Decision Tree based Network Packet Classification Algorithms

T S URMILA
Department of Computer Science, Sourashtra College, Madurai-4 Email: urmi_ts@yahoo.co.in
Dr R BALASUBRAMANIAN
Dean - MCA, KVCET, Mathuranthagam. Email: drrb_1951@gmail.com

---ABSTRACT---

The Internet is comprised of a mesh of routers interconnected by links. Communication among nodes on the Internet (routers and end-hosts) takes place using the Internet Protocol, commonly known as IP. IP datagrams (packets) travel over links from one router to the next on their way towards their final destination. Each router performs a forwarding decision on incoming packets to determine the packet's next-hop router. IP router forwards the packets and also chooses to perform special processing on incoming packets. Such special processing requires that the router classify incoming packets into one of several *flows which is called Packet Classification*. Packet Classification may be based on single Dimension or Multidimensional. The Packet classification is based on set of rules among which one of them is matched by the incoming packet and the corresponding action is taken. Decision Tree is one of the best datastrcture available to store and find the best matching rules. HiCut, HyperCut, Hypersplit, Layered cutting, DimCut, Efficut are some of the packet classification algorithms based on Decision tree Data structure.

Keywords – Packet Classification, HiCut, HyperCut, Hypersplit, Dimcut, Efficut.

---

1. INTRODUCTION

A packet-switch in a router must perform a forwarding decision on each arriving packet for deciding where to send it next. An IP router does this by looking up the packet's destination address in a forwarding table. This yields the address of the next-hop router and determines the outlet port through which the packet should be sent. This lookup operation is called a *route lookup* or an *address lookup* operation. Second, the packet-switch must transfer the packet from the way in to the way out port identified by the address lookup operation. This is called *switching*, and also involves physical movement of the bits carried by the packet.

1.1 Architecture of a packet-by-packet router

The Packet by packet router consists of one line card for each port and a switching fabric (such as a crossbar) that interconnects all the line cards. Typically, one of the line cards houses a processor functioning as the central controller for the router. The path taken by a packet through a packet-by-packet router consists of two main functions on the packet: (1) performing route lookup based on the packet's destination address to identify the outgoing port, and (2) switching the packet to the output port.

The routing processor in a router performs one or more routing protocols such as RIP by exchanging protocol messages with neighboring routers. This enables it to maintain a *routing table* that contains a representation of the network topology state information and stores the current information about the best known paths to destination networks. The router typically maintains a version of this routing table in all line cards so that lookups on incoming packets can be performed locally on each line card, without loading the central processor. This version of the central processor's routing table is referred o as the line card's *forwarding table* because it is directly used for packet forwarding.

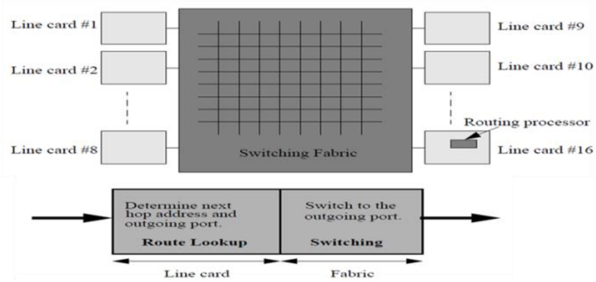


Figure 1 : Architecture and Data path of a high speed packet-by-packet Router

1.2 Flow-aware IP router and Packet Classification

One main reason for the existence of flow-aware routers stems from an ISP's desire to have the capability of providing differentiated services to its users. Traditionally, the Internet provides only a "best-effort" service, treating all packets going to the same destination identically, and servicing them in a first-come-first-served manner. In order to provide differentiated services, routers require additional mechanisms. These mechanisms — admission control, conditioning (metering, marking, shaping, and policing), resource reservation (optional), queue management and fair scheduling require, first of all, the capability to distinguish and isolate traffic belonging to different users based on service agreements negotiated between the ISP and its customer. This has led to service agreements, express them in terms of *rules* or *policies* configured on incoming packets, and isolate incoming traffic according to these rules. The collection of rules or policies is called a *policy database*, *flow classifier*, or simply a *classifier*. Each rule specifies a flow that a packet may belong to based on some criteria on the contents of the packet header. All packets belonging to the same flow are treated in a similar manner. The identified flow of an incoming packet specifies an *action* to be applied to the packet. For example, a firewall router may carry out the action of either *denying* or *allowing* access to a protected network. The determination of this action is called *packet classification* — the capability of routers to identify the action associated with the "best" rule an incoming packet matches. Packet classification allows ISPs to differentiate from their competition and gain additional revenue by providing different value-added services to different customers.

