Classifying Packets with Hierarchical Intelligent Cuttings

Increasing demands on Internet router performance and functionality create a need for algorithms that classify packets quickly with minimal storage requirements and allow frequent updates. Unlike previous algorithms, the algorithm proposed here meets this need well by using heuristics that exploit structure present in classifiers.

Pankaj Gupta
Nick McKeown
Stanford University

Internet routers classify packets to implement a number of advanced Internet services: routing, rate limiting, access control in firewalls, virtual bandwidth allocation, policy-based routing, service differentiation, load balancing, traffic shaping, and traffic billing. These services require the router to classify incoming packets into different flows and then to perform appropriate actions depending on this classification. A classifier—a set of filters or rules—specifies the flows, or classes. For instance, each rule in a firewall might specify a set of source and destination addresses and associate a corresponding deny or permit action with it. Or the rules might be based on several fields of the packet header, including layers 2, 3, 4, and perhaps 5, which contain addressing and protocol information.

The simplest, best-known form of packet classification occurs in routing IP (Internet protocol) datagrams, in which each rule specifies a destination prefix. The associated action is the IP address of the next router to which the packet must be forwarded. The classification process requires determining the longest prefix that matches the packet's destination address.

In this article, we focus on the practical implementation of classification with real-life classifiers. Our approach, which we call HiCuts (hierarchical intelligent cuttings), attempts to partition the search space in each dimension, guided by simple heuristics that exploit the classifier's structure. We discover this structure by preprocessing the classifier. We can tune the algorithm's parameters to trade off query time against storage requirements. In classifying packets based on four header fields, HiCuts performs quickly and requires relatively little storage compared with previously described algorithms (see "Related work" sidebar).

Generic packet classification

Generic packet classification requires the router to classify a packet on the basis of multiple fields in its header. Each rule of the classifier specifies a class that a packet may belong to according to criteria on F fields of the pack-

0272-1732/00/$10.00 © 2000 IEEE

Related work

The simplest classification algorithm is a linear search of each rule of a classifier. Of course, for large classifiers this approach requires a long query time, but it is very efficient in terms of storage requirements. The data structure is simple and can be easily updated as rules change.

A ternary content-addressable memory (CAM) is a hardware device that functions as a fully associative memory. A ternary CAM cell can store three values: 0, 1, or *X*. The *X* represents a don't-care bit and operates as a per-cell mask enabling the ternary CAM to match rules containing wildcards. In its operation, a ternary CAM seems almost ideally suited to packet classification; one can present a whole packet header to the device and determine which entry (or entries) it matches. Thus far, however, the complexity of CAMs has permitted only small, inflexible, and relatively slow implementations that consume a lot of power. Although this technology is not quite ready for packet classification today, improvements in semiconductors might make dense, fast ternary CAMs feasible in the future. In the meantime, the need continues for efficient algorithmic solutions operating on specialized data structures.

Srinivasan et al.[1] proposed such an algorithm for two dimensions, the Grid of Tries. In this scheme, a trie data structure is extended to two fields. The authors showed that on a classifier with 20,000 rules in two dimensions, the scheme requires about nine memory accesses per query in the worst case and about 2 Mbytes for storage. Finding that the scheme does not easily extend to more than two fields, the authors proposed a generalized scheme that they call cross-producting. Cross-producting works well with classifiers smaller than about 50 rules but requires caching for larger classifiers.

Lakshman and Stiliadis[2] proposed an alternative hardware-optimized scheme using bit-level parallelism. The scheme achieves fast query times and small storage requirements for small classifiers. But the storage requirement scales quadratically and the memory bandwidth scales linearly with the size of the classifier, making the scheme impractical for large classifiers. The authors proposed variations that optimize the scheme under certain conditions—for example, decreasing the storage requirement at the expense of longer query time or optimizing for lookups in two fields. The scheme does not appear to work well for large, multidimensional classifiers.

More recently, we proposed a packet classification algorithm for multiple fields called Recursive Flow Classification.[3] The algorithm works well for real-life classifiers, but the storage requirements are still high (up to 3 Mbytes). Also, we have not yet discovered a mechanism for incrementally updating the data structure.

Srinivasan, Suri, and Varghese[4] recently proposed Tuple-Space Search, another algorithm for packet classification on multiple fields. It partitions a classifier's rules into different tuple categories based on the number of specified bits in the rules (a bit is specified in a rule if it is not a don't-care bit). The scheme then uses hashing among rules in the same category. This algorithm achieves fast queries and fast updates when rules change. Hashing, however, leads to lookups or updates of nondeterministic duration.

