Packet Classification Using Multidimensional Cutting

Sumeet Singh
UC San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
susingh@cs.ucsd.edu

Florin Baboescu
UC San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
baboescu@cs.ucsd.edu

George Varghese
UC San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
varghese@cs.ucsd.edu

Jia Wang
AT&T Labs–Research
Florham Park, NJ07932-0971
jiawang@research.att.com

ABSTRACT

This paper introduces a classification algorithm called *HyperCuts*. Like the previously best known algorithm, HiCuts, HyperCuts is based on a decision tree structure. Unlike HiCuts, however, in which each node in the decision tree represents a hyperplane, each node in the HyperCuts decision tree represents a $k-$dimensional hypercube. Using this extra degree of freedom and a new set of heuristics to find optimal hypercubes for a given amount of storage, HyperCuts can provide an order of magnitude improvement over existing classification algorithms. HyperCuts uses 2 to 10 times less memory than HiCuts optimized for memory, while the worst case search time of HyperCuts is $50-500\%$ better than that of HiCuts optimized for speed. Compared with another recent scheme, EGT-PC, HyperCuts uses $1.8-7$ times less memory space while the worst case search time is up to 5 times smaller. More importantly, unlike EGT-PC, HyperCuts can be fully pipelined to provide one classification result every packet arrival time, and also allows fast updates.

Categories and Subject Descriptors

C.2.6 [**Internetworking**]: Routers—*Packet Classification*

General Terms

Algorithms

Keywords

Packet Classification, Firewalls, QoS

1. INTRODUCTION

In the last five years, a large number of papers on packet classification([1, 2, 3, 4, 5, 6, 7, 8]) have been published. Given that the course of packet classification is now considerably downstream from the fresh springs in which it had its source, any new paper must answer the question: *why is there yet another paper on packet classification?*.

We answer this challenge immediately with three propositions.

1, Importance: Packet classification continues to grow in importance, both at the edge and the core.

2, Performance of existing schemes: Existing algorithms still have poor performance, and ternary CAMs still have issues in terms of power consumption and chip density.

3, New ideas are possible: Despite the large number of ideas explored, there are still new ideas in packet classification that can provide major benefits.

Next, we amplify these three propositions to provide more detail.

1, Importance of Packet Classification: Both the core and the edge of the Internet are growing in speed. Dataquest claims that in 2004, 14% of the links between core routers will be OC-768 (40 Gbps), and 21% of edge links will be OC-192 (10 Gbps). Concurrently, demand for packet classification is increasing in order to provide QoS and security. These two trends (increased speed, increased use) combine to put pressure on router vendors, to the point that IP lookup, traditionally considered hard, is now considered a solved problem compared to classification.

What do customers use packet classification for? In packet classification, the handling of a packet can depend on additional header fields besides the destination IP address. Thus at the edge, packet classification can be used to mark IP headers (e.g., TOS, DSCP) with the appropriate QoS levels based on port fields that indicate the application. At both the edge and the core, packet classification can be used to discard or rate control certain offending application packets for the purposes of security.

The increased virtualization of the Internet via Virtual LANs and Virtual Private networks (VPNs) contributes to the growth of classifiers. VPNs are growing in importance, and a single ISP router must support multiple customer VPNs, each of which can contribute a classifier. Thus many classification chips for edge routers routinely support 32K rules.[1] Perhaps surprisingly, some ISPs require that even core routers [5] support up to 2800 rule classifiers.

2, Performance of Existing Schemes: Packet classification algorithms use two dominant resources, memory and

SIGCOMM'03, August 25–29, 2003, Karlsruhe, Germany.

[1]It is also worth noting the increased trend towards "deep" packet classification based on application headers and even on packet contents. While our algorithms can help with deep packet classification, this paper and the evaluation focuses on traditional rule databases based on IP 5-tuples.

time. The need for large memory can be finessed using cheap DRAM; however, speed requirements often dictate the use of expensive SRAM, making memory usage important. The need for speed can sometimes be finessed by pipelining but increased pipeline depths add expense, and pipelines larger than 32 stages are rare. Thus reducing worst-case search time in memory references is equally important.

All existing packet classification algorithms trade memory for time, ranging from schemes like RFC [2] (that is fast but takes excessive storage) to linear search (that is slow but takes minimal storage). The current algorithms with the best time-space tradeoffs appear to be EGT-PC [5] and HiCuts [1]. Unfortunately, while the tradeoffs have been constantly improving, the time taken for a reasonable amount of memory is still too poor for practical deployment.

Because of problems with existing algorithmic schemes, most vendors use Ternary CAMs, which use brute-force parallel hardware to simultaneously check for all rules. The main advantages of TCAMs over algorithmic solutions are *speed* and *determinism* (TCAMs work for all databases not just "typical databases").

Newer TCAM designs use aggressive banking techniques to reduce power, and better processes to increase density. However, CAMs fundamentally have to contend with reduced density (uses compare logic per bit) and increased power (uses parallel comparison). Two less fundamental problems are the need for rules with range specifications to be translated into several CAM entries, and the need for glue logic. It would be foolhardy for us to say that CAMs are not strong contenders. However, problems with CAMs have made vendors consider algorithmic alternatives. These include Cypress, Fast-Chip, EZchip, and Integrated Silicon [9].

Thus two reasons to continue to investigate new classification schemes are: *1, Need:* Given a need for CAM alternatives, any new algorithmic scheme that can improve the space-time tradeoff of existing schemes by an order of magnitude can be useful in practice. *2, Scientific Interest:* Decoupling from the exigencies of the market, new ideas for a fundamental geometric problem should be of interest to academic researchers.

3, New ideas: Besides improving performance on a wide range of classifiers (firewall and core router classifiers) by an order of magnitude, we introduce two intuitive new ideas.

1, Multidimensional Cutting: The classification problem can be considered geometrically as follows: given a set of boxes in N-space and a point, find the set of boxes that contain the point. While this problem is provably hard in the worst case, one of the best existing heuristic algorithms, *HiCuts* [1] solves this problem recursively by cutting the space of boxes at each step using a hyperplane. We explore the natural degree of freedom, which is to use a hypercube instead of a hyperplane. This allows our algorithm to simulate several cuts of HiCuts in one cut.

2, Pulling Rules up the Decision Tree: Recursive cutting can be embodied using a decision tree in which each node represents a cut and leaves represent rules. In general, some linear searching at the leaves is useful to reduce storage. Our observation is that a heavily wildcarded rule often ends up in many leaves, increasing storage unnecessarily. Instead, we simply move all *common* rules in a subtree to a linear list at the root of the subtree.

Thus packet classification is an important problem and there is a need for new algorithms. Finally, this paper introduces a new algorithm that uses two new ideas to (often) produce an order of magnitude improvement in performance. The rest of the paper is organized as follows. Section 2 formally describes the problem, Section 3 motivates our solution, Section 4 describes related work, Section 5 describes our new algorithm, Section 6 summarizes performance results, and Section 7 states our conclusions.

2. PACKET CLASSIFICATION PROBLEM

Individual entries for classifying a packet are called *rules/* The packet classification problem is to determine the first matching rule for each incoming message at a router.

The *classifier* or *rule database* in a router consists of a finite set of rules, $R_1, R_2 \ldots R_N$. Each rule is a combination of K values, one for each header field in the packet. Each field in a rule is allowed three kinds of matches: *exact match*, *prefix match*, or *range match*. In an exact match, the header field of the packet should exactly match the rule field—for instance, this is useful for protocol and flag fields. In a prefix match, the rule field should be a prefix of the header field—this could be useful for blocking access from a certain subnetwork. In a range match, the header values should lie in the range specified by the rule—this can be useful for specifying port number ranges.