Flow-aware routers perform a superset of the functions of a packet-by-packet router. It consists of four main functions on the packet: (1) performing route lookup to identify the outgoing port, (2) performing classification to identify the flow to which an incoming packet belongs, (3) applying theaction (as part of the provisioning of differentiated services or some other form of special processing) based on the result of classification, and (4) switching to the output port.

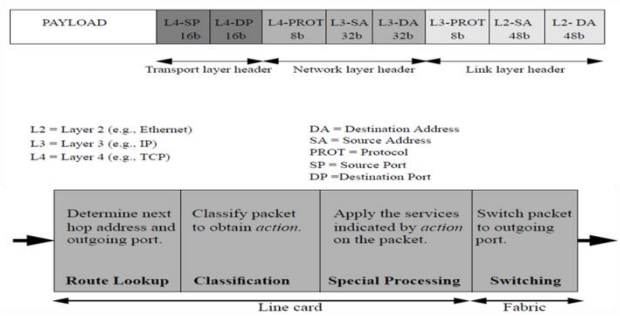


Figure 2: Header fields in a packet and Data path for the Flow aware Router

1.3 Packet Classification

A packet classifier must compare header fields of every incoming packet against a set of rules in order to assign a flow identifier. A rule must specify a set of headers and the policy to be in use. A Rule Space is a collection of rules, specified as a table (flat data base) with, Columns as RuleID, 'D' header field specification as f_1, f_2… f_d, and an Action column. Each record (Row) specifies a rule. Number D would be the Dimension of the rule space. Process of classification requires, applying the rules from the top. Attributes of the packet to be classified are matched with values in the Header columns, and if successful the Action becomes applicable. For example, a typical IP tables rule may specify: A INPUT –p tcp --dport 22 -j ACCEPT. This rule is to be interpreted as "a TCP packet is to be accepted if the destination port is equal to 22 ". If no match is found the next rule in the table is to be tested. In general the process continues sequentially till a match is found.

Mathematically, a packet P is said to *match* a particular rule R, if the *ith* field of the header of P satisfies the regular expression $R[i]$, for all $0 \le i < F$. If a packet P matches multiple rules, the matching rule with the highest priority is returned. The i^{th} component of rule R referred to as R[i], is a regular expression on the i^{th} field of the packet header. When viewed in this way two distinct rules are said to either practically overlapping or non-overlapping or that one is a subset of the other with corresponding set related definition. Given a classifier C with N rules, Rj, $1 \le j \le N$, where Rj consists of three entities:

i)Range expressions: R$j[i]$, $1 \le i \le d$, on each of the d header fields.
ii)Priority: pri(Rj), indicating the priority of the rule in the classifier. Commonly, 1st policy has the highest priority, *N*th policy (normally deny all) has the lowest.
iii)An action: referred to as action(Rj). In firewall, usually there is a default policy as the last policy that matches and denies all.

2. Decision Tree based Algorithms

Decision tree based packet classification algorithms focus on two aspects. The first one is how to select the cut dimension and the second is how to decide the cut-point for dividing address space into subspaces. There are two major methods to pick up the cut dimension: select a single cut dimension one at a time or select multiple cut dimensions at a time. When choosing a single cut dimension, the height of decision tree is usually higher than that by choosing more dimensions. But the node structure size is smaller because choosing multiple dimensions needs to keep more information.

There are two major methods to separate the filters, some algorithms use prefixed as the filter separating method and thus create equal-sized subspaces for dividing the rule table. In other words, they only need to store the "cut bits" in decision tree's internal nodes instead of the keys (or cutpoint). The other method is to divide the rule table by using cutting endpoints. Each rule in the filters generates a range (or interval) between two endpoints. Only endpoints of ranges are used as cut-points. Choosing end-points has more flexibility than choosing prefix.

