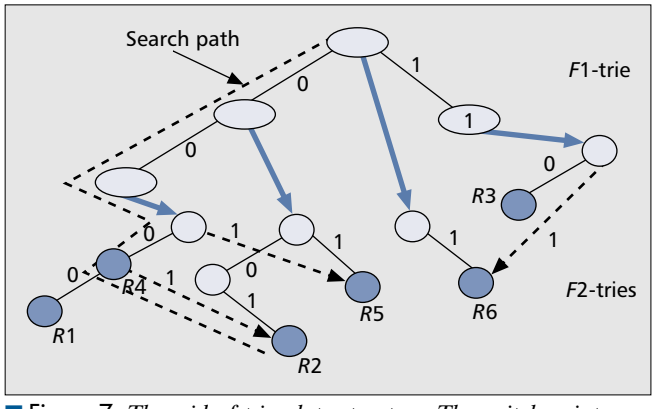
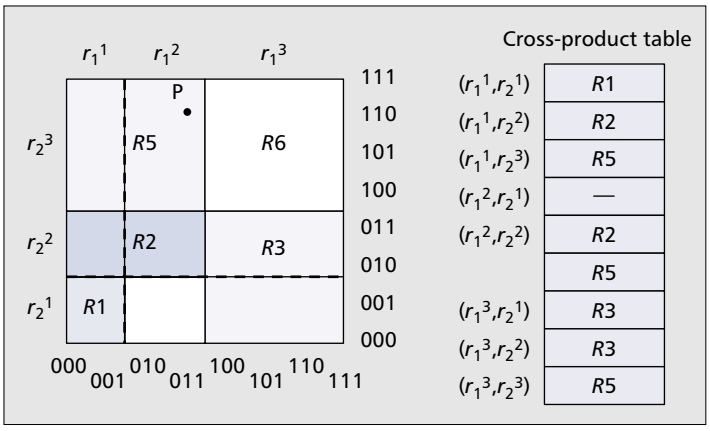



Figure 7. *The grid-of-tries data structure. The switch pointers are shown dashed. The path traversed by the query algorithm on an incoming packet (000, 010) is shown.*




■ Figure 8. *The table produced by the cross-producting algorithm and its geometric representation.*

• 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 $U2(r, root\ (T_w), w)$ in a hierarchical trie, it need only traverse the path $(s, r, root\ (T_w), w, x)$ on a grid-of-tries. This is because paths $U1$ and $U2$ are identical (by condition 2 above) till $U1$ terminates at node $w$ because it has no child branch (by condition 3). The switch pointer eliminates the need for backtracking in a hierarchical trie without the storage of a set-pruning trie. Each bit of the packet header is examined at most once, so the time complexity reduces to $O(W)$, while storage complexity $O(NW)$ is the same as a 2D hierarchical trie. However, switch pointers makes incremental updates difficult, so the authors [5] recommend rebuilding the data structure (in time $O(NW)$) for each update. An example of the grid-of-tries data structure is shown in Fig. 7.

Reference [5] reports 2 Mbytes of storage for a 20,000 2D classifier with destination and source IP prefixes. The stride of the destination (source) prefix trie was 8 (5) bits, respectively, leading to a maximum of nine memory accesses.

Grid-of-tries works well for 2D classification, and can be used for the last two dimensions of a multidimensional hierarchical trie, decreasing the classification time complexity by a factor of $W$ to $O(NW^{d-1})$. As with hierarchical and set-pruning tries, grid-of-tries handles range specifications by splitting into prefixes.

*Cross-Producting* — Cross-producting [5] is suitable for an arbitrary number of dimensions. Packets are classified by composing the results of separate 1D range lookups for each dimension, as explained below.