A packet P matches rule R_i if all the header fields $Field_j$, $j = 1 \ldots k$ of the packet match the corresponding fields in R_i. If a packet matches multiple rules, the matching rule with the smallest index is returned.

3. WHY HYPERCUTS?

In this section, we motivate the use of hypercubes with a simple geometric example. The geometric view of classification is due to Lakshman and Stiliadis [3].

For example, a 32-bit prefix like $00*$ can be viewed as a range of addresses from $000\ldots00$ to $001\ldots11$ on the number line from 0 to 2^{32}. If prefixes correspond to *line segments* geometrically, what do rules correspond to? It is not hard to see that two dimensional rules correspond to rectangles, three dimensional rules correspond to cubes, and so on. A given address becomes a point. Thus from a geometric point of view the problem of packet classification reduces to finding the lowest cost box that contains the given point.

Figure 1 displays a toy example of a two dimensional classifier with 4 rules: $R_1 \ldots R_4$. Each rule is represented by a rectangle in two dimensional space. The left figure shows the action of HiCuts [1]. HiCuts builds a decision tree using *local optimization* decisions at each node to choose the next dimension to test, and how many cuts to make in the chosen dimension. The leaves of the HiCuts tree store a list of rules that may match the search path to the leaf.

The left part of Figure 1 shows how the HiCuts algorithm works on the example rule set. Assuming the maximum number of rules held in a leaf is 1, no matter how many cuts are going to be executed at a time, the HiCuts algorithm requires at least two levels in the decision tree.

The HyperCuts algorithm introduced in this paper eliminates this limitation in HiCuts by introducing one more degree of freedom. Each node in the decision tree represents a decision taken on the most representative *dimensions*, as opposed to using *only a single dimension*. For each of the

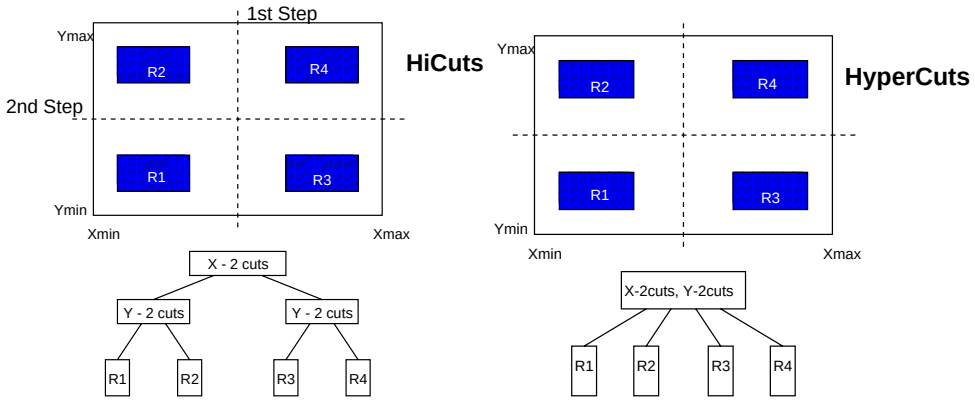

Figure 1: HiCuts vs. HyperCuts. A Geometric representation of a 4-rule classifier. HiCuts(on left) is applied to the 4-rule classifier. If the leaf node can only accommodate one rule than the decision tree in HiCuts has height at least 2. By contrast, HyperCuts (on right) can break the space into four smaller squares in one cut, resulting in a decision tree of height 1.

chosen dimensions, the number of cuts is computed based on a metric dependent on the amount of space that is available for the search structure. In the example in Figure 1 HyperCuts(on the right) cuts the plane into four squares with one direct cut, reducing the height of the decision tree to 1.

This is extremely reminiscent of how a B-tree can reduce the height of a binary tree by using a higher radix, say d. In the case of the B-tree, this does nothing to reduce the fundamental $\log_2 N$ bound on searching for one item among N items. This is because finding which of the d-pointers to follow at each node requires $\log_2 d$ time. Thus the reader may feel we are cheating: the use of higher dimensional cuts may slow down search times at each node, which can in turn offset any decrease in tree height.

However, this is not the case. We decide which pointer to follow at each node using essentially array indexing which costs one memory access regardless of the number of children at a node. It is easiest to see how array indexing works in one dimension. Imagine a 6-bit address space is partitioned into four equally spaced ranges (i.e., cuts) $[0-15]$, $[16-31]$, $[32-47]$ and $[48-63]$. Each range has an associated pointer, and the pointers are stored as four consecutive elements in an array.

To compute which pointer corresponds to a point, say 33, we find the quotient[2] when 33 is divided by the range width 16. Since the quotient is 2, we index into the third element of the array, assuming array indices start at 0. This simple indexing scheme can be generalized to multiple dimensions as long as the cut widths are fixed in each dimension. The bottom line is that search time at a node takes 1 memory access *regardless* of the number of cuts. Note that HiCuts also allows multiple cuts per node; it just restricts these cuts to be along one dimension. HyperCuts uses the extra degree of freedom to reduce tree height without sacrificing node search times.

The use of arrays can, however, increase storage because of empty and redundant pointers. We call such useless array locations "dead space". Fortunately, we can eliminate much of the "dead space" (as in HiCuts) by eliminating redundant subtrees. Whenever, two pointers point to identical subtrees, we eliminate one of the subtrees and make the corresponding pointer point to the other subtree. This converts the Decision tree into a Directed Acyclic Graph. Finally, we apply the heuristic of moving up common rules to reduce storage even further. We note that one could also eliminate empty array pointers using simple bitmap compression as in the Lulea [10] IP lookup algorithm.

Notice also that combining cuts in several dimensions as in HyperCuts can increase storage. For example, consider a HiCuts tree in which the root uses 2 cuts on $Field_1$, which then leads to two children A and B. Suppose A uses 8 cuts on $Field_2$, and B uses 2 cuts on $Field_3$. Then the HiCuts tree will only have $2+8+2=12$ pointers. However, if HyperCuts were to combine all the three fields in a single node, it would require $2*8*2=32$ pointers. But HyperCuts can always simulate HiCuts and not combine fields if the storage increase (relative to the gain in time) is large. We have found in our experiments that this extra degree of freedom in HyperCuts is extremely useful for core router databases, but less so for edge router databases.

4. RELATED WORK

The simplest classification algorithm is a linear search through the rules of the classifier. For a large number of rules this approach implies a large search time. However it is very efficient in terms of memory. Several algorithms have been developed for the case of rules on two fields [3, 8, 11, 6] but these do not solve the general problem of $K-$dimensional packet classification.

Srinivasan et al. [12] build a table of all possible field value combinations (cross-products) and *precompute* the earliest rule matching each cross-product. Search can be done quickly by doing separate lookups on each field, pasting the results together into a crossproduct, and indexing into the crossproduct table. Unfortunately, the size of this table grows astronomically with the number of rules.

In the bit vector linear search algorithm [3], search is first done in each dimension separately to yield the set of rules that match the packet in that particular dimension. These sets are then intersected efficiently using bitmaps to yield the set of rules that match in all dimensions. With hardware assistance, this algorithm works well for moderately size classifiers.

[2]This is easily computed in hardware using shifts.

Gupta and McKeown [1, 2] introduced two new algorithms, RFC (which is very fast but whose memory needs are large) and HiCuts (which we describe later). They also made the seminal observation that a given packet matches only a few rules even in large classifiers.