2.1 Hicut, Hypercuts and Hypersplit

Hicuts and Hypercuts both employ equal-sized cuts. They use a heuristic to decide how many cuts should be employed. The most important difference between Hicuts and Hypercuts is that Hicuts only cuts one dimension in an internal node but Hypercuts cuts multiple dimensions. Therefore, Hypercuts' tree depth is shorter than Hicuts. Hypersplit only cuts a single dimension in an internal node, but it employs end-point to find out the cut-point. First, for each interval, Hypersplit calculates the number of rules that cover the interval and store it in Sr[j] for$1 \le j \le M$, where M is the number of end-points. Then it chooses the smallest endpoint *m* such that [ ] , which is called *heuristic weighted segment balanced strategy*. This strategy tries to make the sum of covering rules of all the intervals at the left side and right side of the end-point *m* equal. Hypersplit only separates subspaces into two parts. Furthermore, Hypersplit only picks up one dimension to cut, so the Hypersplit decision tree is a binary tree.

Table 1 is a 2-D rule table. There are 5 rules and R1's priority is highest. Fig 3 show the decision trees built by Hicuts, Hypercuts, Hypersplit. In Figure 3, Hicuts employs the equal-sized subspace partition, and chooses only one dimension to cut for every internal node. Because only one dimension is selected at a time, the tree height of final decision tree is highest among all the schemes. Higher tree generates more internal nodes, and the memory storage become large. In Figure 3, Hypercuts also employs the equal-sized subspace partition, but it chooses multiple dimensions at each internal node. So, the height of decision tree decreases dramatically. But, there is a critical drawback that some rules are duplicated many times. For example, R2 exists in 4 leaf nodes. It wastes lots of memory to store those duplicated rules. In Figure 3, Hypersplit chooses cut-points. At the first level, the rules in each of three intervals at field-x are 2, 1, and 2. So, value 10 is used as the cut-point which divides the rule table into two groups, {R1, R2, R3} and {R4, R5}. At level 2, left internal node's rules in each of two intervals at field-x are 3 and 1. So, selecting 01 as cut-point can divide rules into {R1, R2} and {R3}. The right internal node's rules in each of three intervals at field-y are 1, 1, ad 2. So, choosing 11 as cut-point can divide rules into {R4} and {R5}. By this rules it could completes the decision tree. This cut-point selection algorithm of Hypersplit reduces the rule duplications effectively.

Rule	Field-X	Field-Y
R1	[00,00]	[00,00]
R2	[00,00]	[00,11]
R3	[01,01]	[00,11]
R4	[10,11]	[11,11]
R5	[10,11]	[00,11]

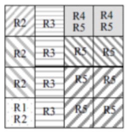
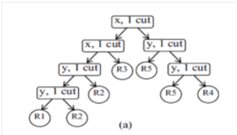
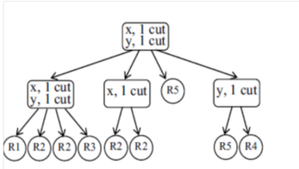


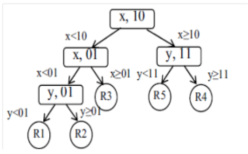
Table 1 : 2D Rule Table



Hicut tree for rule table



Hypercut tree for rule table



Hypersplit tree for rule table

Figure 3: Hicut,Hypercut and Hypersplit trees for rule table

2.2 Layered Cutting Scheme

In a Layered cutting Scheme for a packet classification algorithm picks up multiple dimensions and cutting with end-point to make the height of decision tree much shorter. Then a layered mechanism is proposed to reduce the memory consumption dramatically. The algorithm focuses on two aspects. The first aspect is to pick up the dimensions and the second aspect is to decide the cut-point.

A. Select the cut dimensions:

The Cut dimensions are chosen based on The set of dimensions with Larger Distinct field values, the dimensions with value smaller than the average value of all dimension and dimensions whose number of end-points is greater than average number of endpoints of all dimensions.

B. Space decomposition:

For Space decomposition Weighted Segment balanced scheme and ½ end point schemes are choosen. In ½ end point scheme the cut-point m is selected such that the number of intervals at m's leftside is equal to that of m's right side .i.e . 1/2(lowbound endpoint + upbound end-point)

In the Layered cutting scheme for selecting the cut dimension, distinct field values heuristic, and for select cut-point weighted segment-balanced heuristic is chosen to obtain the best results of memory consumption and number of memory accesses.

Optimization

Rule duplication is a very serious problem in packet classification. It will cause a rule replicated many times and use a lot of memory to keep them.