Constructing the data structure involves computing a 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 cross-product table $C_T$ of size

$$\prod_{k=1}^{d} s_k$$

is constructed, and the best matching rule for each entry

$$\left(r_1^{i_1}, r_2^{i_2}, \ldots r_d^{i_d}\right), 1 \le i_k \le s_k, 1 \le k \le d$$

is precomputed and stored. Classifying a packet ($v_1$, $v_2$, …, $v_d$) involves a range lookup in each dimension $k$ to identify the range $r_k^{ik}$ containing point $v_k$. The tuple

$$\left\langle r_1^{i_1}, r_2^{i_2}, \ldots r_d^{i_d} \right\rangle$$

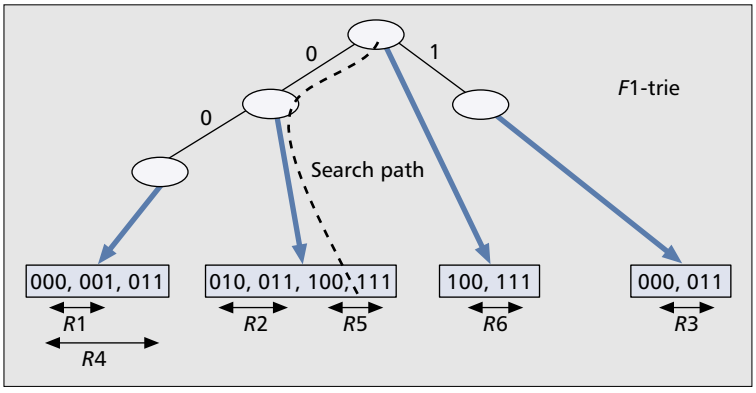
is then found in the cross-product table $C_T$ which contains the precomputed best matching rule. Figure 8 shows an example.

Given that $N$ prefixes leads to at most $2N - 2$ ranges, $s_k \le 2N$ and $C_T$ is of size $O(N^d)$. The lookup time is $O(dt_{RL})$ where $t_{RL}$ is the time complexity of finding a range in one dimension. Because of its high worst case storage complexity, cross-producting is suitable for very small classifiers. Reference [5] proposes using an on-demand cross- producting scheme together with caching for classifiers bigger than 50 rules in five dimensions. Updates require reconstruction of the cross-product table, so cross-producting is suitable for relatively static classifiers.

*A 2D Classification Scheme [6]* — Lakshman and Stiliadis [6] propose a 2D classification algorithm where one dimension, say $F1$, is restricted to have prefix specifications while the second dimension, $F2$, is allowed to have arbitrary range specifications. The data structure first builds a trie on the prefixes of dimension $F1$, and then associates a set $G_w$ of nonoverlapping ranges to each trie node, $w$, that represents prefix $p$. These ranges are created by (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 $w$. An example is shown in Fig. 9.

Searching for point $P(v_1, v_2)$ involves a range lookup in data structure $G_w$ for each trie node, $w$, encountered. The search in $G_w$ returns the range containing $v_2$, and hence the best matching rule. The highest priority rule is selected from the rules $\{R_w\}$ for all trie nodes encountered during the traversal.

The storage complexity is $O(NW)$ because each rule is stored only once in the data structure. Queries take $O(W\log N)$ time because an $O(\log N)$ range lookup is performed for every node encountered in the $F1$-trie. This can be reduced to $O(W + \log N)$ using *fractional cascading* [7], but that makes incremental updates impractical.




■ Figure 9. *The data structure for the example classifier of Table 5. The search path for example packet* P*(011, 110) resulting in* R*5 is also shown.*

*Area-Based Quadtree* — The area-based quadtree (AQT) was proposed by Buddhikot *et al.* [8] for 2D classification. AQT allows incremental updates whose complexity can be traded off with query time by a tunable parameter. Each node of a quadtree [7] represents a 2D space that is decomposed into four equal-sized quadrants, each of which is represented by a child node. The initial 2D space is recursively decomposed into four equal-sized quadrants till each quadrant has at most one rule in it (Fig. 10 shows an example of the decomposition). Rules are allocated to each node as follows. A rule is said to cross a quadrant if it completely spans at least one dimension of the quadrant. For instance, rule $R6$ spans the

quadrant represented by the root node in Fig. 10, while *R*5 does not. If we divide the 2D space into four quadrants, rule *R*5 crosses the northwest quadrant while rule *R*3 crosses the southwest quadrant. We call the set of rules crossing the quadrant represented by a node in dimension $k$ the $k$-crossing filter set ($k$- CFS) of that node.

