EffiCuts: Optimizing Packet Classification for Memory and Throughput

Balajee Vamanan*, Gwendolyn Voskuilen* and T. N. Vijaykumar (* both primary student authors)
School of Electrical and Computer Engineering, Purdue University
{bvamanan, geinfeld, vijay}@ecn.purdue.edu

ABSTRACT

Packet Classification is a key functionality provided by modern routers. Previous decision-tree algorithms, HiCuts and HyperCuts, cut the multi-dimensional rule space to separate a classifier's rules. Despite their optimizations, the algorithms incur considerable memory overhead due to two issues: (1) Many rules in a classifier overlap and the overlapping rules vary vastly in size, causing the algorithms' fine cuts for separating the small rules to replicate the large rules. (2) Because a classifier's rule-space density varies significantly, the algorithms' equi-sized cuts for separating the dense parts needlessly partition the sparse parts, resulting in many ineffectual nodes that hold only a few rules. We propose *EffiCuts* which employs four novel ideas: (1) *Separable trees*: To eliminate overlap among small and large rules, we separate *all* small and large rules. We define a subset of rules to be *separable* if *all* the rules are either small or large in *each* dimension. We build a distinct tree 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. (2) *Selective tree merging*: To reduce the multiple trees' extra accesses which degrade throughput, we selectively merge separable trees mixing rules that may be small or large in at most one dimension. (3) *Equi-dense cuts*: We employ unequal cuts which distribute a node's rules evenly among the children, avoiding ineffectual nodes at the cost of a small processing overhead in the tree traversal. (4) *Node Co-location:* To achieve fewer accesses per node than HiCuts and HyperCuts, we co-locate parts of a node and its children. Using ClassBench, we show that for similar throughput EffiCuts needs factors of 57 less memory than HyperCuts and of 4-8 less power than TCAM.

Categories and Subject Descriptors:

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

General Terms:

Algorithms, Design, Performance

Keywords:

Packet Classification, Decision-Tree Algorithm, Rule Replication

1 INTRODUCTION

Packet classification is the problem of determining the highest-priority rule out of a set of rules to which each network packet matches, where each rule specifies a desired action on a set of packets identified by a combination of the packet fields (e.g., source/destination IP, source/destination port, and protocol). Packet classification continues to be an important functionality provided by routers for various applications in QoS, security, and network traffic monitoring and analysis. The classifiers are growing in size due to (1) the widespread deployment of virtual private networks (VPNs) which customize rules for each VPN, (2) the need for finer-grained differentiation demanded by QoS and other requirements, and (3) the ever-increasing number of hosts as anticipated by IPv6. Combined with the ever-increasing line rates due to advances in fiber optics, these trends imply that routers need to look up larger classifiers at higher throughputs (e.g., several tens of thousands of rules every few nanoseconds).

Packet classification is an old problem with many proposed solutions [8, 2, 5, 14, 6]. As with many of the functionalities provided by modern routers (e.g., IP lookup), packet classification schemes must scale well in throughput, power, and memory size [4]. TCAM-based solutions [13] provide deterministic performance, but do not scale well in throughput and power with classifier size. In contrast, RAM-based algorithmic solutions have the potential to scale well in throughput and power. However, the algorithms to date incur either long searches (e.g., [14,17]) or large memories (e.g., [6,12]), falling short of delivering on the potential. The key contribution of this paper is reducing the memory required by decision-tree based algorithms, specifically HiCuts [6] and its successor, HyperCuts [12], by orders of magnitude while maintaining high throughput. We achieve this memory reduction by virtually eliminating rule replication incurred by HiCuts and HyperCuts — from an average factor of a few hundreds to less than 1.5.

HiCuts introduced the idea of building off-line a decision tree by cutting the multi-dimensional rule space (e.g., IPv4's five-tuple space) into cubes, one dimension at a time, and successively refining the cuts to separate the rules into different subtrees, and eventually, distinct leaves. Each leaf holds at most as many rules as can be searched linearly without much loss of throughput (e.g., 16 rules). Network packets traverse the tree to determine the highest-priority matching rule. Two key metrics are the memory size and number of memory accesses. Because routers overlap multiple packets to hide memory latencies, packet throughput is fundamentally limited by the memory bandwidth demand, and hence the number of accesses. HyperCuts improves upon HiCuts in both the metrics.

Despite its optimizations, HyperCuts incurs considerable memory overhead for large classifiers. We make two key observations about this overhead: (1) Variation in rule size: Many rules in a classifier overlap and the overlapping rules vary vastly in size. The algorithms build a single tree with overlapping small and large rules. Consequently, fine cuts to separate the smaller rules need-

SIGCOMM 2010, August 30-September 3, 2010, New Delhi, India.
Copyright 2010 ACM 978-1-4503-0201-2/10/08...$10.00.

lessly replicate the larger rules that overlap with the smaller rules without achieving much rule separation while incurring considerable replication and memory overhead (e.g., in 100,000-rule classifiers, HyperCuts replicates rules by factors of 2,000-10,000 on average). (2) Variation in rule-space density: A classifier's rule-space density varies significantly. At each tree node, the algorithms use equi-sized cuts, which are powers of two in number, to identify the matching child via simple indexing into an array. However, fine cuts to separate densely-clustered rules needlessly partition the sparse parts of the rule space, creating many ineffectual tree nodes.

To reduce the above overhead while maintaining high throughput, we propose *EffiCuts* which employs four novel ideas, two for each of the above two issues of variations in rule size and rule-space density. To tackle the variation in the size of overlapping rules, we eliminate overlap among small and large rules by separating *all* small and large rules. We define a subset of rules to be *separable* if *all* the rules in the subset are either small or large in *each* dimension. For instance, a set of rules with wildcards *only* in two specific fields is separable (assuming wildcards imply large rules). Accordingly, we place each separable subset in a distinct *separable tree*, our first idea, where each dimension can be cut coarsely to separate the large rules, or finely to separate the small rules without incurring replication. While Modular Classification [17] proposes using multiple trees, our novelty is in binning the rules into different trees based on separability whereas Modular Classification uses some selected prefixes for this binning. We show that separability is key to reducing rule replication.

While separable trees drastically reduce replication, each packet has to be looked up in all the trees, requiring more memory accesses and exerting higher bandwidth demand than in HiCuts' or HyperCuts' single tree. Thus, in the trade-off between memory size and bandwidth, HyperCuts trades size for bandwidth whereas EffiCuts does the opposite. To reduce EffiCuts' number of accesses, we observe that merging two separable trees results in a tree whose depth is usually less than the sum of the original trees' depths. However, merging arbitrary trees may completely destroy separability and may result in significant rule replication. Therefore, we employ *selective tree merging*, our second idea, which merges separable trees mixing rules that may be small or large in at most one dimension. This compromise significantly reduces the number of accesses and incurs only modest rule replication, achieving a 'sweet spot' in the memory size-bandwidth trade-off.