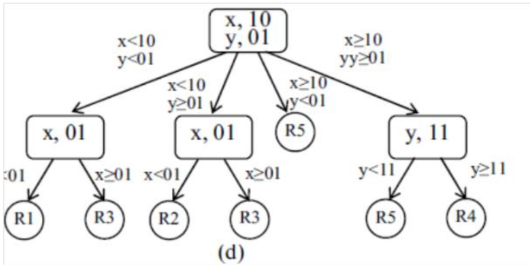


Figure 4 : Layered Cutting Scheme

Figure 5(a) shows rule duplications in a decision tree. R1 exists in node 6 and node 7 and as a result, both the left child of node 6 and node 7 need to store R1. In the same way, R4 exists in node 8 and node 9, and both the right child of node 8 and left child node of 9 need to store the R4. This situation causes a lot of redundant rules. So, we must keep cutting the tree until the number of rules in the node is less than the bucket size. Rule duplication not only increases the memory storage but also increases the tree depth. Hypercuts proposes a solution to tackle this problem, named "*Pushing Common Rule Subsets Upwards*". If all children have the same rules, then the parent node will create a rule list (i.e., bucket) to store this rules instead of duplicating them in its children. Figure 5(b) shows the solution by Hypercuts. R1 is stored in the rule list of node 2 and R4 is stored in the rule list of node 4. When traversing to node 2 and node 4, the rules lists belonging the internal nodes must also be searched. Layered cutting scheme algorithm tackles those duplicated rules by removing those duplicated rules, and uses them to create a duplicated rule table. Figure 5(c) shows how we decrease the tree depth and the number of node. In our algorithm, during constructing our decision tree, if we find a rule could be moved out, then when we traverse to another node which has the same rule, this rule should be eliminated. That ensures the rule not existing in this decision tree and eliminates the replication condition effectively. Then, according to heuristic, another decision tree is constructed from the duplicated rule table. When during search, all the decision trees have to be searched. Partial redundancy can't be pushed up which causes the rule still being duplicated many times. In Figure 5(a), nodes 2, 3, 4 have the same rule R1, but the node 5 doesn't have it. So, R1 can not be pushed up to node 1. Although R1 can be pulled up to node 2, but node 3 and node 4 also need to keep R1 in their child nodes. The pushing up heuristic can be regarded as local operation that the different sub-trees pushing operation is independent. So rule duplication condition still exists. The data structure totally needs 112 bits for each internal node and leaf node. For internal node, 1 bit is needed to identify whether the node is an internal node or a leaf and 5 bits are needed to identify which cut dimensions are selected. The dimensions are constrained only up to 3 and so need 80 bits to store the three cut-points, e.g., 32 bits, 32 bits, and 16 bits for two IP address fields and one port field. Also, need 26 bits to store the address of leaf nodes. Because the sibling nodes are located in to continuous address, so store only the address for first child node and accesses the others by offset. For leaf nodes, the largest rule table we test is 10K, so need 14 bits to discriminate rules, and the bucket size is 8, so all need is 14*8=112 bits for each leaf .

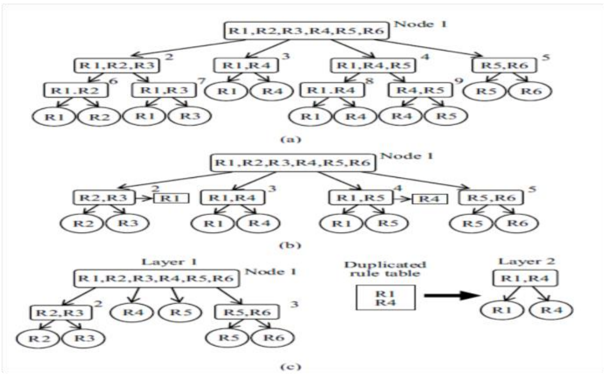


Figure 5: An optimized tree for Layered cutting scheme

2.3 DimCut Algorithm

DimCut algorithm adds some modifications and improvements on the HiCuts algorithm. Consider the following definitions:

Definition: Let wc(H) be the count of wild card entries in the column H in the whole of the rule set.

Definition: Let gd(H) be the geometric distance associated with column H in the whole of the rule set.