Two instances of the same data structure are associated with each quadtree node — each stores the rules in $k$-CFS ($k$ = 1,2). Since rules in crossing filter sets span at least one dimension, only the range specified in the other dimension need be stored. Queries proceed two bits at a time by transposing one bit from each dimension, with two 1D lookups being performed (one for each dimension on $k$-CFS) at each node. Figure 11 shows an example.

Reference [8] proposes an efficient update algorithm that, for N two-dimensional rules, has $O(NW)$ space complexity, $O(\alpha W)$ search time and $O(\alpha^{\alpha}\sqrt{N})$ update time, where $\alpha$ is a tunable integer parameter.

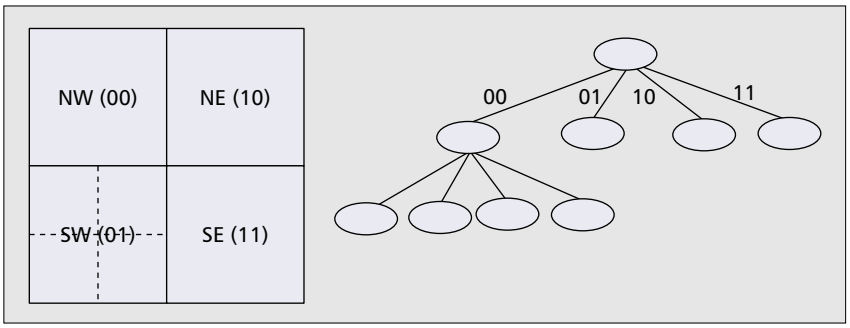



Figure 10. *A quadtree constructed by decomposition of two-dimensional space. Each decomposition results in four quadrants.*

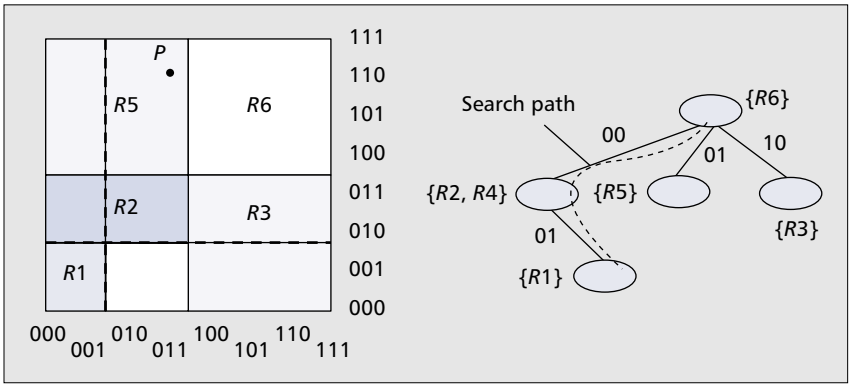



Figure 11. *An AQT data structure. The path traversed by the query algorithm for an incoming packet* P*(001, 010), yields* R*1 as the best matching rule.*

*Fat Inverted Segment Tree (FIS-Tree)* — Feldman and Muthukrishnan [3] propose the fat inverted segment tree (FIS-tree) for 2D classification as a modification of a segment tree. A segment tree [7] stores a set $S$ of possibly overlapping line segments to answer queries such as finding the highest priority line segment containing a given point. A segment tree is a balanced binary search tree containing the endpoints of the line segments in $S$. Each node, $w$, represents a range $G_w$, leaves represent the original line segments in $S$, and parent nodes represent the union of the ranges represented by their children. A line segment is allocated to a node $w$ if it contains $G_w$ but not $G_{parent(w)}$. The highest-priority line segment allocated to a node is precomputed and stored at the node. A query traverses the segment tree from the root, calculating the highest priority of all the precomputed segments encountered. Figure 12 shows an example segment tree.

An FIS-tree is a segment tree with two modifications:

- The segment tree is compressed (made "fat") by increasing the degree to more than two in order to decrease its depth, and occupies a given number of levels $l$.
- Up-pointers from child to parent nodes are used.

The data structure for two dimensions consists of an FIS-tree on dimension $F1$ and a range lookup data associated with each node. 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 query for point $P(v_1, v_2)$ first solves the range lookup problem on 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 up-pointers from this leaf node toward the root node, carrying out 1D range lookups at each node. The highest-priority rule containing the given point is calculated at the end of the traversal.