To tackle rule-space density variation, we employ *equi-dense cuts*, our third idea, which are unequal cuts that distribute a node's rules as evenly among the children as possible. Equi-dense cuts employ fine cuts in the dense parts of the rule space and coarse cuts in the sparse parts, avoiding HyperCuts' ineffectual tree nodes that hold only a few rules. Because equi-dense cuts are unequal, identifying the matching child at a tree node is more involved than simple indexing into an array, and requires comparisons of the packet against a set of values. Thus, our equi-dense cuts trade-off node processing complexity for rule replication and memory overhead. We ensure an acceptable trade-off by constraining the number of children and hence comparisons needed (e.g., 8), and falling back on HyperCuts' equi-sized cuts (with the accompanying rule replication) for nodes that require more comparisons.

Equi-dense cuts provide another benefit. Each HyperCuts node requires at least two memory accesses due to a dependence between two parts of the node (i.e., the first part determines the second part's address); narrow memories would incur more accesses. We co-locate in contiguous memory locations a node's second part with *all* its children's first parts. This *node co-location*, our fourth idea, allows each node to require only one (reasonably-wide) access, achieving better throughput. The co-location precludes HiCuts' optimization of merging identical sibling nodes to reduce memory, and therefore incurs some extra memory. The redundancy is minimal for equi-dense cuts where the nodes are forced to have only a few children which are usually distinct.

Using ClassBench [16], we show that for classifiers with 1,000 to 100,000 rules (1) EffiCuts drastically reduces the worst-case rule replication to less than a factor of nine as compared to HyperCuts' factor of several thousands; and (2) For similar throughput, EffiCuts needs a factor of 57 less memory than HyperCuts and a factor of 4-8 less power than TCAM.

The rest of the paper is organized as follows: Section 2 provides some background on HiCuts and HyperCuts. We describe the details of EffiCuts in Section 3. We present our experimental methodology in Section 4 and our results in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND

Because EffiCuts builds on HiCuts and its successor HyperCuts, we present some background details on these two algorithms before describing the details of EffiCuts.

2.1 HiCuts

The HiCuts algorithm partitions the multi-dimensional rule space (e.g., source IP, destination IP, source port, destination port, protocol) through a decision tree with the goal of evenly separating the rules into the tree's leaves. Starting at the root which covers the entire rule space, HiCuts cuts the space in a single dimension to create a set of equi-sized subspaces which separate the rules as evenly as possible. Each subspace is represented by a child node. Figure 1 shows an example of a two-dimensional rule space. The figure shows five rules and the corresponding decision tree. If a rule spans multiple subspaces then the rule is replicated in each of the corresponding children (e.g., *R5* is replicated in *Leaf 1* and *Leaf 2* in Figure 1). HiCuts repeats this process at each child node until the number of rules at the node is below a threshold called *binth* (e.g., 2 in Figure 1), keeping the node as a leaf. Each leaf holds a set of pointers to its rules. Incoming packets traverse the tree until they reach a leaf node. The rules at the leaf are searched linearly to determine the highest-priority matching rule.

HiCuts adapts the partitioning process by cutting each node independently of another, employing fine (coarse) cuts for dense (sparse) subspaces, even if the nodes are siblings. A key implication of this adaptation is that for non-overlapping small and large rules, HiCuts can employ appropriate fine or coarse cuts without incurring replication. However, for overlapping small and large rules, HiCuts is forced to separate the small rules by using fine cuts which replicate the large rules. Rule replication artificially increases the rule count so that *each* replica eventually falls in a leaf which holds a pointer to the replicated rule. It is due to these pointers that rule replication increases memory.

HiCuts employs four heuristics to optimize the tree. The first two select the appropriate dimension to cut and number of cuts to make at each node. The last two eliminate redundancy in the tree to reduce the storage requirement. The first heuristic selects the dimension to be cut at each node. HiCuts observes that because the

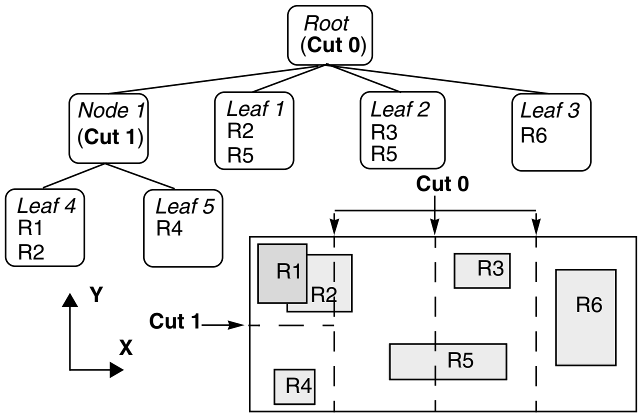

FIGURE 1. HiCuts Example in a 2D Rule Space

tree depth, and hence the number of memory accesses, is affected by the child with the most rules, the selected dimension should minimize the maximum number of rules per child resulting from the cut. To this end, the heuristic selects the dimension where the rules are most spread out (i.e., considering rules as multi-dimensional cubes, the selected dimension has the most unique projections of the rules). For the second heuristic, HiCuts observes that a larger number of cuts at a node partitions the space more finely and reduces the tree depth, but may increase rule replication and also the number of children, some of which may not achieve good rule separation. That is, some children may cover fewer than *binth* rules while others may cover more, so that the node storage for the former children contributes to storage overhead. The second heuristic attempts to maximize the number of cuts, and hence minimize the depth, while limiting the total number of rules at all the children of a node to be within a factor, called *space_factor*, of the number of rules at the node. The third heuristic targets redundant sibling nodes which share an identical set of rules. The heuristic simply merges such siblings into one node. A second form of redundancy exists when a higher-priority rule completely overlaps a lower-priority rule within a node's subspace. In this case, no packet would ever match the lower-priority rule which can be removed, as done by the fourth heuristic.

HiCuts implements each tree node as a structure containing a header identifying the dimension that is cut at the node and the number of cuts, and an array of pointers to the node's children (as many as the cuts). The tree is laid out as an array of nodes. In the tree traversal, a packet looks up a node and uses the specified number of bits in the specified dimension to index into the node's child-pointer array which returns a pointer to the matching child node. Because HiCuts employs equi-sized cuts, which are powers of two in number, identifying the matching child amounts to simple indexing into the child-pointer array. This point is relevant in Section 3.3 where we contrast equi-sized cuts against our equi-dense cuts. The traversal proceeds from one node to the next terminating in a leaf which returns a list of pointers to the rules. A linear search of the rules returns the highest-priority rule matching the packet.