Baboescu and Varghese [4] exploit the sparse matching observation to reduce the search times for the algorithm described in [3]. Qiu et al [13] exploit the observation that any packet matches at most a few distinct values in each field to suggest backtracking trie search as a viable (though fairly slow) alternative.

The algorithms in the previous work with the best performance are HiCuts [1] RFC [2], and EGT-PC [5]. We evaluate HyperCuts against all these three algorithms in Section 6. We now describe *decision tree based classification algorithms* which are the starting point in the development of HyperCuts. To provide a running example, we consider the small firewall database in Figure 2. The example contains twelve rules on five fields.

4.1 Decision Tree Algorithms

Work on decision-tree based classification algorithms based on geometric cutting was pioneered in concurrent papers by Gupta and McKeown [1] and Woo[7]. Both schemes build a decision tree using *local optimization* decisions at each node to choose the next bit (or next field in the case of HiCuts [1]) to test. A simple criterion used in [1] is to balance storage and time.

The paper by Woo [7] also introduced a second important degree of freedom by considering *multiple* decision trees. For example, it may help to place all the rules with wildcards in both the source IP field and destination IP field in one tree, and the remainder in a second tree. While this can increase search time, it can greatly reduce storage. This is because rules with a large number of wildcards often end up replicated in most of the leaf nodes when using a *single* decision tree.

Both papers [7] and [1] use a small amount of linear searching after traversing the decision tree. Each leaf in the decision tree holds a small list of possible matching rules. During classification, the tree is traversed based on the packet header, and a leaf node is identified. The list of rules associated with the leaf node is then traversed to identify the highest priority matching rule. Consider a decision tree with 10,000 leaves; assume that each leaf is associated with up to 4 rules. While it may be possible to distinguish these 4 rules by lengthening the decision tree in height, this lengthened decision tree could add 40,000 extra nodes of storage.

Thus, in balancing storage with time, it may be better to settle for a small amount of linear searching (e.g., among one of 4 possible rules) at the end of tree search. Intuitively, this can help because the storage of a tree can increase exponentially with its height.

The Hierarchical Cuttings (HiCuts) scheme described in [1] is similar in spirit to [7] but uses range checks instead of bit tests at each node of the decision tree. Range checks are slightly more general than bit tests because a range check such as $10 < D < 35$ for a destination address D cannot be emulated by a bit test. A range test (cut) can be viewed geometrically in two dimensions as a line in either dimension that splits the space into half; in general, each range cut is a hyperplane.

We now describe HiCuts in more detail because we use it as point of departure. In HiCuts, each node can be regarded as a $k-$dimensional box cut up into a set of nc smaller boxes. The cutting is done using heuristics which take into account the structure of the classifiers. The size of a box depends on the range covered by the box in each dimension. For example, the root node for a $5-$tuple (IP Source and Destination, Port Source and Destination, Protocol) can be viewed as the box $[0, 2^{32}-1]X[0, 2^{32}-1]X[0, 2^{16}-1]X[0, 2^{16}-1]X[0, 2^{8}-1]$. Associated with each box is the set of rules which intersect the box.

Choosing the number of boxes a node is split into (nc), requires several heuristics which tradeoff the depth of the decision tree versus the tree memory space. The dimension on which a cut may be executed is chosen, roughly speaking, to minimize the maximum number of rules in any partition. Picking the number of partitions (nc) also clearly affects the overall memory space. The algorithm tunes nc as a function of a space measure. It uses two parameters: (1) *binth* (which limits the amount of linear searching at leaves) and (2) *spfac* (a multiplier which limits the amount of storage increase caused by executing cuts at a node).

Figure 3 shows a decision tree for the Example in Figure 2. A range representation for the set of rules in Figure 2 is shown in Figure 4. Assume that a packet with the header $(0010, 1101, 00, 01, TCP)$ needs to be classified. The path followed by this packet is shown in Figure 3. At the root node, marked A, based on the value in its second field the packet is directed to the node marked B. At Node B, search uses information in the third field to direct search to a leaf node containing a small list of two possible matches. In this case, R_7 is the lowest cost rule matching the packet.

For this example, observe that even if we were able to create a number of partitions(cuts) based on each of the distinct values that occur in the chosen dimension, we cannot get a height of 1 for this tree, assuming that the maximum length of the list of rules at a leaf is less than 4. Figure 5 shows that even if we make 16 partitions on the second field (which is the field with the largest number of unique elements) the partition associated with a $Field_2 = 10$ contains 6 rules, which in turn requires one more node for segregating the rules into groups of at most 4. By contrast, the HyperCuts tree will be able to obtain a height of 1.

Secondly, observe also that all the children of the node on $Field_4$ share the set of rules $\{R_7, R_{10}, R_{11}\}$. This duplication contributes to an increase in the memory used by HiCuts which we can eliminate to some extent in HyperCuts by pulling up common rules to the appropriate ancestor.

5. HYPERCUTS DESCRIPTION

HyperCuts is a decision tree based algorithm. At each node in the decision tree, the set of current rules is split based on information from one *or more* fields in the rule. Each time a packet arrives, the decision tree is traversed based on information in the packet header to find a leaf node. A small number of matching rules that are stored in the leaf node are linearly traversed to find the highest priority rule that matches the packet. This basic structure is similar to the work in [7, 1] except for the possible use of two or more fields at each node. Each node in the decision tree has associated with it:

$Rule$	$Field_1$	$Field_2$	$Field_3$	$Field_4$	$Field_5$	$ACTION$
R_0	000∗	111∗	10	∗	UDP	act_0
R_1	000∗	111∗	01	10	UDP	act_0
R_2	000∗	10∗	∗	10	TCP	act_1
R_3	000∗	10∗	∗	01	TCP	act_2
R_4	000∗	10∗	10	11	TCP	act_1
R_5	0∗	111∗	10	01	UDP	act_0
R_6	0∗	111∗	10	10	UDP	act_0
R_7	0∗	1∗	∗	∗	TCP	act_2
R_8	∗	01∗	∗	∗	TCP	act_2
R_9	∗	0∗	∗	01	UDP	act_0
R_{10}	∗	∗	∗	∗	UDP	act_3
R_{11}	∗	∗	∗	∗	TCP	act_4

Figure 2: A simple example with 12 rules on five fields.

$Rule$	$Field_1$	$Field_2$	$Field_3$	$Field_4$	$Field_5$	$ACTION$
R_0	$0-1$	$14-15$	2	$0-3$	0	act_0
R_1	$0-1$	$14-15$	1	2	0	act_0
R_2	$0-1$	$8-11$	$0-3$	2	1	act_1
R_3	$0-1$	$8-11$	$0-3$	1	1	act_2
R_4	$0-1$	$8-11$	2	3	1	act_1
R_5	$0-7$	$14-15$	2	1	0	act_0
R_6	$0-7$	$14-15$	2	2	0	act_0
R_7	$0-7$	$8-15$	$0-3$	$0-3$	1	act_2
R_8	$0-15$	$4-7$	$0-3$	$0-3$	1	act_2
R_9	$0-15$	$0-7$	$0-3$	1	0	act_0
R_{10}	$0-15$	$0-15$	$0-3$	$0-3$	0	act_3
R_{11}	$0-15$	$0-15$	$0-3$	$0-3$	1	act_4

Figure 4: A range based representation of the example with 12 rules on five fields shown in Figure 2.