Queries on an $l$-level FIS-tree have complexity $O((l + 1)t_{RL})$ with storage complexity $O(ln^{l+1})$, where $t_{RL}$ is the time for a 1D range lookup. Storage space can be traded off with search time by varying $l$. Modifications to the FIS-tree are necessary to support incremental updates; even then, it is easier to support inserts than deletes [3]. The static FIS-tree can be extended to multiple dimensions by building hierarchical

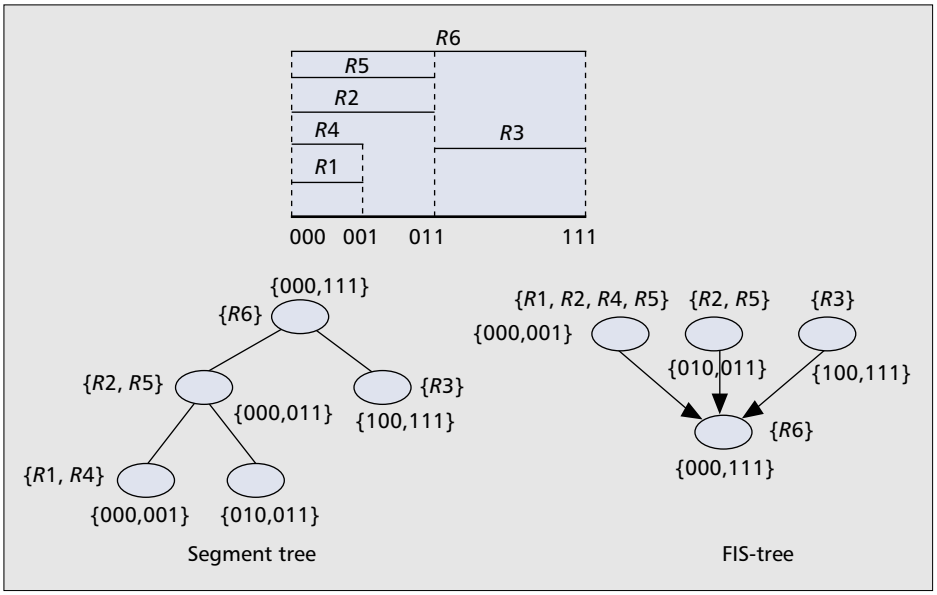



Figure 12. *The segment tree and the 2-level FIS-tree for the classifier of Table 5.*

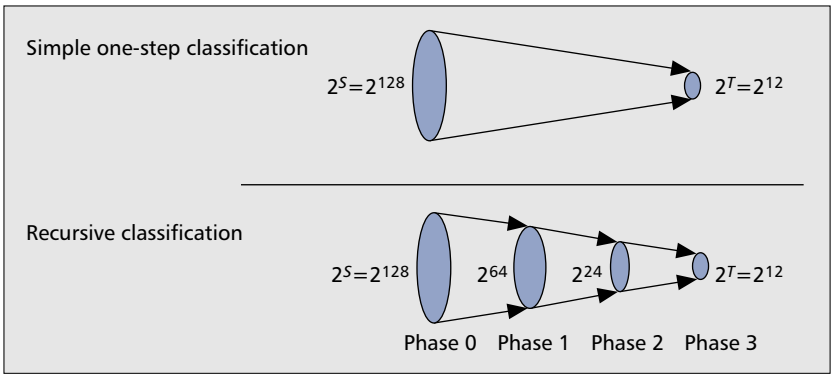



Figure 13. *Showing the basic idea of RFC. The reduction is carried out in multiple phases, with a reduction in phase I being carried out recursively on the image of the phase I – 1. The example shows the mapping of bits to bits in three phases.*

FIS-trees, but the bounds are similar to other methods studied earlier [3].

Measurements on real-life 2D classifiers are reported in [3] using the static FIS-tree data structure. Queries took 15 or less memory operations with a two-level tree, 4–60K rules, and 5 Mbytes of storage. Large classifiers with one million 2D rules required three levels, 18 memory accesses per query, and 100 Mbytes of storage.