To contrast later HiCuts' node-merging heuristic against EffiCuts' equi-dense cuts, we discuss a detail about the heuristic. The heuristic merges identical siblings into one node by simply forcing the corresponding child pointers in the parent node to point to just one child node. However, the pointers themselves are redundant.

Despite its optimizations, HiCuts suffers from two primary drawbacks. First, the tree depth is dependent on the distribution of the rules in the rule space. Classifiers that can be separated by partitioning a single dimension achieve shallow trees, while the others require deep trees even if the number of cuts at each node is not large (i.e., increasing the cuts at each node would increase rule replication without improving rule separation). This limitation arises from the fact that HiCuts considers only one dimension to cut at a node. Second, a large amount of redundancy still remains. HiCuts can capture only the simplest form of *full* redundancy where some of the siblings cover *identical* rules. However, HiCuts does not capture *partial* redundancy when some siblings share *many but not all* the rules. In particular, rules with wildcards are replicated across many siblings but the presence of other rules in the siblings prevents HiCuts from removing the redundancy. As the classifier size scales, this redundancy grows substantially.

2.2 HyperCuts

HyperCuts [12] extends HiCuts to address the above shortcomings. First, instead of cutting only one dimension at a node, HyperCuts proposes simultaneously cutting multiple dimensions to collapse the subtree associated with the single-dimensional cuts in HiCuts into one node in HyperCuts, thereby reducing HyperCuts' tree depth. Second, HyperCuts partly addresses the redundancy where siblings share some, but not all, of the rules. HyperCuts captures the cases where some of the rules are common to *all* the siblings by moving such rules up into the parent node. Because *individual* rules can be moved up, this heuristic is not limited by HiCuts' requirement that *all* the rules of a child need to be common with another sibling. After building the tree, a bottom-up traversal recursively moves up rules that are common to all the siblings into the parent. This approach reduces replication but adds extra memory accesses at each node to search the (multiple) moved-up rules.

In addition, HyperCuts observes that if the rules of a node do not cover the entire space of the node then the node's space can be compacted to remove the empty region to avoid generating empty child nodes (though empty nodes do not exist, the corresponding null child pointers do exist and take up memory). Any packet that falls outside a node's compacted region matches the default rule (whose action is to deny). This heuristic, called region compaction, does incur some memory overhead to store the coordinates of the compacted region at each node.

Although HyperCuts improves the tree depth and amount of memory, there are still two forms of considerable redundancy. First, because HyperCuts applies the moving up heuristic *after* building the tree, the number of moved-up rules along any path is limited to the number of rules at each leaf (i.e., *binth*). However, the number of replicated rules far exceeds this number in large classifiers. Though there are ways to move up more rules by applying the heuristic while the tree is being built, instead of after being built, doing so would increase the number of memory accesses to search the moved-up rules at each node. Further, the moving-up does not cover the cases where some but not all of the rules are common to only *some* of the siblings. The root of the problem lies in the fact that HyperCuts does not prevent replication but attempts to cure replication after the cuts have caused replication. We show that our idea of separable trees simply prevents nearly all replication. Second, HyperCuts inherits the redundant child pointers of HiCuts' third heuristic (see Section 2.1). Because the average node

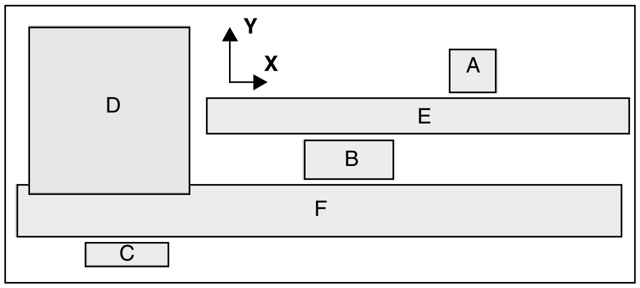

FIGURE 2. Separability of Rules

degree tends to be high for large classifiers (e.g., 50), this redundancy is significant. We show that our idea of equi-dense cuts significantly reduces this redundancy.

3 EFFICUTS

Despite its optimizations, HyperCuts incurs considerable memory overhead for large classifiers (e.g., in many classifiers with 100,000 rules, HyperCuts replicates rules by factors of 2,000-10,000 on average). Recall from Section 1 that our key observations are that the fundamental sources of memory overhead in HiCuts and HyperCuts are the large variations in (1) the size of overlapping rules, which are numerous, and (2) the density of the rule space. Because the algorithms build a single tree with overlapping small and large rules, the large variation in rule size causes significant rule replication. Fine cuts to separate the smaller rules needlessly partition the larger rules without achieving much rule separation while incurring considerable replication and memory overhead. The algorithms employ equi-sized cuts so as to identify the matching child via simple indexing into the child pointer array (Section 2.1). However, the large variation in the rule-space density causes a considerable increase in the tree size. Fine cuts to separate densely-clustered rules needlessly partition the sparse parts of the rule space which create many ineffectual tree nodes that separate only a few rules while incurring considerable overhead.

Our proposal, called *EffiCuts*, addresses the above overhead via 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. EffiCuts extends HyperCuts with these four ideas while retaining HyperCuts' basic framework to perform the cuts. Specifically, EffiCuts employs all of HiCuts' and HyperCuts' heuristics except for rule moving-up.

3.1 Separable Trees

To understand the concept of separable trees, we first pinpoint the root causes of rule replication in HiCuts and HyperCuts. Clearly, a single tree with overlapping small and large rules requires fine cuts to separate the small rules but causes replication of the large rules. In Figure 2, we see that the cuts that separate the small rules *A*, *B*, and *C* would partition the large rules *D*, *E*, and *F* resulting in significant rule replication. We find that HyperCuts replicates each of the largest of rules in our 100,000-rule classifiers by several hundred thousand times. Placing small and large rules in different trees would reduce this replication (e.g., in Figure 2, one tree for *A*, *B*, and *C*, and another for *D*, *E*, and *F*). Large rules are identified easily as those that have wildcards in many fields. Indeed, previous work [17] briefly suggests the possibility of creating two trees — one for rules with many wildcard fields and the other for the rest[1]. One could extend the scheme to more than two trees by partitioning the classifier into subsets of some number of similarly-sized rules so that each subset is in a different tree. We tried such an extension where we sorted the rules by size and grouped the largest 1% of the rules, the next 2%, the next 4%, the next 10%, and the remainder of the rules into distinct trees. This assignment achieved a factor of 10 reduction in replication. However, there was still considerable replication.

We discovered that there is another factor called *separability*, 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 our solution to avoid assigning and optimizing arbitrary percentages of the rules to distinct trees.

As discussed in Section 2.1, cutting the dimensions in which small and large rules overlap results in significant replication. In Figure 2, cutting the X dimension causes rampant replication of rules *D*, *E*, and *F*. As such, simply creating distinct trees for small and large rules would still incur replication if the large rules are not separable — i.e., there are dimensions in which not all the rules are either small or large. For example in Figure 2, even if the rules *D*, *E*, and *F* are in a different tree than the rules *A*, *B*, and *C*, the rules *D* and *F* would still be replicated.