Some of the guidelines and principles followed in this algorithm are: i. Dimension Selection: Select the two fields Ha, Hb which have the least wc( ) values, as the two selected dimensions or alternatively select Ha, Hb which have least gd( ) values. ii. Number of cuts and Bucket size: Compute the number of cuts as the number of cuts, and the bucket size threshold as: NC1 = [20 + (N/1000)] = Number of cuts, B = [N/ (20 + (N/1000))] = Bucket size (The threshold). Here N = Total Number of rules, in the complete rule set, iii. Separate those rules in the same chosen field as cut dimension which have wildcard value and shift them to the bucket and reject their use for making the decision tree. iv. Building index tables to facilitate search within: Build an index table for each bucket. v. The number of cuts has to be decided at the first time cutting and also the bucket size threshold of the algorithm has to be identified, with the purpose of trying to avoid splitting of rules while cutting. vi. Recursion: The best dimension for next cut level is identified after the first cut, again using the same principles. vii. New algorithm has two separate levels, preprocessing level (tree construction and making index table) and search level. viii. Use the Link list data structure at the input stage and work on large rule sets.

Optimize the decision tree: The decision tree has been constructed by DimCut algorithm. Eliminating the empty nodes, merging the nodes that are associated with the same set of rules, in case the region covered by the rules, is smaller than the overall size of the region governing the node, one shrinks the region associated with the node to minimum cover, and if the same rule repeated in all nodes in the same level, then separate that rule and make a bucket of that for use at the time of search. Set the default action for those entry packets that do not match with any bucket. All the rules in all the buckets should be sorted by priority.

Index table making: The field that is chosen for cut dimension in each bucket will make an index table. The framework will contain two stages: an index table and rule buckets. Use the same field of the input packet to search in the index table. If the specific field matches, the matching filter will be selected out of the set in the bucket via linear search (using smaller set of rules). All incoming packets need to check at the fields selected during preprocessing. The decision tree traverses to find the buckets that cover the incoming packet. There is priority sorting of all rules. When first match index is found a packet will traverse all regions of possible belonging. The packet will check the all header fields of governing rules linearly. The most prioritized packet is picked up via those that match completely. So the final action (Accept/Deny) will be taken for that incoming packet and the search will end. It supports incremental update but in case of significant decreasing performance it needs reconstruction. Updating will work in the same manner as the search algorithm. For firewalls a very slow update rate would suffice and entries can be added manually or infrequently.

The Briefed Preprocessing Algorithm:

Read rules and create a link list to store them, ii. Find the cut dimension by using any of 2 heuristics (any dimension that has the smallest geometric length/ any dimension that has the smallest number of wildcards), iii. Calculate the number of cuts by using of (NC= [20+(Number of rules/1000)]) and Calculate the Threshold T= [(Number of rules)/NC], iv. Separate those rules that has wildcard value in the same chosen field as cut dimension in the bucket, v. Construct the tree, For i=1 to NC do, Create buckets (nodes), Assign the rules that covered by buckets (nodes) region , If the number of rules in bucket > Threshold, Split buckets(nodes), Create the index table for rules in buckets, Optimize and compress the tree, END.

The Briefed Search Algorithm: i. Use Search part: Read Packets, For each Packet: Find the buckets that cover the packet, Search in the related index table of those buckets, Find the specific matched rules, Select the higher priority one as a target, Act as its action, iii. End

Priority	Source Add.	Destination Add.
R1	1010	*
R2	1100	1010
R3	0101	1001
R4	*	10*
R5	111*	11*
R6	001*	1100
R7	*	00*
R8	0*	10*
R9	0110	011*
R10	1*	11*

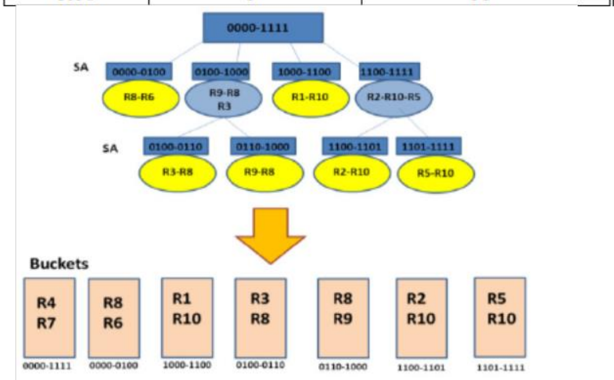


Figure 7 : Dimcut tree with Optimization

2.4 EFFICUTS

EffiCuts Algorithm implements the new ideas of *separable trees* combined with *selective tree merging* to tackle the variation in the size of overlapping rules and *equi-dense cuts* to tackle the variation in the rule-space density. EffiCuts also leverages equi-dense cuts to achieve fewer accesses per node than HiCuts and HyperCuts by *co-locating* parts of information in a node and its children.