One can find the worst-case bounds on the complexity of queries and storage requirements by casting packet classification as a problem in computational geometry. In particular, one can reduce the point location problem—finding the enclosing region of a query point, given a set of nonoverlapping regions—to packet classification. For *n* nonoverlapping regions in *F* dimensions (fields), two well-known techniques trade off query time against storage requirements. Optimizing for point location query time leads to a complexity of $O(\log n)$ for query time but a complexity of $O(n^F)$ for storage. Optimizing for storage leads to $O(n)$ for storage but $O(\log^{F-1} n)$ for queries.[5]

Clearly, both extremes are impractical; with just 1,000 rules and three fields, n^F storage is about 1 Gbyte, and $\log^{F-1} n$ time is about 100 memory accesses. Moreover, the constants are big, and hence these techniques have little practical significance. Also, one cannot straightforwardly generalize the algorithms and data structures (not described here) to the case of overlapping regions.

We draw two conclusions from previous work:

- The theoretical bounds tell us that arriving at a practical worst-case solution is not possible. Fortunately, we don't have to. Real-life classifiers have some inherent structure that apparently we can exploit using simple heuristics.
- No single algorithm will perform well for all cases. A simple and memory-efficient linear search or Lakshman and Stiliadis's[2] hardware solution might be sufficient when the number of filters is small. A hybrid scheme might combine the advantages of several approaches.

References

1. V. Srinivasan et al., "Fast and Scalable Layer-4 Switching," *Proc. ACM SIGCOMM 1998*, ACM Press, New York, 1998, pp. 203-214.
2. T.V. Lakshman and D. Stiliadis, "High-Speed Policy-Based Packet Forwarding Using Efficient Multidimensional Range Matching," *Proc. ACM SIGCOMM 1998*, pp. 191-202.
3. P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. ACM SIGCOMM 1999*, pp. 147-160.
4. V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," *Proc. ACM SIGCOMM 1999*, pp. 135-146.
5. M.H. Overmars and A.F. van der Stappen, "Range Searching and Point Location Among Fat Objects," *J. of Algorithms*, Vol. 21, No. 3, 1996, pp. 629-656.

et header and associates an identifier, classID, with each class. (For example, each rule in a flow classifier is a flow specification, in which each flow is in a separate class.) This identifier uniquely specifies the action associated with the rule. Each rule has *F* components. The *i*th

Table 1. A classifier in two dimensions. A rule specifies a range of values in each dimension. The range may consist of only one value, so that the rule can represent a rectangle, a line, or a point in the plane.

Rule	*X* range	*Y* range
R1	0-31	0-255
R2	0-255	128-131
R3	64-71	128-255
R4	67-67	0-127
R5	64-71	0-15
R6	128-191	4-131
R7	192-192	0-255

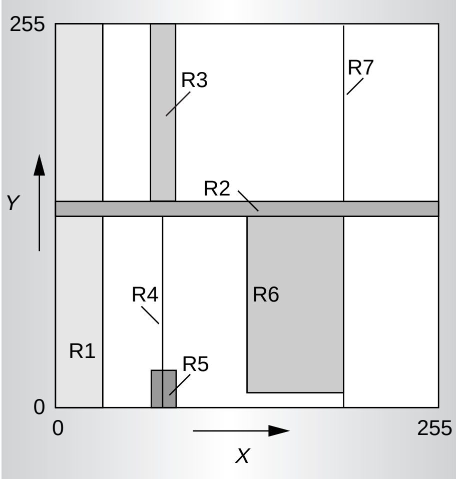

Figure 1. Geometrical representation of the seven rules in Table 1.

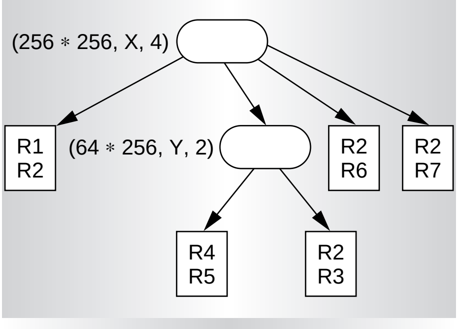

Figure 2. A possible tree for the classifier in Figure 1 (*binth* = 2). Each ellipse denotes an internal node *v* with a triplet (*B*(*v*), *dim*(*C*(*v*)), *np*(*C*(*v*))). Each square is a leaf node that contains the rules.

component of rule *R*, referred to as *R*[*i*], is a regular expression on the *i*th field of the packet header. (In practice, the regular expression is limited by syntax to a simple address/mask or operator/number specification.) A packet *P* matches a particular rule *R* if for every *i*, the *i*th field of the header of *P* satisfies the regular expression *R*[*i*].