## Heuristics

As we saw earlier, the packet classification problem is expensive to solve in the worst case: theoretical bounds state that solutions to multifield classification require either storage that is geometric or a number of memory accesses that is polylogarithmic in the number of classification rules. We can expect that classifiers in real networks have considerable structure and redundancy that might be exploited by a heuristic. That is the motivation behind the algorithms described in this section.

*Recursive Flow Classification* — RFC [9] is a heuristic for packet classification on multiple fields. Classifying a packet involves mapping bits in the packet header to a $T$ bit action identifier, where $T = \log N$, $T \ll S$. A simple but impractical method could precompute the action for each of the $2^S$ different packet headers, yielding the action in one step. RFC attempts to perform the same mapping over several phases, as shown in Fig. 13; at each stage the algorithm maps one set of values to a smaller set. In each phase a set of memories return a value shorter (i.e., expressed in fewer bits) than the index of the memory access. The algorithm, illustrated in Fig. 14, operates as follows:

- In the first phase, $d$ fields of the packet header are split up into multiple chunks that are used to index into multiple memories in parallel. The contents of each memory are chosen so that the result of the lookup is narrower than the index.
- In subsequent phases, memories are indexed using the results from earlier phases.
- In the final phase, the memory yields the action.

The algorithm requires construction of the contents of each memory, detailed in [9].

Reference [9] reports that with real-life 4D classifiers of up to 1700 rules, RFC appears practical for 10 Gb/s line rates in hardware and 2.5 Gb/s rates in software. However, the storage space and preprocessing time grow rapidly for classifiers larger than about 6000 rules. An optimization described in [9] reduces the storage requirement of a 15,000 four-field classifier to below 4 Mbytes.

*Hierarchical Intelligent Cuttings* — HiCuts [10] partitions the multidimensional search space guided by heuristics that exploit the structure of the classifier. Each query leads to a leaf node in the HiCuts tree, which stores a small number of rules that can be searched sequentially to find the best match. The characteristics of the decision tree (its depth, degree of each node, and the local search decision to be made at each node) are chosen while preprocessing the classifier based on its characteristics (see [10] for the heuristics used).

Each node, v, of the tree represents a portion of the geometric search space. The root node represents the complete $d$-dimensional space, which is partitioned into smaller geometric subspaces, represented by its child nodes, by cutting across one of the $d$ dimensions. Each subspace is recursively partitioned until no subspace has more than $B$ rules, where $B$ is a tunable parameter of the preprocessing algorithm. An example is shown in Fig. 15 for two dimensions with $B = 2$.

Parameters of the HiCuts algorithm can be tuned to trade off query time against storage requirements. On 40 real-life 4D classifiers with up to 1700 rules, HiCuts requires less than 1 Mbyte of storage with a worst case query time of 20 memory accesses, and supports fast updates.

*Tuple Space Search* — The basic tuple space search algorithm [11] decomposes 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 specified in the $i$th dimension of the rule (the scheme