To eliminate overlap among small and large rules, we separate *all* small and large rules by defining a subset of rules as *separable* if *all* the rules in the subset are either small or large in *each* dimension (e.g., in Figure 2, {*A*, *B*, *C*}, {*D*}, and {*E*, *F*} are separable). We build a distinct tree 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.

3.1.1 Identifying Separable Rules

To identify separable rules, we assume that wildcards imply large rules. (We refine this assumption later.) Accordingly, 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. Indeed, because non-wildcard rules typically contribute many unique projections in a field, HiCuts and HyperCuts would choose the field to cut, replicating wildcard rules (the first heuristic in Section 2.1). In contrast, with separable rules, HyperCuts can cut the non-wildcard fields with many unique projections without replicating the wildcard rules.

Based on the above considerations, our 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, *Cate-*

1. HyperCuts briefly mentions this two-tree scheme to reduce replication but does not pursue it further. In the results, the analysis claims that simultaneously cutting multiple dimensions suppresses the replication problem especially for rules with wildcard fields.

gory 1 has a sub-category for each non-wildcard field, for a total of $^5C_1 = 5$ sub-categories. *Category 2* has a sub-category for each pair of non-wildcard fields for a total of $^5C_2 = 10$ sub-categories. *Category 3* has a sub-category for each triplet of non-wildcard fields for a total of $^5C_3 = 10$ sub-categories. Because *Category 4* contains mostly small rules, we find that further sub-categories are unnecessary (but can be employed if needed).

So far, we have equated large rules with wildcards. However, a true wildcard may be too strict a condition to classify a rule as large. For instance, a value of "0.0.0.0/1" for the source IP field is not a wildcard, but is large compared to other possible values for the field (e.g., a few IP addresses). Therefore, we broaden our definition of largeness in a dimension to include rules that cover a large fraction, called *largeness_fraction*, of the dimension. In practice, rules are bimodal in that they cover either a large fraction of the dimension (e.g., 95% or more) or a small fraction (e.g., 10% or less). Consequently, varying *largeness_fraction* between 0.1 and 0.95 does not produce any noticeable effect on the resulting categorization. For our results, we use *largeness_fraction of* 0.5 for most fields. Because the source IP and destination IP fields are much larger than the other fields, we use a smaller *largeness_fraction* of 0.05 for these fields classifying rules covering even 5% of the fields to be large.

After partitioning a classifier into the above sub-categories, we employ HyperCuts to build a distinct tree for each sub-category.

3.2 Selective Tree Merging

The above categorization produces a total of 26 sub-categories requiring 26 distinct trees. In practice, however, many of the sub-categories may be empty (e.g., we saw, on average, 11 and, at most, 15 non-empty sub-categories for our 100,000-rule classifiers). Nevertheless, a key issue is that each packet has to look up all the separable trees requiring more lookups than those needed in HiCuts' or HyperCuts' single tree. We see that EffiCuts and HyperCuts make opposite choices in the memory size-bandwidth trade-off. To achieve a good compromise in this trade-off, we observe that merging two separable trees usually results in a tree whose depth, and hence number of accesses, is much less than the sum of the original trees' depths. However, merging arbitrary separable trees may completely destroy separability and result in significant rule replication. Therefore, we propose *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). We find that this compromise on separability 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, we do not merge the *Category 4* tree with any other tree.

Despite the above constraint of merging wildcards with non-wildcards in at most one field, there are many choices for merging. For example, a given *Category 2* tree which has two non-wildcard fields can merge with any of (a) two *Category 1* trees with non-wildcards in one of the two fields, or (b) three *Category 3* trees with non-wildcards in the two fields. *Category 2* trees cannot merge with each other or *Category 4* trees without violating the merging constraint because such a merge would lead to wildcards merging with non-wildcards in at least two fields. This reasoning can be generalized to state that for the merging constraint not to be violated, a *Category i* tree can merge only with either a *Category i-1* tree or a *Category i+1* tree. The specific choices for each tree can be worked out as done in the above example. Further, once a tree has been merged with another tree, the merged tree cannot be merged with yet another tree without violating the merging constraint. This observation is true because an additional merge would imply that either multiple trees from the same category have been merged together, or *Category i-1* and *Category i+1* trees have been merged, both of which violate the merging constraint. Therefore, merging can occur with at most one other tree.

To choose the pairs of trees for merging, we observe that merging smaller trees (containing fewer rules) usually results in less replication than merging larger trees. We also observe that trees generally contain more rules and are larger as we go from *Category 1* to *Category 4* (i.e., down the bulleted list of categories in Section 3.1.1). Accordingly, we consider one tree at a time from Category *2* or Category *3* (in that order), and we try to merge a *Category i* tree with a suitable *Category i-1* tree — i.e., one that satisfies the merging constraint. There are two cases to consider: (a) If there are more than one suitable *Category i-1* tree then we greedily choose the tree so that the field in which the small and large rules are mixed is the smallest (e.g., source and destination IP fields at 4 bytes are larger than port and protocol fields at 2 bytes). The rationale is that mixing small and large rules in smaller fields leads to less replication than in larger fields. This greedy choice may be sub-optimal in that the chosen tree may be the only suitable choice for another tree which would now remain unmerged. However, we find that the greedy choice works well in practice because there are not many suitable *Category i-1* trees from which to choose. (b) If no suitable *Category i-1* tree exists or all the suitable *Category i-1* trees have already merged with other trees, then we choose a *Category i+1* tree that satisfies the merging constraint using the same greedy choice as case (a).

While selective tree merging reduces the number of trees, each packet still has to look up a handful of trees (e.g., 5-6 trees). Consequently, the sequential search of the rules at the leaves of each tree would add up to a significant number of memory accesses if we are limited to retrieving only one rule instead of wider accesses. However, the leaves in HiCuts and HyperCuts hold pointers to the rules which are held in a table separate from the trees. The rules at a leaf need not be contiguous in the rule table, preventing wide-access retrievals. In EffiCuts, we place a copy of, instead of a pointer to, each rule at the leaf, forcing the rules to be in contiguous memory locations. One may think that this strategy may need extra memory because rules (13 bytes) are larger than pointers (4 bytes). 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. HiCuts' and HyperCuts' significant rule replication, however, makes this strategy unprofitable, but they lookup only one

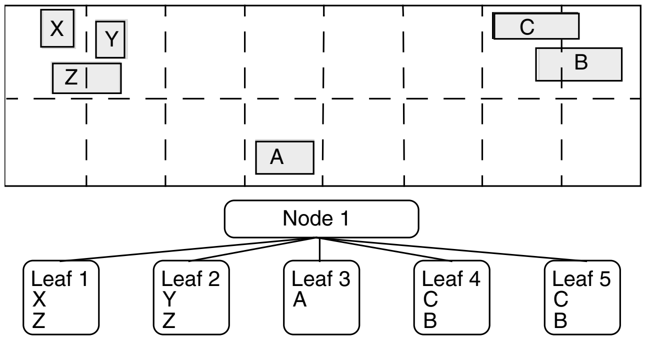

FIGURE 3. Variation in Density of Rule Space

leaf of a single tree, and hence search fewer rules than EffiCuts, which obviates wide-access retrievals.