Separable Trees

Placing small and large rules in different trees would reduce the replication. Large rules are identified easily as those that have wildcards in many fields. There is a possibility of two trees — one for rules with many wildcard fields and the other for the rest. Another factor called *separability*, is more fundamental than rule size, which determines the extent of replication. While the above scheme ignores the rule space's dimensions, separability considers variability of rule size in each dimension. Separability enables the solution to avoid assigning and optimizing arbitrary percentages of the rules to distinct trees. To eliminate overlap among small and large rules, *all* small and large rules are separated by defining a subset of rules as *separable* if *all* the rules in the subset are either small or large in *each* dimension. A distinct tree is built for each such subset where each dimension can be cut coarsely to separate the large rules, or finely to separate the small rules without incurring replication.

Identifying Separable Rules

Separability implies that *all the rules in a tree are either wildcard or non-wildcard in each field;* otherwise, cuts separating the non-wildcard rules would replicate the wildcard rules. The categories assuming the standard, five-dimensional IPv4 classifier are:

• *Category 1*: rules with four wildcards
• *Category 2*: rules with three wildcards
• *Category 3*: rules with two wildcards
• *Category 4*: rules with one or no wildcards

To capture separability, each category is broken into sub-categories where the wildcard rules and non-wildcard rules are put in different sub-categories on a per-field basis. Accordingly, *Category 1* has a sub-category for each non-wildcard field, for a total of C1 = 5 sub-categories. *Category 2* has a sub-category for each pair of non-wildcard fields for a total of C2 = 10 sub-categories. *Category 3* has a sub-category for each triplet of non-wildcard fields for a total of C3 = 10 sub-categories. Because *Category 4* contains mostly small rules, the further sub-categories are unnecessary.

Selective Tree Merging

Selective tree merging, which merges two separable trees mixing rules that may be small or large in at most one dimension. For instance, a *Category 1* tree that contains rules with non-wildcards in field *A* (and wildcards in the other fields) is merged with *Category 2* tree that contains rules with non-wildcards in fields *A* and *B*, and wildcards in the rest of the fields. This choice ensures that wildcards (of *Category 1*) are merged with non-wildcards (of *Category 2*) in only field *B*; in each of the rest of the fields, either non-wildcards are merged with non-wildcards (field *A*) or wildcards with wildcards (the rest). This significantly reduces the number of lookups while incurring only modest rule replication. One exception is the single *Category 4* tree which is not broken into sub-categories, and hence, already mixes wildcard and non-wildcards in multiple fields. As such, merging this tree with other *Category 3* trees would cause such mixing in additional fields and would lead to significant rule replication. Therefore, do not merge the *Category 4* tree with any other tree.

In EffiCuts, copy of rules, instead of a pointer to, each rule at the leaf, forcing the rules to be in contiguous memory locations. However, if a rule is not replicated then this strategy requires less memory as it stores only the rule, and not a pointer and the rule. Because EffiCuts' rule replication is minimal, these two effects nearly cancel each other resulting in little extra memory.

Equi-dense Cuts

Recall that HyperCuts' equi-sized cuts, which are powers of two in number, simplify identification of the matching child but result in redundancy due to rule-space density variation. Fine cuts to separate densely-clustered rules needlessly partition the sparse parts of the rule space resulting in many ineffectual tree nodes that separate only a few rules but incur considerable memory overhead. This redundancy primarily adds ineffectual nodes and also causes some rule replication among the ineffectual nodes. The child-pointer redundancy enlarges the node's child-pointer array which contributes about 30-50% of the total memory for the tree. Consequently, reducing this redundancy significantly reduces the total memory. Similarly, the partial redundancy in siblings' rules manifests as rule replication which is rampant in HyperCuts even after employing node merging and moving up. To tackle both the child-pointer redundancy and partial redundancy in siblings' rules, we propose *equi-dense cuts* which are unequal cuts that distribute a node's rules as evenly among the children as possible. Equi-dense cuts achieve fine cuts in the dense parts of the rule space and coarse cuts in the sparse parts. Unequal cuts are constructed by fusing unequal numbers of HyperCuts' equi-sized cuts. By fusing redundant equi-sized cuts, our unequal cuts (1) merge redundant child pointers at the parent node into one pointer and (2) remove replicas of rules in the fused siblings.