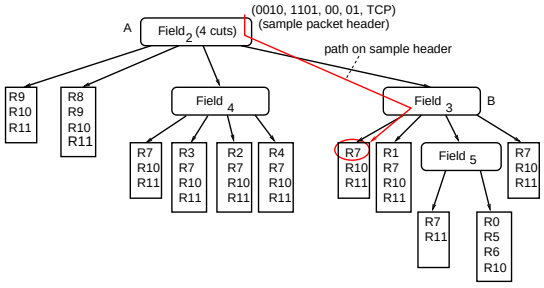

Figure 3: A HiCuts decision tree built for the database of Figure 2. $Field_2$ is chosen for the root node because it has the largest number of unique values. A sample packet header and a sample search path are also shown.

- **i:** A region $R(v)$ that is covered. In the case of a 5−field classifier, this can be represented as a 5 tuple: *[IPSmin-IPSmax, IPDmin-IPDestmax, PSmin-PSmax, PDmin-PDmax, ProtMin-ProtMax]*;
- **ii:** A number of cuts (NC) and a corresponding array of NC pointers;
- **iii:** A list of rules that may match. The maximum number of rules stored in a node is predetermined.

Figure 6 shows HyperCuts in action. The decision tree is built for the same database as in Figure 4. The tree consists of a *single* root node which covers the region $[0-15, 0-15, 0-3, 0-3, 0-1]$ that is split into sub-regions with 16 cuts. There are 4 cuts based on field $Field_2$, 2 cuts based on $Field_4$, and 2 cuts based on the $Field_5$. Because of the difficulty of drawing a 3-dimensional array on three fields, Figure 6 shows the root node in terms of its projections on $Field_2$. This results in four 2-dimensional subarrays for each of the four possible ranges of $Field_2$. Each 2D subarray is built on $Field_4$ (shown vertically) and $Field_5$ (shown horizontally). Note that since there are 4 cuts in $Field_2$, the assumed 4-bit field splits into four ranges of equal size $0-3, 4-7, 8-11$ and $12-15$.

As an example, consider the fourth and rightmost subarray, and the rightmost element in the second row of this subarray. This corresponds to a $Field_4$ range of $0-1$, and a $Field_5$ value of 1, and hence represents the leaf node associated with the region $[0-15, 12-15, 0-3, 0-1, 1]$. This subarray element has two associated rules: R_7 and R_{11}. It is easy to verify from the rule set of Figure 4 that these are the two only rules that match this region.

Note that the HyperCuts algorithm for this database reaches any of the leaf nodes in at most one step. By comparison, the HiCuts algorithm (as shown in Figure 5) cannot reach the leaf nodes in less than two steps (for an equivalent bucket size).

To describe HyperCuts, we need to describe two different algorithms. The first is the *Preprocessing Algorithm* in which the decision tree is built based on the rules in the

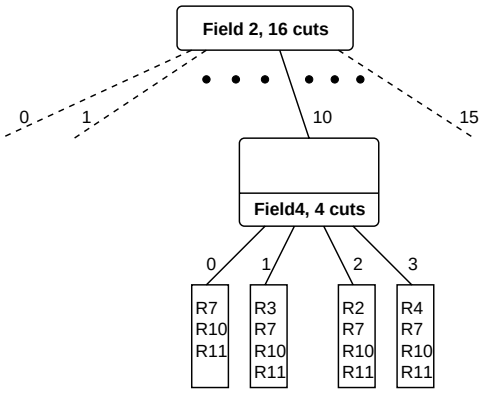

Figure 5: This picture helps explain why no HiCuts tree for the database of Figure 2 can have height equal to 1. Compared to Figure 3, even if we increase the number of cuts in $Field_2$ to the maximum number that it can possibly use (16), the region associated with $Field_2 = 10$ still has 6 rules. This in turn requires another search node because linear searching at a leaf is limited to 4 rules.

classifier. The second algorithm is the *search algorithm*, in which for any packet the tree is traversed to identify the matching rule. A fast update algorithm can also be implemented; however we do not go into the details of incremental update in this paper.

Before describing our algorithm, we make some observations:

- **i.** The decision tree should try at each step(node) to eliminate as many rules as possible from further consideration.
- **ii.** The maximum number of steps to be taken during a search should be minimized.
- **iii.** Certain rules may not be able to be segregated without a further increase in the overall complexity of the algorithm (both space and time). Therefore a separate approach should be taken to deal with them. Rules in this category are rules with wildcards (any) in both IP source and IP destination fields.[3]
- **iv.** As in any packet classification scheme there is always a tradeoff between the search time and the memory space occupied by the search structures. As with [7, 1], when the number of rules is small, linear search may be a tradeoff. That is why, in order to reduce the overall memory space and to keep the depth of the decision tree at a minimum, a node is not subdivided if the number of rules associated with the node is smaller than a predefined threshold.

Our approach takes the four observations into consideration to build a tree as follows. At each node **(1)** it identifies the dimensions (fields) with the highest number of distinct elements **(2)** for each of the identified dimensions it determines the number of cuts to be done based on a tradeoff between the depth of the tree to be obtained and the memory size that is available, **(3)** it executes the cuts on the chosen dimension creating a number NC of children to the number of cuts that are executed. No further cuts are executed if the number of rules associated with a node are smaller than a predetermined value that we call $bucketSize$.

[3]A separate decision tree could be built for the subset of rules that contain wildcards in both IP source and destination fields. Both the original decision tree as well as the additional one are traversed during the search. For any packet the matching rule is the highest priority rule from the union of the result sets.

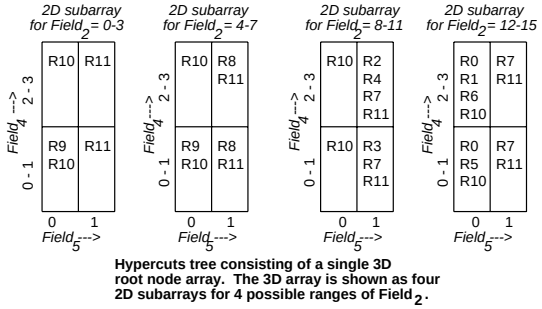

Figure 6: The HyperCuts decision tree for the database of Figure 4 consists of a *single* 3-dimensional root array built using 4 cuts on field $Field_2$, and 2 cuts each on $Field_4$ and $Field_5$. The 3D root array is shown as four 2D subarrays for the four possible ranges of $Field_2$. Contrast this single node tree with Figure 5 which shows that any HiCuts tree must have height at least two.

5.1 Building the HyperCuts Tree from Scratch

The algorithm starts with a set of N rules, each of the rules containing K dimensions. A subset of all the rules containing wildcards in both IP source and IP destination fields is created. We call this *W-Set*. We call *R-Set* the subset of remaining rules after extracting *W-Set*. A decision tree is built for both *R-Set* and *W-Set*.

Each node identifies a region and has associated with it a set of rules S that match the region. If the size of the set of rules at the current node is larger than the acceptable bucket size, the node is split in a number (NC) of child nodes, where each child node identifies a sub-region of the region associated with the current node. Identifying the number of child nodes as well as the sub-region associated with each of the child nodes is a two step process, which tries to locally optimize the split(s) such that the distribution of the rules among the child nodes is optimal. This process includes **(1)** identifying the most suitable set of dimensions to split and **(2)** determining the number of splits to be done in each of the chosen dimensions.

5.1.1 Choosing the dimensions

The challenge is to pick the dimensions which will lead to the most uniform distribution of the rules when the node is split into sub-nodes. To the best of our knowledge there is no consummate method of picking this set of dimension(s). Therefore, we propose a set of heuristics to help make an effective decision.