Additionally, we also tried using fewer rules at each leaf to reduce the linear search overhead at the cost of some increase in the tree depths. We found that using 4 or 16 rules results in similar number of memory accesses but the latter has less rule replication.

3.3 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.

In Figure 3, we see that fine cuts are needed to separate the rules *X*, *Y*, and *Z* while minimizing the tree depth. However, such fine cuts partition and replicate the rules *A*, *B*, and *C*, and result in various subsets of the rules to be common among the siblings *Leaf 1* through *Leaf 5*. While *Leaf 4* and *Leaf 5* are fully redundant containing identical rules, *Leaf 1* and *Leaf 2* are partially redundant as they share some of the rules. Instead of fine cuts, if coarse cuts were used then we would need additional cuts to separate the rules *X*, *Y*, and *Z*. Recall that HiCuts and HyperCuts do not completely remove this redundancy. HiCuts' node merging (third heuristic in Section 2.1) merges the fully redundant *Leaf 4* and *Leaf 5* but incurs redundant child pointers in the parent *Node 1*. HyperCuts' moving up (Section 2.2) cannot remove the partial redundancy in the rules of siblings *Leaf 1* and *Leaf 2* where some of the rules are common among only some of the siblings.

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. We construct our unequal cuts 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.

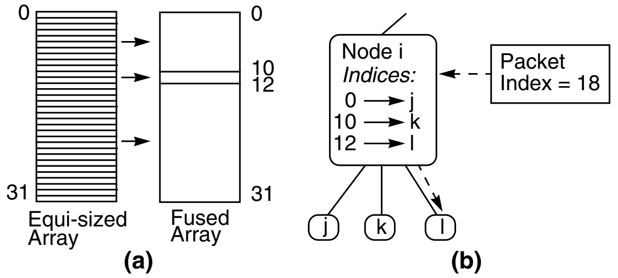

FIGURE 4. Equi-dense Cuts: (a) Fusion (b) Lookup

3.3.1 Fusion Heuristics

For our fusion of equi-sized cuts to produce unequal cuts, we propose a conservative, a moderate, and an aggressive heuristic.

Our 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. While HiCuts' node merging fuses identical siblings(Section 2.1), this heuristic fuses non-identical siblings.

To capture rule replication in denser regions, our 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.

Our 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. However, because the heuristic frequently leads to more replication and/or a deeper tree, we omit this heuristic in our implementation.

3.3.2 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 (Section 2.1). 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, we constrain the number of unequal cuts per node, and hence the number of comparators needed, not to exceed a threshold, called *max_cuts* (8 in our experiments). For nodes that need more cuts, we fall back

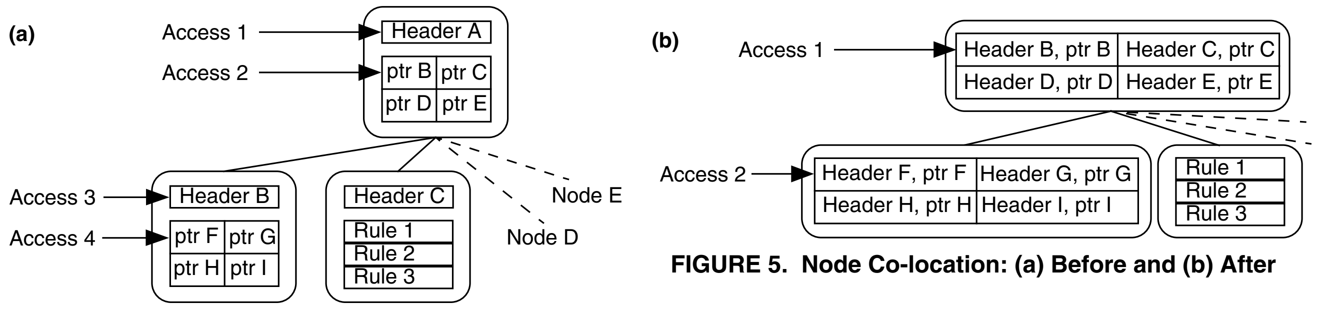

FIGURE 5. Node Co-location: (a) Before and (b) After

on equi-sized cuts, as in HiCuts and HyperCuts, with the accompanying redundancy. We use one bit per node to indicate whether the node uses equi-sized or equi-dense cuts. While the comparisons add some complexity to the processing at each node, we argue that this complexity is justified because of the immense reduction in memory size achieved by equi-dense cuts. We consider the node processing overhead in our results. To avoid memory management problems due to multiple node sizes, we always allocate space for a node to be the larger of equi-dense and equi-sized nodes (22 bytes).

Each node using equi-dense cuts stores the number of unequal cuts and an array of the starting indices of the cuts. To illustrate, consider a node with 32 equi-sized cuts of which 0-9, 10-11, and 12-32 are fused into 3 equi-dense, unequal cuts (see Figure 4). An incoming packet which generates a value of 18 would compare against the three indices, 0, 10, and 12, stored at the parent node, and select the closest index that is less than or equal to the packet value, in this case 12. We note that the fusion heuristics employed by equi-dense cuts fuse empty nodes into an adjacent non-empty sibling, obviating HyperCuts' region compaction. In place of the compaction information, we store the array of indices for equi-dense cuts. The difference between the information held in HyperCuts' nodes and EffiCuts' nodes are shown in Table 1.

3.4 Node Co-location

Apart from reducing memory size, equi-dense cuts have another benefit. In HiCuts and HyperCuts, the first part of a node holds the header identifying the dimension(s) cut at the node and the number of cuts, and the second part holds an array of pointers to the child nodes (Section 2.1). In EffiCuts' nodes using equi-dense cuts, the first part additionally holds the table of starting indices of each cut while the second part is similar to the above. In all three schemes, however, a packet has to look up the cut dimension and the number of cuts in each node's first part to determine its index into 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, as shown in Figure 5(a) (Narrow memories would require more accesses.).

To enable each node to require only one access and thereby achieve better memory bandwidth, we co-locate in contiguous memory locations a node's child-pointer array (the second part) with *all* the children's headers (their first parts), as shown in Figure 5(b). 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 co-located 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.)