The classes specified by the rules may overlap; that is, one packet can match several rules. Without loss of generality, we assume throughout this article that when two rules overlap, the order in which they appear in the classifier determines their relative priority. In other words, a packet that matches multiple rules belongs to the class identified by the classID of the rule among them that appears first in the classifier.

HiCuts packet classification

The HiCuts algorithm builds a decision-tree data structure by carefully preprocessing the classifier. Each time a packet arrives, the classification algorithm traverses the decision tree to find a leaf node, which stores a small number of rules. A linear search of these rules yields the desired matching. During the building of the decision tree, we choose its shape and depth, as well as the local decisions to be made at each tree node.

With each internal node *v* of a *k*-dimensional classifier, we associate

- A box *B*(*v*), which is a *k*-tuple of ranges or intervals: ($[l_1:r_1]$, $[l_2:r_2]$, …, $[l_k:r_k]$).
- A cut *C*(*v*), defined by a dimension *d*, and *np*(*C*), the number of times *B*(*v*) is cut (partitioned) in *d* (that is, the number of cuts in the interval $[l_d:r_d]$). The cut thus divides *B*(*v*) into smaller boxes, which we then associate with the children of *v*.
- A set of rules *R*(*v*). The tree's root has all the rules associated with it. If *u* is a child of *v*, then *R*(*u*) is defined as the subset of *R*(*v*) that collides with *B*(*u*). That is, every rule in *R*(*v*) that spans, cuts, or is contained in *B*(*u*) is also a member of *R*(*u*). We call *R*(*u*) the colliding rule set of *u*.

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

We perform cutting on each level and recursively on the children of the nodes at that level until the number of rules in the box associated with each node falls below a threshold we call *binth*. A node with fewer than *binth* rules is not partitioned further and becomes a leaf of the tree. To illustrate this process, Table 1 shows an example classifier. Figure 1 illustrates the same classifier geometrically. Figure 2 shows the decision tree obtained by the recursive partitioning process (in this example, *binth* equals 2).

For any classifier, there are many ways to construct a decision tree, so we guide the preprocessing with intelligent heuristics based on structure present in the classifier. When performing cuts on node v, the preprocessing algorithm uses the following four heuristics:

- A heuristic that chooses a suitable $np(C)$, the number of interval cuts to make. A large value of $np(C)$ will decrease the tree's depth, which will accelerate query time, at the expense of increased storage. To balance this trade-off, we follow a heuristic that is guided and tuned by a predetermined space measure function, *spmf*(). We let *NumRules*(u) equal the cardinality of $R(u)$. For cut C, we define a space measure,

$$sm\left(C\left(v\right)\right) = \sum_i NumRules\left(child_i\right) + np\left(C\left(v\right)\right).$$

  We make as many cuttings as *spmf*()allows at a certain node, depending on the number of rules at that node. This requires performing a simple binary search on the number of cuttings until $sm(C(v))$ gets close enough to *spmf*(*NumRules*(v)).
- A heuristic that chooses the dimension to cut along at each internal node. For example, as Figure 1 shows, cutting along the Y-axis would be less beneficial than cutting along the X-axis. We can use various methods to choose the dimension: 1) Minimizing $max_j(NumRules(child_j))$ in an attempt to decrease the worst-case depth of the tree. 2) Treating $NumRules(child_j)/sm(C)$ as a probability distribution with $np(C)$ elements, and maximizing the entropy of the distribution. Intuitively, this method attempts to pick a dimension that leads to the most uniform distribution of rules across nodes. 3) Minimizing $sm(C)$ over all dimensions. 4) Choosing the dimension that contains the largest number of distinct rule components (range specifications in either dimension). For example, in Figure 1, R3 and R5 share the same rule component (range) in the X dimension.
- A heuristic that maximizes the reuse of child nodes. We have observed that in real classifiers many child nodes have identical colliding rule sets. Hence, we can use a single child node for each distinct colliding rule set and have identical child nodes point to that node. Figure 3 illustrates this heuristic.
- A heuristic that eliminates redundancies in the tree. As a result of the partitioning of a node, rules may become redundant in some of the child nodes. In our example, if R6 had a higher priority than R2, then R6 in the third child of the root node would make R2 redundant. Detecting and eliminating these redundant rules can decrease the data structure storage requirements at the expense of increased preprocessing time. In our experiments, we invoked this heuristic only when the number of rules at a node fell below a threshold.