A first solution is to consider the set of dimensions with the largest number of unique elements. However, this is not very satisfactory because choosing more dimensions will almost always increase the number of unique elements, and thus the algorithm might always choose all 5 dimensions. But this can have adverse affects on storage because of the

storage cost of 5-dimensional arrays. On the other hand, observe that if adding a dimension adds only a few unique elements, then the small increase in search discrimination is probably not worth the extra storage penalty.

To quantify this point of diminishing returns, we consider instead the set of dimensions for which the number of unique elements is *greater than the mean* of the number of unique elements for all the dimensions under consideration. For example, if for the five dimensions the number of unique elements in each of the dimensions are: 45, 15, 35, 10 and 3 with a mean of 22, then the dimensions which should be selected for splitting are the first and the third. This is because these have values greater than the mean.

We have also experimented with the ratio of the number of unique elements to the size of the region represented by that dimension. For example consider the root node which covers the region $[0 \dots 2^{32} - 1]$ for IP source and destination fields, and a region $[0 \dots 2^{16} - 1]$ for source and destination port numbers. If the number of unique elements in the IP Source and the number of unique elements in the source port number is the same, then it may be logical to first select the source port as the first field to split because the resulting split region sizes are smaller than when using the IP source field. Formally, we can use as a measure the ratio of the number of unique elements to the total number of possible values covered by the range representing the dimension.

5.1.2 Picking the number of cuts

Once the set of dimensions (D) on which the splits are to be executed at a node is chosen, the next step is to establish the number of cuts to execute in each of the dimensions. This means picking the set of numbers $\{nc(i)\}_{i \in D}$, where $nc(i)$ represents the number of cuts to be executed on the $i-th$ dimension.

Since our goal is to create a search tree with minimal memory requirements, we steal a leaf from the HiCuts building algorithm and limit the maximum number of child nodes that the current node can be split into by a factor of the number of rules contained in the node. We define this as the function $f(N) = spfac * \sqrt{N}$ where N is the number of rules in the current node and $spfac$ is a space factor parameter that can be varied to tradeoff storage against time.

The total number of split operations to be executed is $NC = \prod_{i \in D} nc(i)$. Ideally we should try all possible combinations of $\{nc(i)\}_{i \in D}$ where the $NC = \prod_{i \in D} nc(i)$ is bounded by $f(N) = spfac * \sqrt{N}$ to determine which set of $\{nc(i)\}$ provides the best distribution with the least amount of memory increase. However, considering every possible combination is computationally infeasible. Hence, we select the following greedy approach: we first choose separately, for each dimension i, the local optimum number of cuts $nc(i)$ to be executed, and then determine the best combination centered around these values.

To identify the number $nc(i)$ of cuts for each of the cutting dimensions we keep track of: **(1)** the mean of the number of rules in each of the child nodes, **(2)** the maximum number of rules in any one of the child nodes, **(3)** the number of empty child nodes[4]. A set of iterations are executed; at each step the current value for $nc(i)$ is multiplied by two.

If after a number of subsequent steps there is no significant change in the mean or the maximum number of rules in the child nodes, or there is a significant increase in the number of empty child nodes, then we backtrack and use the last known best value as the chosen number of splits to be made along the dimension under consideration.[5]

5.1.3 Algorithm Refinements

The number of child nodes as well as the number of rules stored in a node have a direct relationship to the memory space occupied by the search structure. Therefore we consider the following mechanisms to reduce them and implicitly to reduce the memory space occupied by the algorithm.

We use a set of four heuristics for doing these reductions. They are based on: **(1)** node merging, **(2)** rule overlap, **(3)** region compaction and **(4)** identifying a common subset of rules which are covered by all the child nodes. The first two heuristics were also discussed in [7, 1]. However the last two are introduced here for the first time.

Node Merging: We reduce the memory space occupied by the algorithm by merging the nodes which have associated with them the same set of rules. Figure 8 shows a situation in which two nodes that share the same set of rules are merged into a single node. The single node has been assigned the same set of rules as the previous ones, and covers a contiguous region that is the union of the regions of its children.

Rule Overlap: The leaf nodes store a list of rules which are matched by values from the subregion covered by the node. However there may be situations, such as in Figure 7, in which a rule R_2 is assigned to a node in which there is a rule R_1 with a higher priority which also covers the whole region covered by the rule R_2 in the node. In this case there is no reason to store the rule R_2 in the node because it will never be chosen. As a result rule R_2 may be eliminated from the list of rules that are covered by the node.

Region Compaction: Each node has associated with it a region that it covers as well as a set of rules that are a match. However there are cases as in Figure 9 in which the region covered by the rules is smaller than the overall size of the region associated with the node. Therefore a reasonable optimization is to shrink the region associated with the node to the minimum cover that includes the areas covered by all the rules in the set of rules associated with the region. Figure 9 shows an example in which a node has an associated region $\{[X_{min}, X_{max}], [Y_{min}, Y_{max}]\}$ and a set of rules it covers $\{R_1, R_2, R_3, R_4\}$. The area associated with the node is reduced to $\{[X'_{min}, X'_{max}], [Y'_{min}, Y'_{max}]\}$ which is the minimum cover that includes the areas covered by all the rules $\{R_1 \dots R_4\}$.

Pushing Common Rule Subsets Upwards: In this heuristic, if all the child nodes have associated a subset of rules that are identical, then the parent node will store this subset instead of the children. If we choose to do this optimization, then rules may be associated with non-leaf nodes. Searching for such rules in non-leaf nodes can add an unnecessary memory access at each node with an empty list if implemented naively using a list pointer. Instead, we use a bitmap in the header of each node to distinguish between nodes that have empty and non-empty lists without using an extra memory access.

[4]These need to be taken into consideration in order to avoid a possible memory blowup

[5]A hash based cut is done for the fields in which an exact match needs to be executed (e.g. the protocol field)

The procedure to implement this optimization executes a bottom-up traversal of the tree in order to identify all the possible situations. Figure 10 shows an example in which all the child nodes of A share the same subset of rules $\{R_1, R_2\}$. As a result A will store this subset of rules $\{R_1, R_2\}$ instead of being kept at the children.

The pseudocode for the tree building algorithm is:

```
1  CreateNodeP(l_1, r_1, l_2, r_2, ..., l_k, r_k, R);
2  if (|R| < bucketSize) return;
3  for i ← 1 to k do
4    N_i ← numberOfUniqueValuesOnDim(R, i);
5  Mean ← mean(N_1 ... N_k);
6  for i ← 1 to k do
7    if N_i > Mean then Dims ← Dims ⋃ {i};
6  for i ∈ Dims do
7    NC(i) ← optimumNoCutsOnDimension(i, l_i, r_i, R);
8  N ← ∏_{i∈Dims} NC_i;
9  for i ← 1 to N do
10   (l_1^i, r_1^i, ..., l_k^i, r_k^i, R^i) ← createCut();
11   CreateNode(l_1^i, r_1^i, ..., l_k^i, r_k^i, R^i);
12 return;
```

Line 8 in the actual algorithm is subtler than the simplified pseudocode shown above. The dimensions are ranked first by unique elements and then (in case of ties) by the ratio of unique elements to the size of the region. Then the algorithm considers the candidate dimensions in ranked order, first calculating the optimal number of cuts for that dimension (following a procedure similar to HiCuts), choosing this dimension only if allowed by space factor constraints. At the end of this procedure, the pruned set of dimensions is optimized further by considering every possible subset of the pruned set, and choosing the best subset in terms of a storage/time tradeoff. Note that this may result in choosing a single dimension as in HiCuts.