The co-location precludes HiCuts' optimization of node-merging to reduce memory where array entries point to a single shared node instead of multiple, identical child nodes (Section 2.1). 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. However, the redundancy would be considerable for HiCuts and HyperCuts whose nodes usually have hundreds or thousands of children many of which are identical, making the co-location unprofitable for HiCuts and HyperCuts. For the same reason, we do not apply co-location to the EffiCuts nodes that use equi-sized cuts. Because in practice about 90% of nodes employ equi-dense cuts rather than equi-sized cuts, this restriction does not undo co-location's effectiveness.

To reduce further the number of memory accesses per node, we eliminate HyperCuts' rule moving-up optimization in EffiCuts 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 (Section 3.2). Rule moving-up reduces HyperCuts' rule replication, which is minimal for EffiCuts, and therefore, the elimination makes sense.

Finally, we consider updates to the classifiers. By providing more degrees of freedom than HiCuts and HyperCuts, EffiCuts facilitates incremental updates in at least two ways. First, because separable trees drastically reduce replication, updates are unlikely

Table 1: Node data in bytes

HyperCuts	**2** for header (number/dimension of cuts, other information such as internal node or leaf) **4** per moved-up rule pointer (up to 1) or leaf rule pointer (up to *binth* = 16) **16** for compacted region boundary **4** per child pointer
EffiCuts	**2** for header **13** per leaf rule (up to *binth* = 16) (leaf) **or** **16** for compacted region (equi-size) **or** **2** per unequal cut index (*max_cuts* = 7) (equi-dense) **4** per child pointer

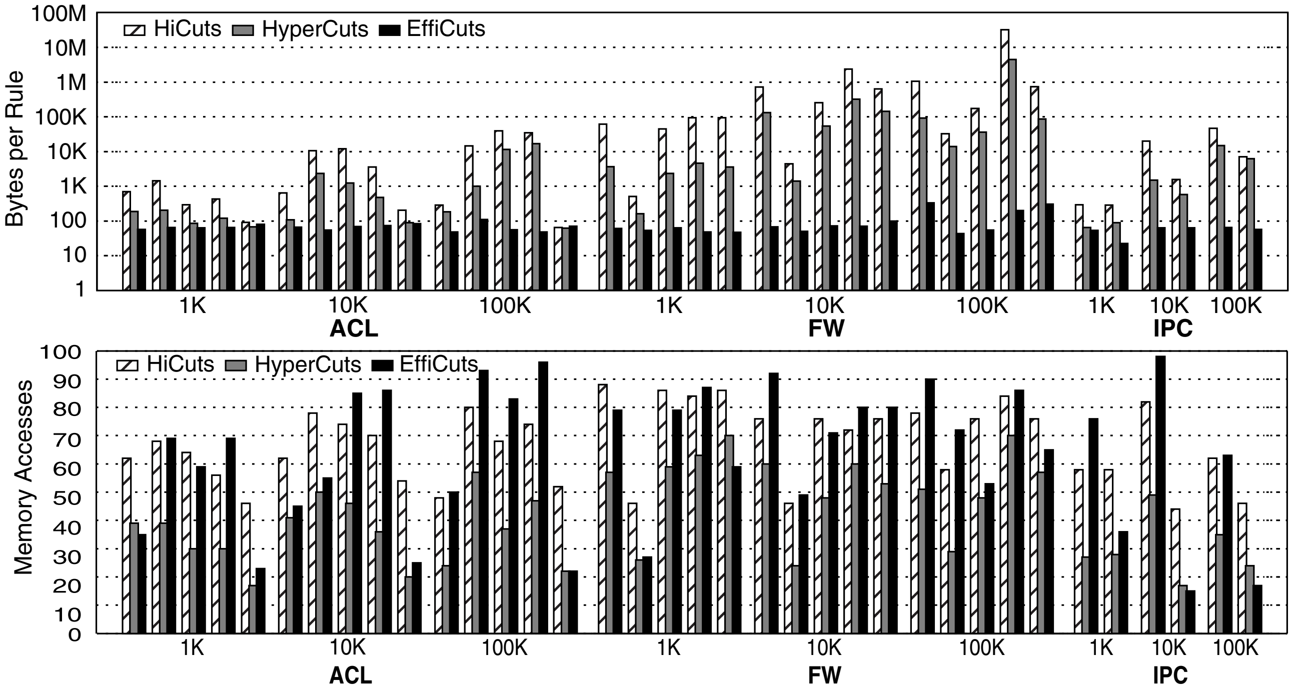

FIGURE 6. Total Memory (top) and Accesses (bottom) for HiCuts, HyperCuts, and EffiCuts

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.

4 EXPERIMENTAL METHODOLOGY

We implement HiCuts and HyperCuts faithfully capturing all the optimizations — HiCuts' four heuristics (Section 2.1) and HyperCuts' three heuristics (Section 2.2) — and setting *space_factor* to 4. We slightly modify HyperCuts' rule moving-up heuristic. First, to reduce the memory accesses of moved-up rules at a node, we constrain the number of moved-up rules per node to one. Second, while HyperCuts perform the moving up as a bottom-up pass after the tree is built (Section 2.2), we perform the moving up as the tree is built to increase the number of moved-up rules. The bottom-up pass moves up at most *binth* rules at leaves whereas our approach moves up more rules at both internal nodes and leaves.

We implement EffiCuts on top of HyperCuts and employ all but the rule moving-up heuristic of HyperCuts, as mentioned in Section 3.4. We run the EffiCuts algorithm with *space_factor* set to 8 and *largeness_fraction* set to 0.5 for all fields except source IP and destination IP which use 0.05. Because of its significantly lesser memory requirements, EffiCuts can use a larger *space_factor* than HyperCuts.

In Table 1, we show the information stored at each node in HyperCuts and EffiCuts. Region compaction requires 16 bytes because information only for the dimensions being cut is necessary. We assume memory widths for HiCuts, HyperCuts, and EffiCuts to be 13, 22, 22 bytes, respectively, These widths ensure that a single access returns an entire node, a pointer, or a rule. As discussed previously, HiCuts and HyperCuts require two accesses per node, whereas EffiCuts requires one access for nodes using equi-dense cuts, and two for nodes using equi-sized cuts.

Because we do not have access to large real-world classifiers, we use ClassBench [16] which creates synthetic classifiers with characteristics representative of real-world classifiers. We generate classifiers for all the types captured by ClassBench, namely, access control (ACL), firewall (FW) and IP chain (IPC). To study the effect of classifier size, we generate classifiers containing 1,000, 10,000 and 100,000 rules.

5 EXPERIMENTAL RESULTS

We begin by comparing EffiCuts against HiCuts and HyperCuts in terms of memory and number of memory accesses per packet match (Section 5.1). We then compare EffiCuts' power and throughput relative to TCAM and HyperCuts (Section 5.2). To analyze HyperCuts' and EffiCuts' memory requirements, we present a breakdown of the total memory into components (Section 5.3). Finally, we isolate the effects of separable trees and equi-dense cuts on memory (Section 5.4.1) and of co-location and tree-merging on memory accesses (Section 5.4.2).