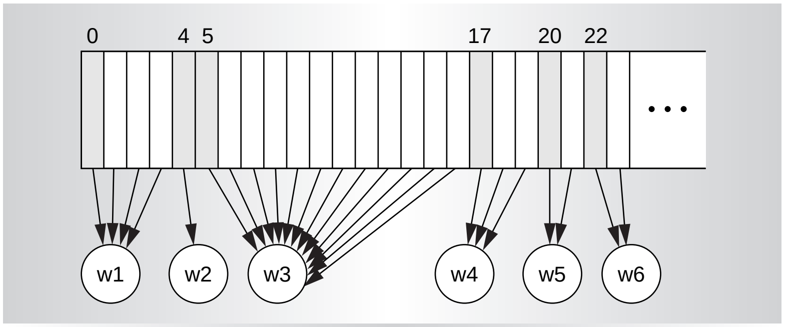

Figure 3. An example of the heuristic maximizing the reuse of child nodes. The shaded regions correspond to children with distinct colliding rule sets.

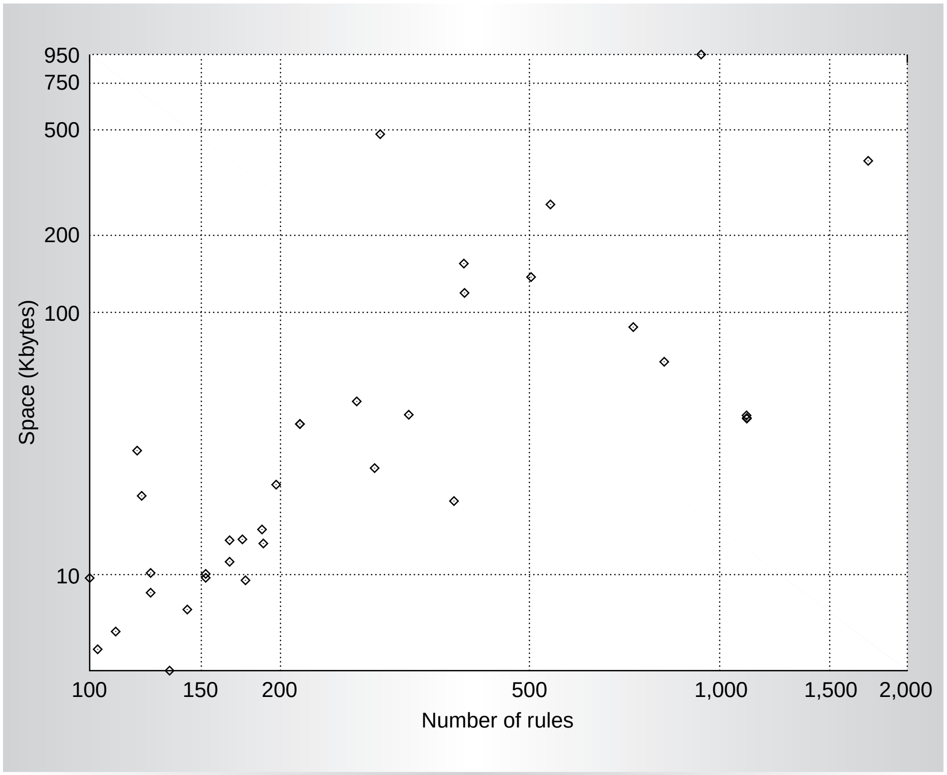

Figure 4. Storage requirements of four-dimensional classifiers for *binth* = 8 and *spfac* = 4.