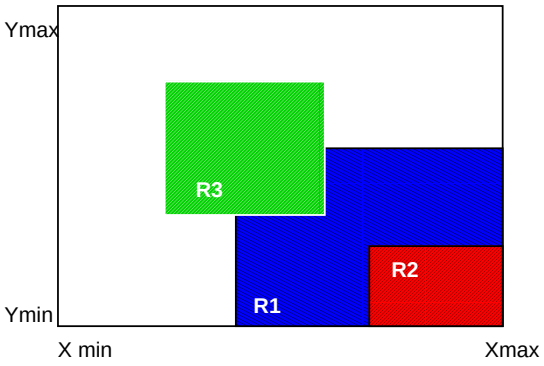

Figure 7: A two dimensional region $\{[x_{min}, x_{max}], [y_{min}, y_{max}]\}$ associated with a node which has assigned three rules R_1, R_2, R_3. The highest priority rule is R_1 followed by the rules R_2 and R_3. The node does not need to keep track of rule R_2 because any of the packets which might be associated with R_2 are also covered by the rule R_1 that has a higher priority.

5.2 HyperCuts Search Algorithm

We explain the search algorithm by first going through a small example. Figure 11 shows a node A in the decision tree structure together with a packet header that has arrived

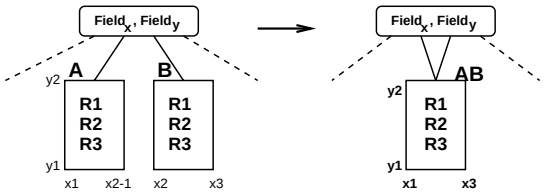

Figure 8: A node in the decision tree is split into 4 child nodes each one of them associated with a hyper region by doing cuts on two dimensions X and Y. The child nodes A and B cover the same set of rules R_1, R_2, R_3 therefore they may be merged into a single node AB associated with the hyper region $\{[x_1, x_3], [y_1, y_2]\}$ that covers the set of rules R_1, R_2, R_3.

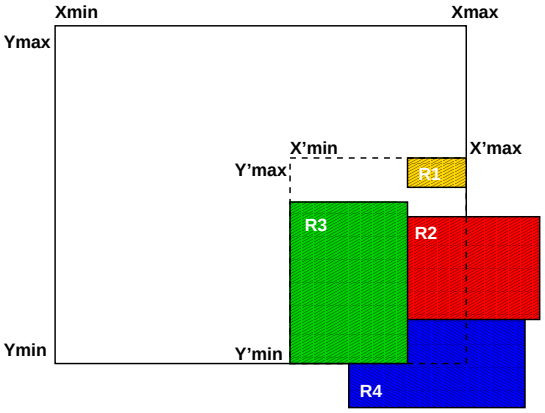

Figure 9: A node in the decision tree originally covers the region $\{[X_{min}, X_{max}], [Y_{min}, Y_{max}]\}$. However all the rules that are associated with the node are only covered by the subregion $\{[X'_{min}, X'_{max}], [Y'_{min}, Y'_{max}]\}$. Using region reduction the area that is associated with the node shrinks to the minimum space which can cover all the rules associated with the node. In this example this area is:$\{[X'_{min}, X'_{max}], [Y'_{min}, Y'_{max}]\}$.

at this node. The packet header has the value $X = 215$ and $Y = 111$. The current node covers the regions $200 - 239$ in the X dimension and $80 - 159$ in the Y dimension. During the search the packet header is escorted by a set of registers carrying information regarding the hyper-region to which the packet header belongs at the current stage. In this example the current hyper-region is $\{[200 - 239], [80 - 119], \dots\}$[6]

Node A has 16 cuts, with 4 cuts for each of the dimension X and Y. To identify the child node which must be followed for this packet header, the index in each dimension is determined as follow. First, $X_{index} = \lfloor \frac{215-200}{10} \rfloor = 1$. This is because each cut in the X dimension is of size $(239 - 200 + 1)/4 = 10$. Similarly, $Y_{index} = \lfloor \frac{111-80}{20} \rfloor = 1$. This is because each cut in the Y dimension is of size $(159 - 80 + 1)/4 = 20$[7]

[6]The hyper-region associated with the packet header is different than the hyper-region covered by the node A because the node A is obtained as a result of merging two nodes that cover the hyper-regions $\{[200 - 239], [80 - 119], \dots\}$ and $\{[200 - 239], [120 - 159], \dots\}$ respectively.

[7]The division operation can be easily replaced with a binary shift operation by using a multiple of two for the number of cuts.

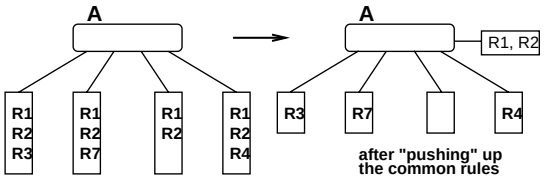

Figure 10: An example in which all the child nodes of A share the same subset of rules $\{R_1, R_2\}$. As a result only A will store the subset instead of being replicated in all the children.

As a result the child node B is picked and the set of registers is updated with the new values describing the hyper-region covering the packet header at this stage. This hyper-region is now: $\{[200-219],[100-119],\dots\}$

The search ends when a leaf node is reached in which case the packet header is checked against the fields in the list of rules associated with the node.

The pseudocode for the search algorithm (using F_i for $Field_i$ for brevity) is:

```
1  Search(F_1, ... F_k);
2  Node curNode = root;
3  for i ← 1 to k do
4    regionL[i] ← Min[i];
5    regionR[i] ← Max[i];
6  while curNode! = LEAF do
7    for i ∈ curNode.Dims do
8      cut[i] ← ⌊(F_i − curNode.MinDim[i]) / curNode.Δ[i]⌋;
9      regionL[i] ← curNode.Dims[i].left[cut[i]];
10     regionR[i] ← curNode.Dims[i].right[cut[i]];
11     curNode ← curNode.child(cut[0], cut[1], ... cut[k]);
12 matchRule ← findMatchRule(curNode.listRules);
13 return matchRule;
```

The search for a packet with a $k-$dimensional header $F_1,\dots F_k$ starts with an intialization phase (steps $2-5$) in which the current node of the search is set to the root node of the search structure, and the regions which cover the packet header are set to the maximum value of the ranges for each of the dimensions. For example, if the first two dimensions correspond to IP values than these values are 0 and $2^{32}-1$ respectively.

The next steps $(6-11)$ traverse the decision tree until it finds either a leaf node or a $NULL$ node. At each step in the traversal it updates the hyper-regions that cover the values in the packet header

Once a leaf node is found on step 12 the list of rules associated with this node is traversed and the first matching rule is returned in step 13. If there is no match a $NULL$ is returned;

6. EVALUATION

6.1 Classifier Characteristics

We evaluate HyperCuts both on firewall databases as well as on edge and core router databases. All the router rule databases are from major ISPs. CR_1 through CR_4 are rule databases provided by four major ISPs. ISP_2 is the provider for the databases $ER_1\dots ER_4$ which are considered to be representative for the edge router databases[8]. The firewall

[8]While the other ISPs use a large amount of rules in their core routers, ISP_2 uses very small core router databases which we did not study here. However, ISP_2 did have large edge router databases. We investigated about $38,000$ of these databases, each with up to $5,000$ rules per database; because of space limitations, we show the results on only four sample edge router databases.