Fusion Heuristics

For the fusion of equi-sized cuts to produce unequal cuts, the simple and conservative heuristic is to fuse contiguous sibling leaves (i.e., corresponding to contiguous values of the bits used in the cut) if the resulting node remains a leaf (i.e., has fewer than *binth* rules). This fusion does not affect the tree depth but reduces the number of nodes in the tree and reduces rule replication among siblings. This heuristic serves to remove fine cuts in sparse regions along with the accompanying rule replication. To capture rule replication in denser regions, the moderate heuristic fuses contiguous, non-leaf siblings if the resulting node has fewer rules than (1) the sum of the rules in the original nodes, and (2) the maximum number of rules among all the siblings of the original nodes (i.e., including those siblings that are not being fused). The first constraint ensures that the original nodes share some rules so that the heuristic reduces this redundancy. The second constraint decreases the chance of the tree becoming deeper due to the fusion. However, there is no guarantee on the tree depth because the resultant node could have a different set of rules than the original nodes which may lead to a deeper tree. The aggressive heuristic is to fuse non-leaf nodes as long as the resulting node does not exceed some percentage (e.g., 40%) of the number of rules in the sibling with the maximum number of rules. This heuristic always reduces the number of children and thereby shrinks the child-pointer array.

Lookup by Packets

Because equi-dense cuts are unequal, identifying the matching child at a tree node is more involved than simple indexing into an array. Equi-sized cuts, which are powers of two in number, result in a one-to-one, ordered correspondence between the index values generated from the bits of the appropriate field(s) of the packet and the entries in the child-pointer array at each node. This correspondence enables simple indexing into the array. In contrast, unequal cuts destroy this correspondence by fusing multiple equi-sized cuts into one equi-dense cut, causing multiple indices to map to the same array entry. Consequently, simple indexing would not work and an incoming packet needs to compare against all the array entries to find the matching child. To control the complexity of the comparison hardware, that the number of unequal cuts per node is constrained, and hence the number of comparators needed, not to exceed a threshold, called *max_cuts*. For nodes that need more cuts, the algorithm fall back on equi-sized cuts, as in HiCuts and HyperCuts, with the accompanying redundancy. One bit per node is used to indicate whether the node uses equi-sized or equi-dense cuts. Each node using equi-dense cuts stores the number of unequal cuts and an array of the starting indices of the cuts.

Node Co-location

In EffiCuts' nodes using equidense cuts, the first part additionally holds the table of starting indices of each cut . A packet has to look up the cut dimension and the number of cuts in each node's first part to determine its index nto the array in the second part, and then retrieve the child node pointer at the index. Consequently, each node requires at least two memory accesses. To enable each node to require only one access and thereby achieve better memory bandwidth, a node's child-pointer array is co-located in contiguous memory locations (the second part) with *all* the children's headers (their first parts). This co-location converts the array of pointers into an array of headers and pointers to the children's arrays (rather than pointers to the child nodes themselves). Accessing each such collocated node retrieves the header of the indexed child node in addition to a pointer to the child node's array (assuming the memory is wide enough), thereby combining the node's second access with the child node's first access. Thus, each node requires only one reasonably-wide access. (While narrower memories would require more than one access, the co-location would still reduce the number of accesses by one.)

With the co-location, the array now holds the children's headers (and the pointers to the children's arrays). The headers must be unique for each child node in order for the index calculated from the parent node's header to work correctly. Consequently, the headers for identical children have to be replicated in the array, incurring some extra memory (though identical children may still share a single child node's array). Fortunately, the redundancy is minimal for EffiCuts' equi-dense cuts where the nodes are forced to have only a few children which are usually distinct (*max_cuts* is 8), making it worthwhile to trade-off small amounts of memory for significant bandwidth demand reduction. To reduce further the number of memory accesses per node, HyperCuts' rule moving-up optimization in EffiCuts is eliminated because each moved-up rule requires two accesses: one for the pointer to the rule and the other for the rule itself whereas a rule that is not moved-up in EffiCuts would fall in a leaf where the rule may contribute only a part of a wide access. Rule moving-up reduces HyperCuts' rule replication, which is minimal for EffiCuts, and therefore, the elimination makes sense. EffiCuts facilitates incremental updates in at least two ways. First, because separable trees drastically reduce replication, updates are unlikely to involve replication, and hence do not require many changes to the tree. Second, equi-dense cuts afford new flexibility that does not exist in HyperCuts. If a new rule falls in an already-full leaf (i.e., a leaf with *binth* rules) then equi-dense cuts provide two options: (1) the existing cuts can be nudged to create room for the new rule by moving some of the rules from the already-full leaf to a not-full sibling; or (2) if the leaf's parent has fewer cuts than *max_cuts*, then a cut can be added to accommodate the new rule.