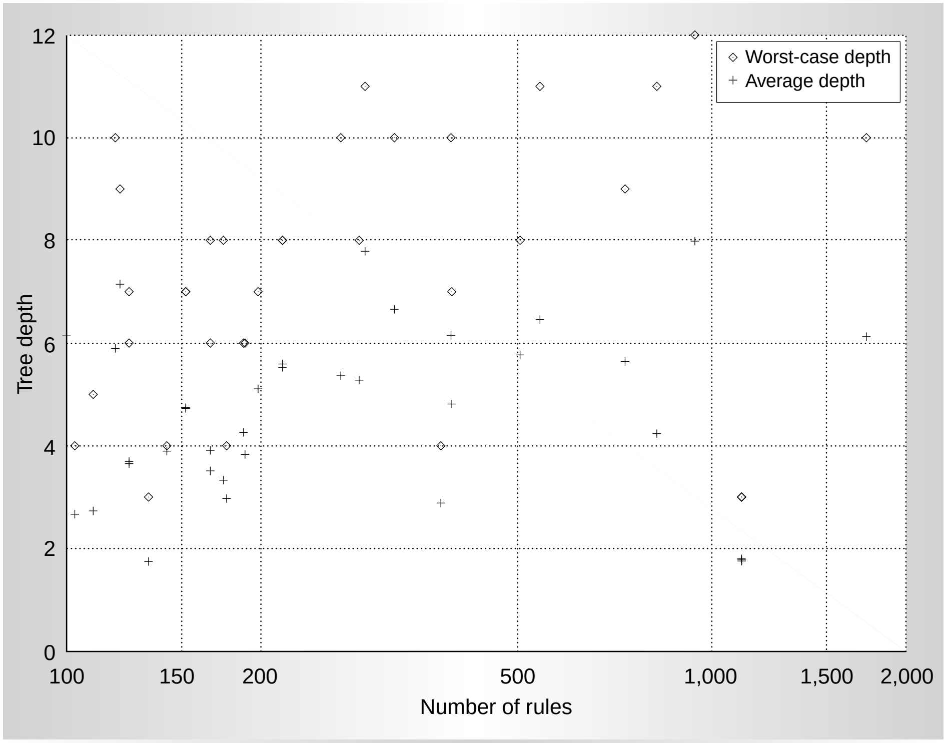

Figure 5. Average and worst-case tree depth for *binth* = 8 and *spfac* = 4.

Implementation results

To test how well the HiCuts algorithm works with real and synthesized classifiers, we built a simple simulator. For each classifier, we built a search tree using the heuristics described earlier. We tuned the preprocessing algorithm with two parameters: *binth* and *spfac*. The latter occurs in the function *spmf*(), defined as *spmf*(*N*) = *spfac* ∗ *N*.

Two dimensions

We created two-dimensional classifiers by choosing values (prefixes) in both dimensions at random from publicly available routing tables.[1] We also added wildcards at random to each dimension. Our results showed that for a *binth* of 4, a classifier with 20,000 rules consumes about 1.3 Mbytes of memory with a tree depth of 4 in the worst case and 2.3 on average. We do not present the complete set of results here because they were very similar to the results for higher-dimension classifiers described next.

Higher dimensions

For more than two dimensions, we obtained about 40 classifiers containing between 100 and 1,733 rules from ISP (Internet service provider) and enterprise networks. These classifiers, used as access control lists in firewalls, had fields in four dimensions: source IP address, destination IP address, layer-4 protocol, and layer-4 destination port. Another work provides further details about the characteristics of these classifiers.[2]

Figure 4 shows the total storage requirements for the classifiers (for the tree data structure and the classifier itself) when *binth* equals 8 and *spfac* equals 4. As the figure shows, the maximum space any classifier consumes is about 1 Mbyte, with the second highest less than 500 Kbytes. These small storage requirements mean that the data structure would easily fit in the L2 cache of most CPUs today.

Figure 5 shows the maximum and average tree depth for the classifiers with a *binth* of 8 and *spfac* of 4. (To calculate the average, we assume that each leaf is accessed in proportion to the number of rules in its colliding rule set.) The worst-case tree depth was 12, with an average value close to eight. That is, in the worst case, the classifier required a total of 12 memory accesses, followed by a linear search on eight rules to complete the classification. This makes a total of 20 memory accesses in the worst case.

An important consideration is the preprocessing time needed to build the decision tree. Figure 6 shows that the highest preprocessing time was 50.5 seconds, and the next highest was approximately 20 seconds. All but four classifiers have a preprocessing time of less than eight seconds.

The preprocessing time is clearly quite high, mainly because of the number and complexity of the heuristics. We expect this preprocessing time to be acceptable in most applications, as long as the time taken to incrementally update the tree remains small. In practice, update time depends on the rule being inserted or deleted. We have found (Figure 7) that it takes