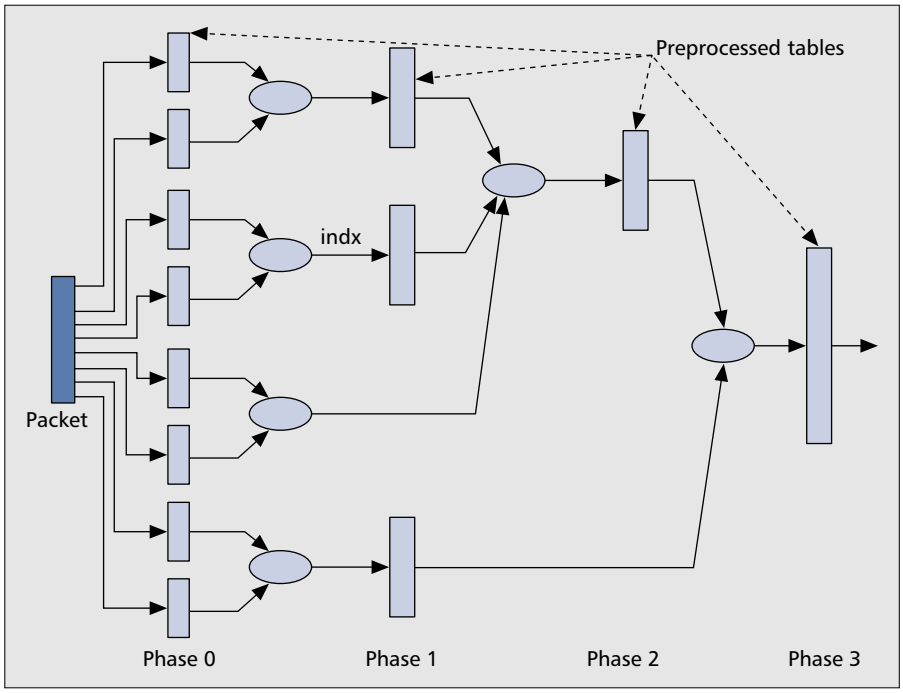



Figure 14. *Packet flow in RFC.*

supports only prefix specifications). Hence, the set of rules mapped to the same tuple are of a fixed and known length, and can be stored in a hash table. Queries perform exact match operations on each of the hash tables corresponding to all possible tuples in the classifier. An example is shown in Fig. 16.

Query time is $M$ hashed memory accesses, where $M$ is the number of tuples in the classifier. Storage complexity is $O(N)$ since each rule is stored in exactly one hash table. Incremental updates are supported and require just one hashed memory access to the hashed 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 nondeterministic. 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 because each range could be split into $O(W)$ prefixes in the manner explained earlier.

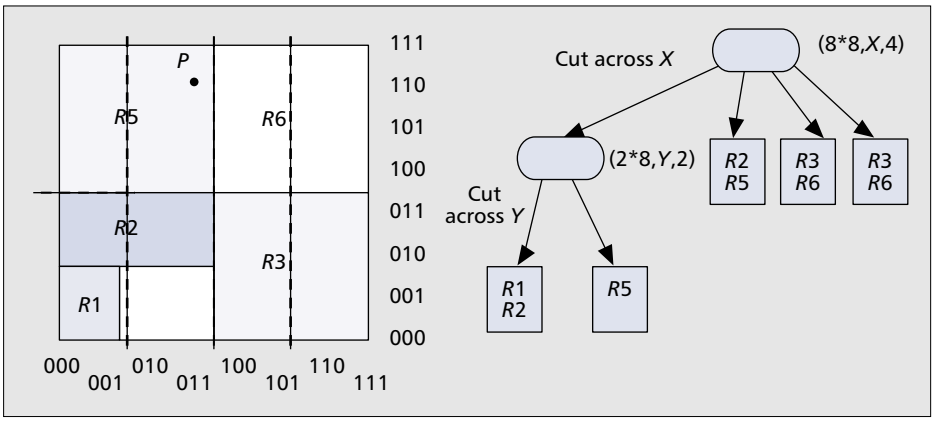



Figure 15. *A possible HiCuts tree for the example classifier in Table 5. Each ellipse in the tree denotes an internal node with a tuple (size of 2D space represented, dimension to cut across, number of children). Each square is a leaf node that contains the actual classifier rules.*

| 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 16. *The tuples and associated hash tables in the tuple space search scheme for the example classifier of Table 5.*

## Hardware-Based Algorithms

*Ternary CAMs* — A TCAM stores each $W$-bit field as a (*val*, *mask*) pair; where *val* and *mask* are each $W$-bit numbers. For example, if $W = 5$, a prefix 10* will be stored as the pair (10000, 11000). An element matches a given input key by checking if those bits of *val* for which the *mask* bit is 1 match those in the key.

A TCAM is used as shown in Fig. 17. The TCAM memory array stores rules in decreasing order of priorities, and compares an input key against every element in the array in parallel. The $N$-bit bit vector, *matched*, indicates which rules match, so the $N$-bit priority encoder indicates the address of the highest-priority match. The address is used to index into a RAM to find the action associated with this prefix.