5.1 Memory Size and Accesses

In Figure 6, we compare EffiCuts against HiCuts and HyperCuts in terms of memory per rule (top) and worst-case number of memory accesses (bottom). The metric memory per rule normalizes the memory needed across classifier sizes and also shows the memory overhead [15]. In the X axis, we show all the twelve classifiers generated by ClassBench — five ACL, five FW, and two IPC. We vary the classifier sizes as 1000, 10,000, and 100,000.

In the top graph, the Y axis shows bytes per rule in log scale. For each classifier of a specific size, we show three bars, from left to right, one each for HiCuts, HyperCuts, and EffiCuts. We see that

Table 2: EffiCuts' Average Memory Reduction over HyperCuts

	ACL			FW			IPC			Mean
	1K	10K	100K	1K	10K	100K	1K	10K	100K	
HyperCuts (MB)	0.12	7.58	538	2.37	1151	77045	0.05	9.45	986	
EffiCuts (MB)	0.06	0.59	5.11	0.05	0.63	13.3	0.03	0.58	5.77	
Reduction	2.0	13	105	51	1816	5787	1.7	16	171	57.4

as the classifier size increases within each classifier type (ACL, FW, or IPC), HiCuts' and HyperCuts' memory requirements grow quickly (the Y axis is log scale) and vary widely across classifier types. In contrast, EffiCuts' memory requirements are almost constant across classifier types and sizes. For the 100,000-rule classifiers, EffiCuts stays well below HiCuts and HyperCuts. HiCuts and HyperCuts require significantly large memories for large firewall classifiers due to the fact that a significant fraction of firewall rules (about 30%) include many wildcard fields. These large rules incur rampant replication in HiCuts and HyperCuts because of being mixed in with the smaller rules. In contrast, EffiCuts' separable trees isolate these rules from the smaller rules and hence prevent replication. As an aside, the trend between HiCuts and HyperCuts is in line with the results reported in the HyperCuts paper. In Table 2, we show the average total memory requirement for HyperCuts and EffiCuts as well as the factors by which EffiCuts reduces HyperCuts' total memory. EffiCuts' total memory requirement scales linearly with classifier size as compared to HyperCuts' super-linear growth.

In the bottom graph. the Y axis shows the worst-case number of memory accesses per packet for each of the schemes. Again, the trend between HiCuts and HyperCuts matches the HyperCuts paper results. In most cases, EffiCuts' multiple trees require more memory accesses than HyperCuts' single (albeit much larger) tree. However, EffiCuts requires on average 50% and at worst 3 times more accesses than HyperCuts. Although EffiCuts needs more accesses than HyperCuts, EffiCuts' orders-of-magnitude reduction in memory implies a much smaller SRAM than HyperCuts. In the following section, we analyze the effect of the number of memory accesses on packet throughput.

5.2 Comparing EffiCuts, HyperCuts, & TCAM

To get the complete picture, we compare EffiCuts to TCAM and HyperCuts in terms of throughput (cycle time — time between two accesses) and power. For this analysis, we use CACTI [10] to model TCAM and SRAM. We assume a 45 nm process and model the SRAM sizes needed for the typical case of HyperCuts and EffiCuts. Additionally, recall that HyperCuts and EffiCuts require some processing for region compaction (equi-sized cuts in HyperCuts and EffiCuts) and to determine the matching child (equi-dense cuts in EffiCuts). Our estimate for this processing overhead is less than the cycle time of SRAM (for 45 nm technology, roughly 15 levels of logic for comparing and choosing one of 8 16-bit values fit well within 170 ps which is one cycle time). Thus, packet throughput would be limited by SRAM cycle time rather than node processing. For TCAM, we assume 100K entries, 13-byte width with a fairly low range-expansion factor of 2 [9]. To achieve high throughput, CACTI employs wave-pipelined implementations [10].

Table 3 shows the throughput in terms of cycle time and per-access energy of TCAM, HyperCuts, and EffiCuts. For HyperCuts and EffiCuts, we also include the number of accesses and memory size for a typical classifier from each type (ACL, FW, and IPC). While TCAM cycle time and per-access energy are worse than those of the HyperCuts' and EffiCuts' SRAMs due to raw hardware differences, TCAM needs only one access whereas HyperCuts and EffiCuts need many. These numerous accesses degrade the throughput by a factor equal to the number of accesses, causing EffiCuts to fall behind TCAM (e.g., for ACL 100K, EffiCuts' net throughput is 1/(83*0.18) = 67 Mpps whereas TCAM's throughput is 1/7.46 = 134 Mpps). (Our TCAM throughput is close to that of OC-768.). To address this issue, we propose to use multiple copies of the EffiCuts' search structure to achieve the same throughput as TCAM, as shown in Table 3. Multiple copies are a viable option because of EffiCuts' small memory requirement. We see that only 1-2 copies suffice. (Both TCAM's and EffiCuts' throughputs can be improved further by more copies.) We also show the number of copies needed for HyperCuts though multiple copies may be impractical due to its large memory.

While EffiCuts' energy also degrades due to the numerous accesses, TCAM's energy is worse even after adjusting for the number of accesses (e.g., for ACL 100K, EffiCuts' net energy per packet is 83*0.51 = 93.3 nJ whereas TCAM's energy is 169.51 nJ). Due to its large memory, HyperCuts consumes significantly more energy than EffiCuts (e.g., for ACL 100K, EffiCuts' net energy per packet is 83*0.51 = 93.3 nJ whereas HyperCuts' net energy is 37*5.57 = 206.09 nJ). Because power is a better metric for packet classification than energy, we show HyperCuts' and EffiCuts'

Table 3: Typical Power and Throughput Comparison

Classifier Type	TCAM			HyperCuts						EffiCuts					
	Cycle time (ns)	Energy (nJ)	Power (W)	# Memory accesses	Size (MB)	Cycle time (ns)	Energy (nJ)	# Copies	Power (W)	# Memory accesses	Size (MB)	Cycle time (ns)	Energy (nJ)	# Copies	Power (W)
ACL 100K	7.46	169.5	23	37	1084	0.18	5.6	1	31	83	5.3	0.18	0.51	2	6
FW 100K	7.46	169.5	23	48	2433	0.21	8.3	2	81	53	3.7	0.20	0.43	2	4
IPC 100K	7.46	169.5	23	24	575	0.17	4.3	1	26	17	5.5	0.18	0.51	1	3