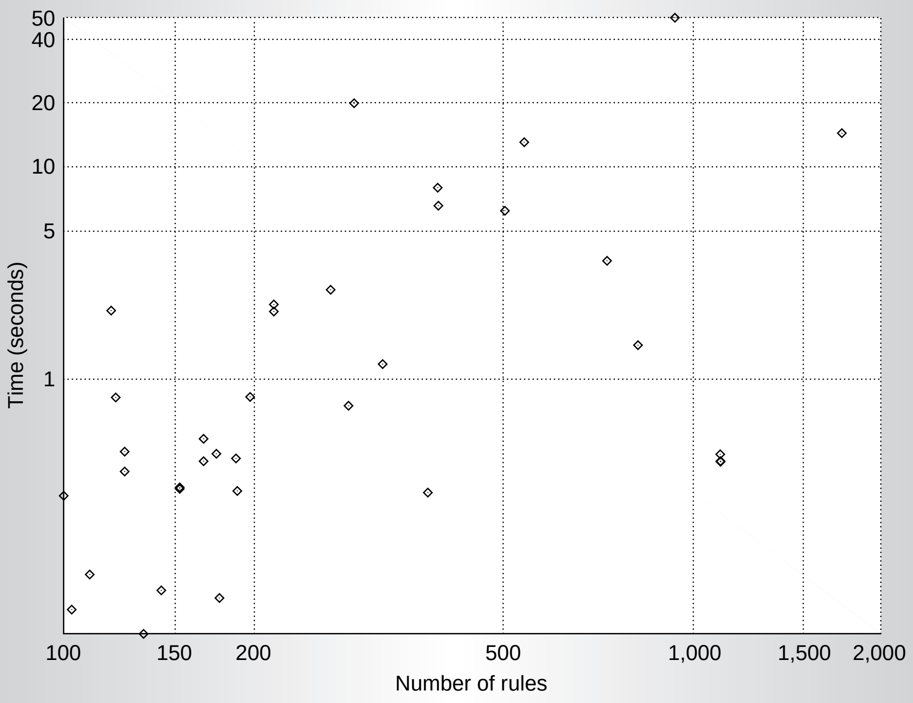

Figure 6. Time required for preprocessing the classifier to build the decision tree. For these measurements, we used the *time*() Linux system call in user-level C code on a 333-MHz Pentium-II PC with 96 Mbytes of memory and 512 Kbytes of L2 cache.

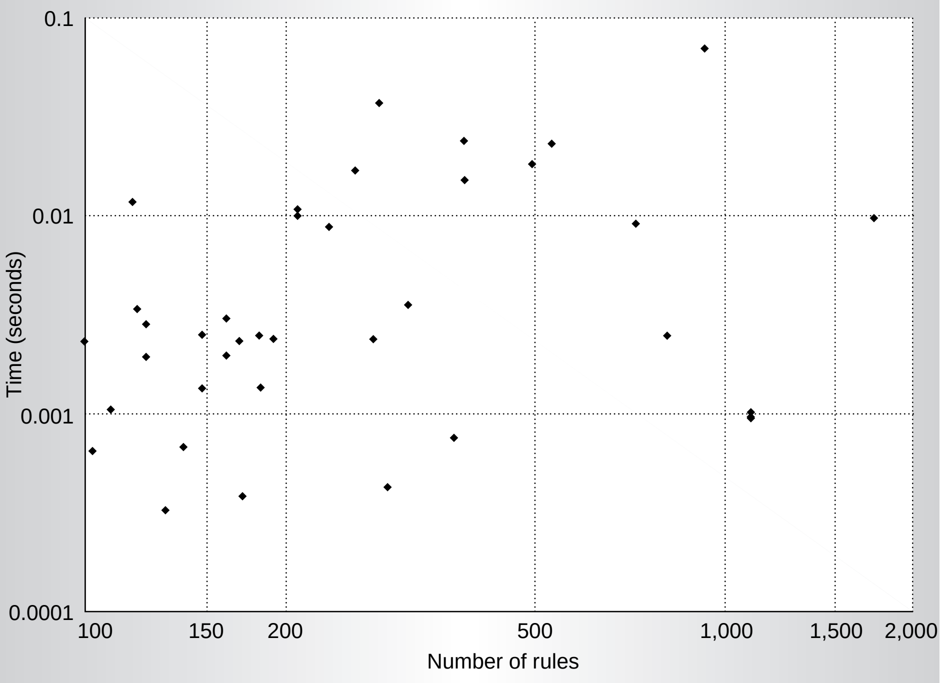

Figure 7. Average update time over 10,000 insertions and deletions of randomly chosen rules for a classifier. We made these measurements in the manner described for Figure 6.