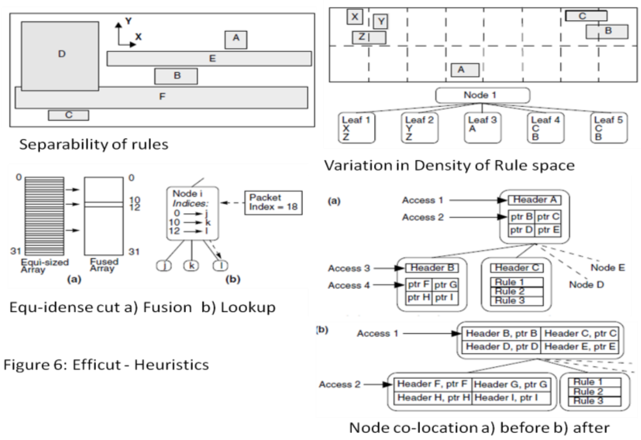


Figure 6: Efficut - Heuristics

CONCLUSIONS

The Hicut, Hypercut, Hypersplit algorithms are the early developed algorithms among the Decision tree based packet classification algorithms. They have their own advantages and disadvantages. The DImcut, Layered Cutting scheme and Efficut algorithms are improved from the Hicut and Hypercut algorithms by minizing the memory requirements and access time for the Firewall databases and access control lists.

REFERENCES

1. Algorithms for routing lookups and Packet classification Pankaj Gupta December 2000

2. Brodnik, S. Carlsson, M. Degermark and S. Pink, ―Small Forwarding Tables for Fast Routing Lookups.‖ *Proc. ACM SIGCOMM 1997*, pp. 3-14, Cannes, France

3. Yaxuan Qi and Jun Li "Packet Classification with Network Traffic Statistics"

4. Pankaj Gupta and Nick McKeown, ― Packet Classification using Hierarchical Intelligent Cuttings‖

5. Hediyeh AmirJahanshahi Sistani, Haridas Acharya ―Comparative evaluation of Recursive Dimensional Cutting Packet Classification, DimCut, with Analysis‖ International Journal of Computer Science & Engineering Technology (IJCSET)

6. Sumeet Singh, Florin Baboescu, George Varghese, Jia Wang ―Packet Classification Using Multidimensional Cutting‖
7. Bo Xu, Dongyi Jiang, ―HSM: A Fast Packet Classification Algorithm‖
8. Mrudul Dixit, Anuja Kale, Madhavi Narote, Sneha Talwalkar, and B. V. Barbadekar ―Fast Packet Classification Algorithms‖ International Journal of Computer Theory and Engineering, Vol. 4, No. 6, December 2012
9. Safaa O. Al-Mamory Wesam S. Bhaya, Anees M. Hadi ―Taxonomy of Packet Classification Algorithms‖ Journal of Babylon University/Pure and Applied Sciences/ No.(7)/ Vol.(21): 2013
10. Hediyeh Amir Jahanshahi Sistani1, Sayyed Mehdi Poustchi Amin1 and Haridas Acharya2 Packet Classification Algorithm Based on Geometric Tree by using Recursive Dimensional Cutting(DimCut) Research Journal of Recent Sciences ISSN 2277-2502 Vol. **2(8),** 31-39, August **(2013)** nternational Science Congress Association
11. Balajee Vamanan*, Gwendolyn Voskuilen* and T. N. Vijaykumar EffiCuts: Optimizing Packet Classification for Memory and Throughput, 2011 International Conference on Advanced Information Networking and Applications
12. Yeim-Kuan Chang and Han-Chen Chen Layered Cutting Scheme for Packet Classification, *SIGCOMM 2010*, August 30-September 3, 2010, New Delhi, India.