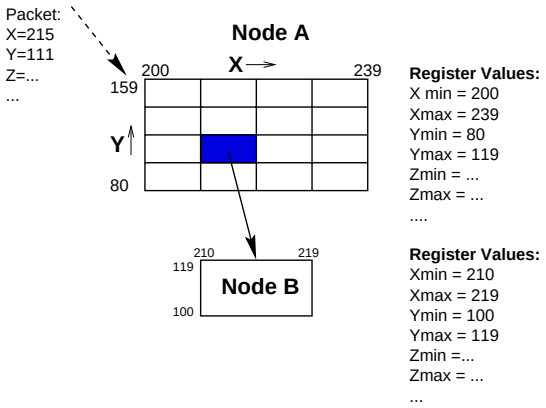

Figure 11: A search through the HyperCuts decision tree in which a packet arrives to a node that covers the regions $200-239$ in the X dimension and $80-159$ in the Y dimension. The packet header has the value 215 in the X dimension and 111 in the Y dimension.

databases are named $FW_1\dots FW_4$, and were obtained from real firewalls.

The number of rules in the core router classifiers varies from 85 to 2800 as is shown in Figure 12. All the classifiers are five dimensional with the IP source and destination field represented as prefixes while the port fields are represented as ranges.

In the case of core router databases, with the exception of one database which appears to have rules connecting sub-networks (prefix lengths with values of $16-24$), all the other databases have similar maximums at lengths 0, 16, 24 and 32. The prefix length distribution in the case of firewall databases has maximums at lengths 0 and 32, while for other lengths the distribution is much more uniform than in the case of the core router classifiers.

In the case of the edge router databases, the IP prefix length distribution has values which are identical for both source and destination. With very few exceptions, the prefixes in the classifier all have length 32. More specifically, most of the rules are made up of pairs of prefixes with the same length. The only exception to this rule is when a rule contains a wildcard in one of the fields. This appears to be be a consequence of the policies used by ISP_2.

The number of rules matching all five fields is somewhere between 3 and 5 for the core router databases, and up to 7 matches in the case of the firewall databases. This result is consistent with results by Gupta and McKeown in [2, 1]. A value of 3 is easily achieved by a classifier which contains a default rule to be executed on all packets, a second rule to be executed on all the packets carrying a TCP message, and a third rule to be executed on all packets for an *established* TCP connection.

For more details regarding the structure of the rule sets, the interested reader may consult [14].

Overall, analyzing the rule sets we come to the surprising conclusion that packet classification databases in the core,

edge, and firewall spaces have totally different characteristics. This is in sharp contrast to the uniform testing methodology used in past papers.

6.2 Metrics

We wish to do packet classification at wire speed for minimum sized packets and thus speed is the dominant metric. We focus on *worst case search time* expressed in number of memory accesses.

To allow the database to fit in high speed memory it is crucial to also reduce the amount of storage. On-chip SRAM can provide latencies of around $1-4$ nsec per operation. However, on-chip SRAM is limited in size. For example, in a .13 micron technology the area occupied by about 16 Mbits of SRAM is around $85mm^2$. By comparison, an ARM 966E core without any additional caches occupies only about $1mm^2$ using the same technology. Therefore, the second metric we study is the *space* occupied by the search structure.

6.3 Performance on Real Life Classifiers

As mentioned earlier, we evaluate our algorithm on a set of classifiers which we consider to be representative for core routers, edge routers, and firewalls. The memory space used by the HyperCuts search structure depends on the number and size of nodes. A node consists of a header plus an array of pointers to child nodes, one for each cut. The header size is 4 bytes, each pointer takes 4 bytes, and the number of entries in the array is equal to the number of child nodes[9]. A bitmap in the header is used to distinguish between types of nodes.

The code used for the other classification schemes can be found in our public repository [15]. We start by showing the results for core router databases. Figure 12 displays the memory utilization for HyperCuts vs. memory utilization for RFC, ABV, EGT-PC and HiCuts while Figure 13 shows the worst case search time for the same algorithms. Our results show that the main contenders against HyperCuts are HiCuts and EGT-PC. However, in terms of memory utilization, HyperCuts uses up to an order of magnitude less memory than either HiCuts optimized for space or EGT-PC. In terms of worst case search time, HyperCuts is 3 to 10 times faster than HiCuts. In what follows, we show the results of HyperCuts versus the two best previous algorithms: HiCuts and EGT-PC.

Despite using ACL lists, the edge router databases only specify the two fields for IP source and destination, and most with length 32 prefixes in these two fields; two dimensions does not provide sufficient degrees of freedom for HyperCuts to differentiate itself from HiCuts. This can be clearly seen in Figures 14 and 15.

The results of running HiCuts, HyperCuts and EGT-PC on firewall databases are shown in Figure 16 and 17. The firewall databases are distinctly different from the core router databases as they consist of a large number of unique ranges, as well as a large number of values in the protocol and the port fields. Growth in the number of popular applications has a direct impact on the number of port and protocol combinations in firewall databases. More importantly, there are a large number of rules with wildcards in either the source or the destination IP field. All these contribute to an increase in the effectiveness of cutting on more than one field at a time as in HyperCuts.

If we consider cutting on only one dimension as in HiCuts, the selection picks the dimension with the largest number of unique elements to cut on. For firewall databases, the source and destination IP fields have the largest number of unique elements. However, both fields also have a large number of wildcards. As a result, using either of the IP fields for cuts results in replicating a large number of rules, and hence limits the number of cuts due to storage factor limits.

By contrast, HyperCuts allows cutting both source and destination IP fields in a single node, which not only disperses the rules among the child nodes in a single step but also reduces the effect of rule replication. Further, in HyperCuts pushing up common sets of rules reduces the damage due to replication. For the firewall databases under consideration this optimization resulted in a memory reduction of 10%. Overall for firewall databases, HyperCuts uses an amount of memory similar to EGT-PC while its search time is up to 5 times better than HiCuts optimized for speed.

6.4 Performance on Synthetic Classifiers

We have seen that edge, core, and firewall databases have very different attributes. Thus it is inappropriate to build a single synthetic model of a classifier to test scalability. Instead, we used the real life databases we already had as generators to produce new synthetic databases of larger sizes.

To create a new rule, we randomly pick two prefixes from a pool of values that follows the same prefix distribution as the one found in the original databases, one for the source and one for the destination. We append the other fields — source and destination port as well as protocol number — by randomly picking them from a pool of all the values that were in the corresponding real life generator. This procedure is then iterated N times to create a classifier of size N.

Our evaluation results are shown in Figure 18. For synthetic core-router style databases of size 20000, HyperCuts requires only 11 memory access for search in the worst case; for edge-router style databases, HyperCuts requires 35 memory for a database of 25000 rules. In the case of firewall-like databases, the presence of about 10% wildcards in either of the source and destination IP field contributes to a steep memory increase. This is possibly because of a large number of rules replicated in leaves.

7. CONCLUSIONS

The papers in [1, 7] pioneered work on packet classification based on decision trees and geometric cuts, but chose only a single field at a time to cut on. HyperCuts introduces one more degree of freedom compared with these algorithms by allowing more than one dimension to be cut within a single node, while still using only one memory access per node. The decision procedure we use for HyperCuts to choose dimensions is inspired by HiCuts but differs from it in several subtle ways including the use of the mean number of unique elements, the region density, and a different space factor function.

[9]The refinements discussed in section 5.1.3 increase the header size to maintain information about the region covered by a node. Use of region compaction requires saving the difference in offsets between new and old regions covered; node merging also requires storing the new borders of the region in the header. Overall, this can result in an increase of $2-8$ bytes per dimension.