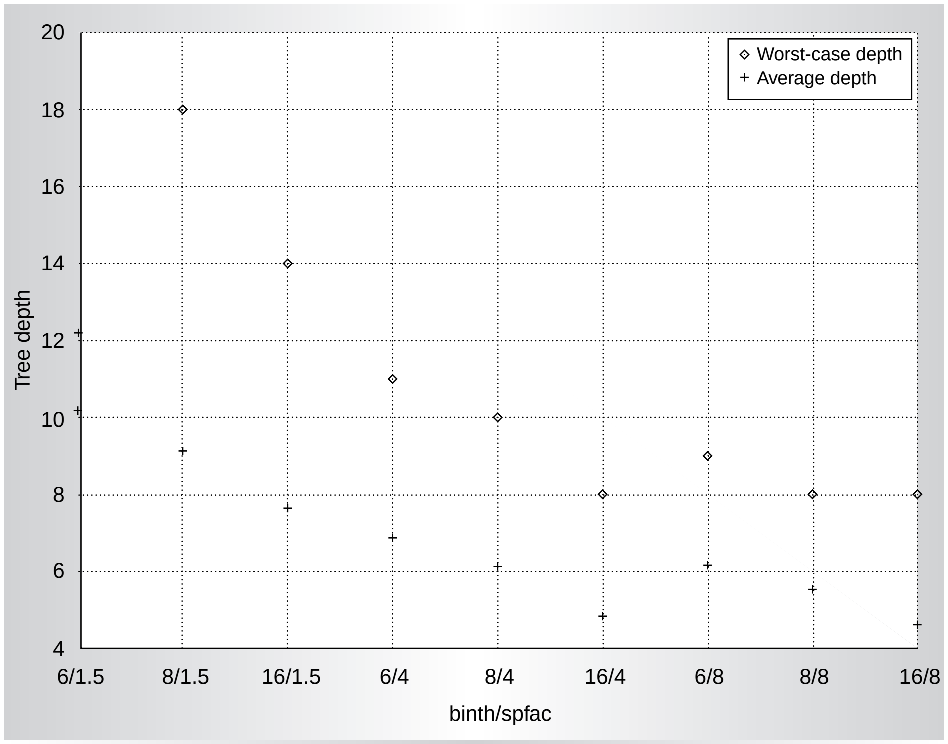

Figure 8. Tree depth variation with *binth* and *spfac* for a classifier with 1,733 rules.

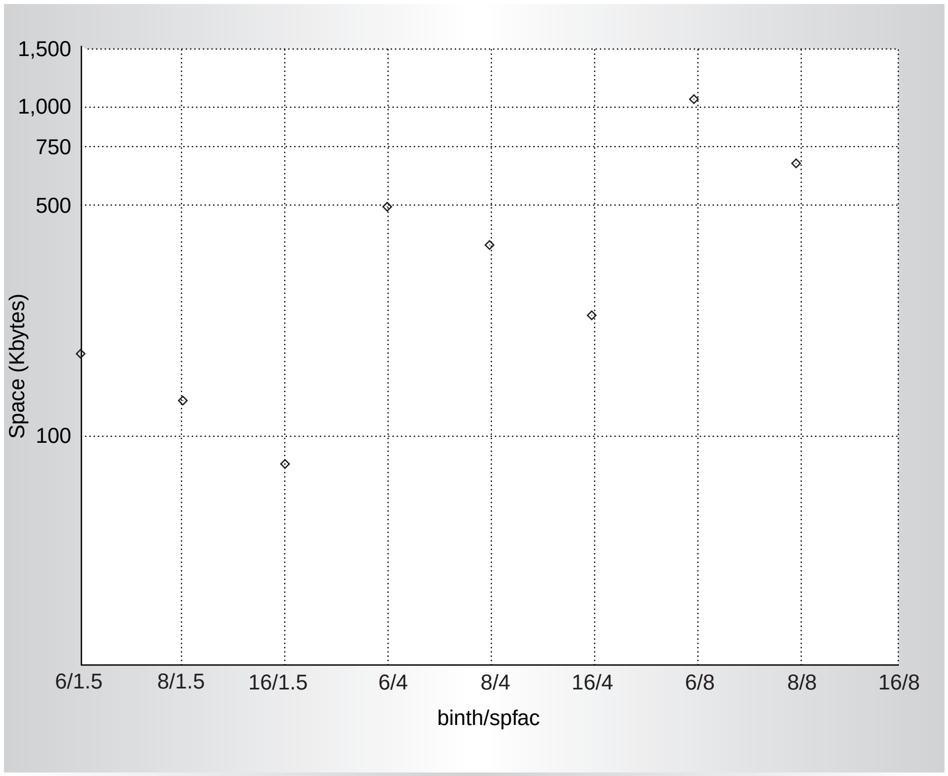

Figure 9. Memory consumption variation with *binth* and *spfac*.

from 1 to 70 milliseconds to incrementally update the data structure on an insertion or deletion, averaged over all the classifier's rules.

Variation with binth and spfac

Next, we show the effect of tuning parameters *binth* and *spfac* on the data structure for the largest four-dimensional classifier available to us, which contained 1,733 rules. In our experiments, *binth* took the values 6, 8, and 16; and *spfac* took the values 1.5, 4, and 8 for each value of *binth*. We made the following observations:

- The HiCuts tree depth is inversely proportional to *binth* and to *spfac* (Figure 8).
- The data structure storage requirement is directly proportional to *spfac* (as expected) but inversely proportional to *binth* (Figure 9).
- The preprocessing time is proportional to the storage consumed by the data structure (Figure 10).