TCAMs are increasingly being deployed because of their simplicity and speed (the promise of single clock-cycle classification). Several companies produce 2 Mb TCAMs capable of single and multifield classification in as little as 10 ns. Both faster and denser TCAMs can be expected in the near future. There are, however, some disadvantages to TCAMs:

- A TCAM is less dense than a RAM, storing fewer bits in the same chip area. One bit in an SRAM typically requires 4–6 transistors, while one bit in a TCAM requires 11–15 transistors [12]. A 2 Mb TCAM running at 100 MHz costs about $70 today, while 8 Mb of SRAM running at 200 MHz costs about $30. Furthermore, range specifications need to be split into multiple masks, reducing the number of entries by up to $(2W_{-2})^d$ in the worst case. If only two 16-bit dimensions specify ranges, this is a multiplicative factor of 900. Newer TCAMs, based on DRAM technology, have been proposed and promise higher densities. One unresolved issue with DRAM-based CAMs is the detection of soft errors caused by alpha particles.
- TCAMs dissipate more power than RAM solutions because an address is compared against every TCAM element in parallel. At the time of writing, a 2 Mb TCAM chip running at 50 MHz

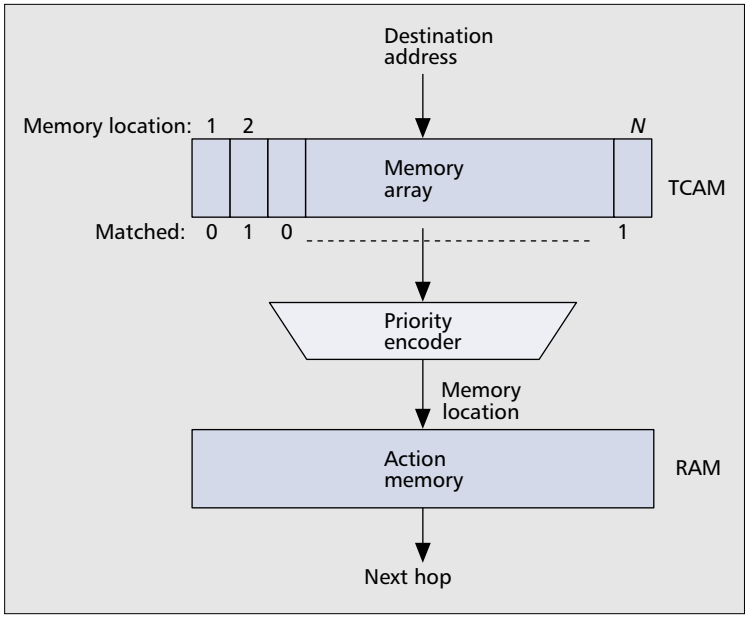
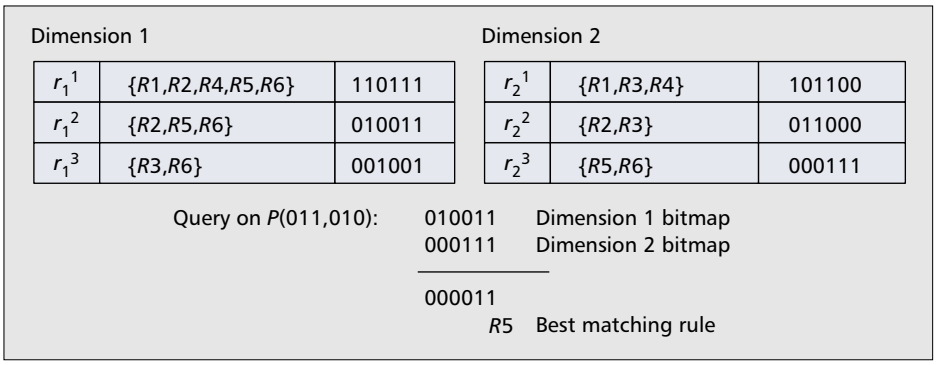



Figure 17. *The lookup operation using a ternary CAM.*




■ Figure 18. *Bitmap tables used in the bitmap-intersection classification scheme. See Fig. 8 for a description of the ranges. Also shown is a classification query on an example packet P(011, 110).*

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

■ Table 8. *A summary of classification schemes.*

dissipates about 7 W [13, 14]. In comparison, an 8 Mb SRAM running at 200 MHz dissipates approximately 2 W [15].