$Database$	$No.Rules$	RFC	$HiCuts-4$	$HiCuts-1$	ABV	$EGT-PC$	$HypCuts-4$	$HypCuts-1$
CR_1	85	55,202	11,608	1,346	1,572	1,168	453	226
CR_2	125	114,080	10,704	1,986	1,606	1,472	589	610
CR_3	351	100,991	64,541	19,001	4,651	2,261	15,395	11,210
CR_4	2799	747,271	117,801	25,543	285,099	30,753	16,631	11,030

Figure 12: The total memory space occupied by the search structure in all 5 heuristics RFC, HiCuts($spfac = 1, 4$), BV, ABV, EGT-PC and HyperCuts for the four core router databases. The size is in memory words, one memory word is 32 bits.

$Database$	$No.Rules$	RFC	$HiCuts-4$	$HiCuts-1$	ABV	$EGT-PC$	$HypCuts-4$	$HypCuts-1$
CR_1	85	12	25	32	111	32	14	14
CR_2	125	12	25	36	106	54	8	11
CR_3	351	12	35	57	126	47	22	31
CR_4	2,799	12	38	66	196	87	18	25

Figure 13: The total number of memory accesses for a worst case search in all 5 heuristics RFC, HiCuts($spfac = 1, 4$), BV, ABV, EGT-PC and HyperCuts for the four core router databases. One memory access is one word. One word is 32 bits.

As in HiCuts, there may be wasted space for cuts because of empty or replicated array elements. HyperCuts deals with replicated array elements by "pushing up common rules" to ancestors in the search tree. HyperCuts also reduces empty pointers using "region compaction" which shrinks a node region to includes only the areas covered by all rules in the node.

We evaluated HyperCuts on both industrial firewall databases and synthetically generated databases. We found very little improvement over HiCuts in the case of edge routers (because they have simple structure using rules on source-destination pairs). However, both HiCuts and HyperCuts do very well on such databases. Further, the packet classifiers from ISP_2 edge routers have recently migrated to a new set of rules, which use all five fields, and seem more similar to core router classifiers. Thus we believe that HyperCuts will outperform HiCuts significantly on the new edge router databases being deployed.

On firewall databases and core routers, HyperCuts produces an order of magnitude in memory utilization while at the same time reducing worst-case search time by a factor of up to 3. The difference between the performance on the three different types of databases underscores the need for more careful modeling of firewall databases.

Finally, we note that HyperCuts can easily be implemented in hardware at line speeds using a pipeline and on-chip SRAM. Since the trees generated by HyperCuts have heights no greater than 10, this requires only 10 pipeline stages, which is well within current hardware limits. For all these reasons, we believe that HyperCuts can be a viable algorithmic contender to Ternary CAMs.

8. ACKNOWLEDGEMENTS

The work of the first three authors was supported by NSF Grant ANI 0074004 and by a grant from NIST for the Sensilla Project.

9. REFERENCES

[1] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. Hot Interconnects*, 1999.

[2] ——, "Packet classification on multiple fields," in *SIGCOMM 99*, 1999.

[3] T. Lakshman and D. Stiliadis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," in *SIGCOMM*, 1998.

[4] F. Baboescu and G. Varghese, "Scalable packet classification," in *SIGCOMM*, 2001.

[5] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?" in *INFOCOM*, 2003.

[6] A. Feldman and S. Muthukrishnan, "Tradeoffs for packet classification," in *INFOCOM*, 2000.

[7] T. Woo, "A modular approach to packet classification: Algorithms and results," in *INFOCOM*, 2000.

[8] V. Srinivasan et al, "Fast and scalable layer 4 switching," in *SIGCOMM*, 1998.

[9] C. Matsumoto, "CAM vendors consider algorithmic alternatives," in *EETimes*, may 2002.

[10] M. Degermark et al, "Small forwarding tables for fast routing lookups," in *SIGCOMM*, 1997.

[11] M. Buddhikot et al, "Space decomposition techniques for fast layer-4 switching," in *Proc. PHSN*, 1999.

[12] V.Srinivasan, S.Suri, and G.Varghese, "Packet classification using tuple space search," in *SIGCOMM*, 1999.

[13] L. Qiu, G. Varghese, and S. Suri, "Fast firewall implementation for software and hardware based routers," in *Proc. ICNP*, 2001.

[14] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *UCSD Technical Report CS2003-0736*, 2003.

[15] S. Singh and F. Baboescu, "Packet classification repository." [Online]. Available: http://ial.ucsd.edu/ classification

Database	*No. of Rules*	$EGT-PC$	$HiCuts-4$	$HiCuts-1$	$HyperCuts-4$	$HyperCuts-1$
ER_1	4740	284,159	6695	6659	6659	6482
ER_2	2505	149,470	3730	3470	3456	3393
ER_3	995	62,266	1501	1459	1527	1465
ER_4	2458	154,976	3263	3274	3295	3295

Figure 14: The total memory space occupied by the search structure in EGT-PC, HiCuts($spfac = 1, 4$) and HyperCuts for the four edge router databases. The size is in memory words, one memory word is 32 bits.

Database	*No. of Rules*	$EGT-PC$	$HiCuts-4$	$HiCuts-1$	$HyperCuts-4$	$HyperCuts-1$
ER_1	4740	62	15	18	15	18
ER_2	2505	63	15	18	15	18
ER_3	995	49	15	18	15	15
ER_4	2458	65	15	15	11	15

Figure 15: The total number of memory accesses for a worst case search in EGT-PC, HiCuts($spfac = 1, 4$) and HyperCuts for the four edge router databases. One memory access is one word. One word is 32 bits.

Database	*No. of Rules*	$EGT-PC$	$HiCuts-4$	$HiCuts-1$	$HyperCuts-4$	$HyperCuts-1$
FW_1	279	7,477	48,347	16,978	9,574	6,026
FW_2	183	3,642	20,995	2,872	4,311	6,675
FW_3	158	2,962	18,207	6,675	2,164	943
FW_4	266	4,275	14,624	6,375	9,477	6,991

Figure 16: The total memory space occupied by the search structure in EGT-PC, HiCuts($spfac = 1, 4$) and HyperCuts for the four firewall databases. The size is in memory words, one memory word is 32 bits.

Database	*No. of Rules*	$EGT-PC$	$HiCuts-4$	$HiCuts-1$	$HyperCuts-4$	$HyperCuts-1$
FW_1	279	63	41	74	26	32
FW_2	183	55	74	74	20	26
FW_3	158	56	74	74	17	17
FW_4	266	38	23	50	17	23

Figure 17: The total number of memory accesses for a worst case search in EGT-PC, HiCuts($spfac = 1, 4$) and HyperCuts for the four firewall databases. One memory access is one word. One word is 32 bits.

Database	*No. of Rules*	*Memory Space*	*Search*
$ER-5K$	4,970	6,745	14
$ER-10K$	9,940	13,482	29
$ER-15K$	14,910	20,188	32
$ER-20K$	19,880	27,098	35
$CR-5K$	5,000	4,508	8
$CR-10K$	10,000	8,495	8
$CR-15K$	15,000	12,822	11
$CR-20K$	20,000	16,857	11
$FW-5K$	5,000	14,733	14
$FW-10K$	10,000	45,158	14
$FW-15K$	15,000	80,486	17
$FW-20K$	20,000	150,551	17

Figure 18: The total memory space and worst case search time for HyperCuts with a space factor of 4(optimized for speed) using synthetic databases. The databases are generated with the same prefix rule distribution as in the edge router (ER), core router(CR) and firewall(FW) databases. One memory access is one word. Memory space is expressed in words. One word is 32 bits.