Worst-case bounds on query time and storage requirements are so onerous as to make generic algorithms unusable, hampering the design of classification algorithms. Instead, we must search for the characteristics of real classifiers that we can exploit in pursuit of algorithms that are fast enough and storage-efficient. The difficulty of this task increases with the almost complete absence of classifiers (the set available to us is small and confidential and comes from a somewhat narrow range of networks). Even if classifiers were widely available today, we are not sure that they would represent the types of classifiers that future networks will use.

Like others before us, we have resorted to heuristics that make assumptions about classifier structure to reduce query time and storage requirements. A key difference is that we attempt to discover this structure at runtime for the classifier being preprocessed.

The HiCuts scheme performs well on the classifiers available to us, requiring smaller storage and comparable query time compared with schemes proposed previously. The complexity of the heuristics means that building the decision tree can take nearly a minute, but, fortunately, updating a rule in the data structure takes less than 100 milliseconds. Fast and flexible packet classification forms the basis for deployment of advanced Internet services. In this context, HiCuts seems to satisfy the requirements well and is a step toward a more intelligent Internet. MICRO

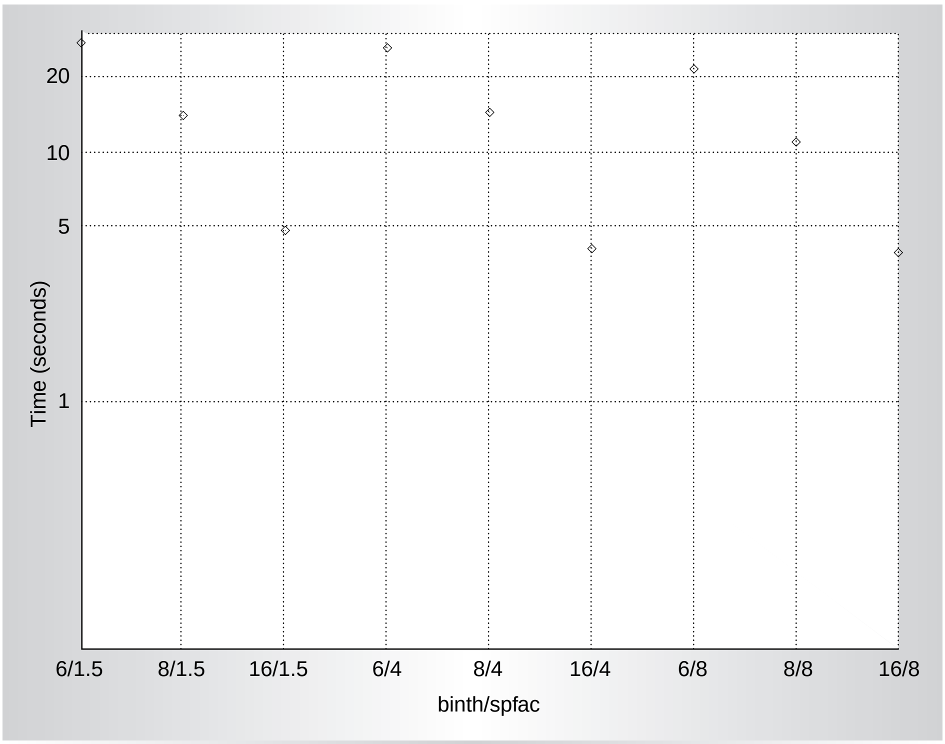

Figure 10. Preprocessing-time variation with *binth* and *spfac*.

Acknowledgment

We gratefully acknowledge Darren Kerr of Cisco Systems for providing us access to the classifiers we used in our experiments.

References

1. "Internet Routing Table Statistics," http://www.merit.edu/ipma/routing_table.
2. P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. ACM SIGCOMM 1999*, ACM Press, New York, pp. 147-160.

Pankaj Gupta is a PhD candidate in the Department of Computer Science at Stanford University. His research interests are fast packet switching and routing for the Internet. Gupta received a BTech in computer science and engineering from the Indian Institute of Technology, Delhi. He is a student member of the IEEE.

Nick McKeown is a professor of electrical engineering and computer science at Stanford University, where he works on the theory, design, and implementation of high-speed Internet routers and switches. He is currently on leave from Stanford and working at Abrizio Inc., Mountain View, Calif. Previously, he worked for Hewlett-Packard and Cisco Systems. McKeown is an editor of *IEEE Transactions on Communications*. The recipient of an NSF career award, he is the Robert Noyce Faculty Fellow at Stanford and a research fellow at the Alfred P. Sloan Foundation. McKeown received his PhD from UC Berkeley. He is a member of the IEEE.

Send questions and comments about this article to Pankaj Gupta, Computer Science Dept., Gates Bldg. 3A, Rm. 342, Stanford University, Stanford, CA 94305-9030; pankaj@cs.stanford.edu.