TCAMs are appealing for relatively small classifiers, but will probably remain unsuitable in the near future for:

- Large classifiers (256K–1M rules) used for microflow recognition at the edge of the network
- Large classifiers (128–256K rules) used at edge routers that manage thousands of subscribers (with a few rules per subscriber)
- Extremely high-speed (greater than 200 Mpackets/s) classification
- Price-sensitive applications

*Bitmap-Intersection* — The bitmap-intersection classification scheme proposed in [6] is based on the observation that the set of rules, $S$, that match a packet 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 cross-producting precomputes $S$ and stores the best matching rule in $S$, this scheme computes $S$ and the best matching rule during each classification operation.

In order to compute intersection of sets in hardware, each set is encoded as an $N$-bit bitmap with each bit corresponding to a rule. The set of matching rules is the set of rules whose corresponding bits are 1 in the bitmap. A query is similar to cross-producting: First, a range lookup is performed in each of the $d$ dimensions. Each lookup returns a bitmap representing the matching rules (precomputed for each range) in that dimension. The $d$ sets are intersected (a simple bitwise AND operation) to give the set of matching rules, from which the best matching rule is found (Fig. 18).

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

Reference [6] reports that the scheme can support up to 512 rules with a 33 MHz field-programmable gate array and five 1 Mb SRAMs, classifying 1 Mpacket/s. The scheme works well for a small number of rules in multiple dimensions, but suffers from a quadratic increase in storage space and linear increase in classification time with the size of the classifier. A variation is described in [6] that decreases storage at the expense of increased query time.

## References


[1] M.H. Overmars and A.F. van der Stappen, "Range Searching and Point Location Among Fat Objects," *J. Algorithms*, vol. 21, no. 3, Nov. 1996, pp. 629–56.
[2] F. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.
[3] A. Feldman and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. INFOCOM*, vol. 3, Mar. 2000, pp. 1193–1202.
[4] P. Tsuchiya, "A Search Algorithm for Table Entries with Non-contiguous Wildcarding," unpublished report, Bellcore.
[5] V. Srinivasan *et al.*, "Fast and Scalable Layer four Switching," *Proc. ACM Sigcomm*, Sept. 1998, pp. 203–14.
[6] T. V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," *Proc. ACM Sigcomm*, pp. 191–202, Sept. 1998.
[7] M. de Berg, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2nd rev. ed. 2000.
[8] M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space Decomposition Techniques for Fast Layer-4 Switching," *Proc. Conf. Protocols for High Speed Networks*, Aug. 1999, pp. 25–41.
[9] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. Sigcomm, Comp. Commun. Rev.*, vol. 29, no. 4, Sept. 1999, pp. 147–60.
[10] P. Gupta and N. McKeown, " Classification Using Hierarchical Intelligent Cuttings," *Proc. Hot Interconnects VII*, Aug. 1999, Stanford,; also available in *IEEE Micro*, vol. 20, no. 1, Jan./Feb.2000, pp. 34–41.
[11] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification using Tuple Space Search," *Proc. ACM Sigcomm*, Sept. 1999, pp. 135–46.
[12] F. Shafai *et al.*, "Fully Parallel 30-Mhz, 2.5 Mb CAM," *IEEE J. Solid-State Circuits*, vol. 33, no. 11, Nov. 1998.
[13] Netlogic Microsystems: http://www.netlogicmicro.com. CIDR products: http://www.netlogicmicro.com/products/cidr/cidr.html
[14] Sibercore Technologies: http://www.sibercore.com
[15] ZBT-Datasheet docs from IDT: http://www.idt.com/products/pages/ZBT_DS_p.html


## Biographies


PANKAJ GUPTA (pankaj@stanford.edu) got his Ph.D. in computer science from Stanford University in 2000. His research interests lie in the architecture and design of high-speed switches and routers, and in traffic engineering for integrated optical and data networks. He has previously worked for Cisco Systems, and is the cofounder of Sahasra Networks, Inc.

NICK MCKEOWN (nickm@stanford.edu) is assistant professor of electrical engineering and computer science at Stanford University, where he works on the theory, architecture, and implementation of high-speed Internet routers and switches. He completed his Ph.D. at Berkeley in 1995. He has worked for Hewlett-Packard Labs and Cisco Systems, and co-founded Abrizio Inc. He is as an editor of *IEEE Transactions on Networking*, the Robert Noyce Faculty Fellow at Stanford, and a research fellow of the Alfred P. Sloan Foundation.