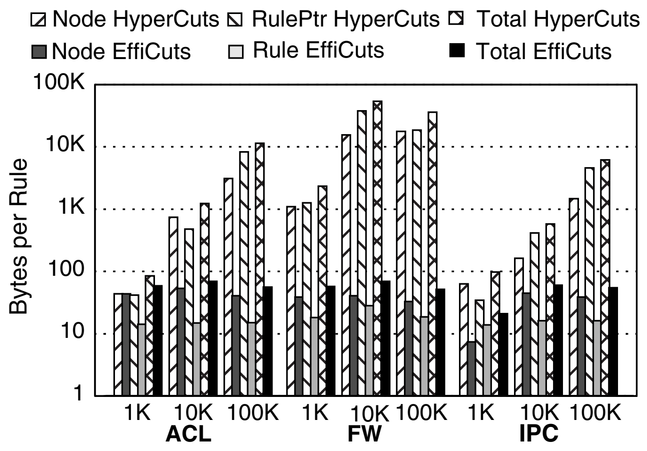

FIGURE 7. HyperCuts & EffiCuts Memory Breakdown

power relative to TCAM's (where power = copies * per-access energy/cycle time). We see that EffiCuts consumes a factor of 4-8 less power than TCAM. Due to its large memory, HyperCuts consumes more power than TCAM. Further, HyperCuts consumes more power than EffiCuts despite requiring fewer accesses. We note that though the cycle times and energies of real products may be different than our numbers, the overall improvements of EffiCuts will likely hold.

5.3 Breakdown of Memory into Components

To analyze the memory required by EffiCuts, we decompose the memory requirement per rule into two components: overhead for nodes (excluding HyperCuts' moved-up rule pointers), and storage for the rules in the leaves (including HyperCuts' moved-up rule pointers). Recall that EffiCuts' leaves contain rules while HyperCuts' leaves contain rule pointers. We choose this decomposition because our separable trees decrease the storage for rules by reducing rule replication and our equi-dense cuts decrease the storage for nodes by fusing nodes. In Figure 7, we show these two components for HyperCuts and EffiCuts. In the X axis, we show one typical classifier for each type (ACL, FW, and IPC) while varying the classifier size as 1000, 10,000, and 100,000 rules. We show only one typical classifier per type to avoid cluttering the graph. The Y axis shows in log scale the two components and the total memory for the two schemes. The bars are ordered, from left to right, as node storage for HyperCuts and EffiCuts, rule pointer storage for HyperCuts, rule storage for EffiCuts, and the total memory for HyperCuts and EffiCuts. Though the components add up to the total, it may not seem so due to the log scale.

We see that, in most cases, EffiCuts' node storage is significantly less than that of HyperCuts. This reduction is due to both separable trees which drastically reduce rule replication and equi-dense cuts which fuse redundant nodes. Similarly, we see that in all cases EffiCuts' rule storage is significantly less than HyperCuts' rule pointer storage, although rules are larger than the pointers (13 versus 4 bytes). This reduction is larger than that for internal node storage and is primarily due to separable trees.

To support the above analysis, we show the amount of rule replication and the number of nodes in the typical case for both HyperCuts and EffiCuts in Table 4. We quantify rule replication as the ratio between the number of rules in the classifier and the total number of rules (EffiCuts) or rule pointers (HyperCuts) in the tree. We see that EffiCuts reduces both rule replication and node count

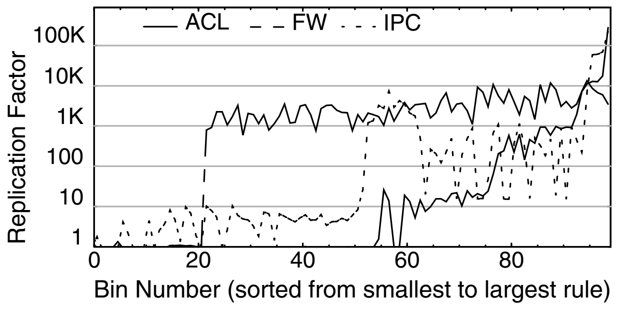

FIGURE 8. Rule Replication Factor in HyperCuts

by large factors. The reduction in the rule replication is generally more than that in the node count, matching the data shown in Figure 7 and the discussion above. For the studied classifiers, EffiCuts' replication remains below the remarkably low factor of 2 in the typical case and 9 in the worst case (not shown). Further, in 33 of the 36 classifiers, replication remains below a factor of 3.

Finally, while Table 4 shows the typical rule replication factor for HyperCuts, we show the full distribution in Figure 8. The X axis shows the rules binned into 100 bins of increasing rule size left to right. The Y axis shows in log scale the replication factor for the rules in each bin. We see that the smaller rules are replicated less than the larger rules while the largest of rules are replicated more than a few hundreds of thousands of times. More than 80% of the FW rules are replicated more than 1000 times whereas about 35% and 10% of the IPC and ACL rules, respectively, are replicated between 100-1000 times. We do not show this distribution for EffiCuts because its replication factor is fairly low.

5.4 Sensitivity Study

In this section we isolate the effects of each of EffiCuts' four novel ideas — separable trees, equi-dense cuts, selective tree merging, and node co-location — on memory and the number of memory accesses. Because separable trees and equi-dense cuts reduce replication but not the number of memory accesses, we show their effect only on memory (Section 5.4.1), and vice versa for tree merging and co-location (Section 5.4.2).

Table 4: Typical Memory Reduction Comparison

Type	Size	HyperCuts		EffiCuts			
		Rule repl-ratio	#Nodes (x 10^3)	Rule repl. ratio	#Nodes (x 10^3)	Rule repl. reduction	#Nodes reduction
ACL	1k	7	2	1.02	0.2	7	10
	10k	115	280	1.06	2.1	108	133
	100k	2078	27412	1.07	28	1942	997
FW	1k	313	47	1.3	0.3	241	157
	10k	9396	10993	1.89	7.4	4971	1486
	100k	4611	77530	1.09	62	4230	1255
IPC	1k	5	1	1.02	0.1	5	10
	10k	99	196	1	6.7	99	29
	100k	1140	22794	1	67	1140	341

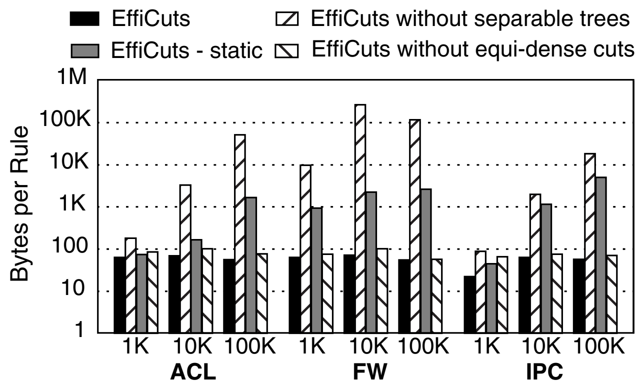

FIGURE 9. Impact of Separability & Equi-dense Cuts

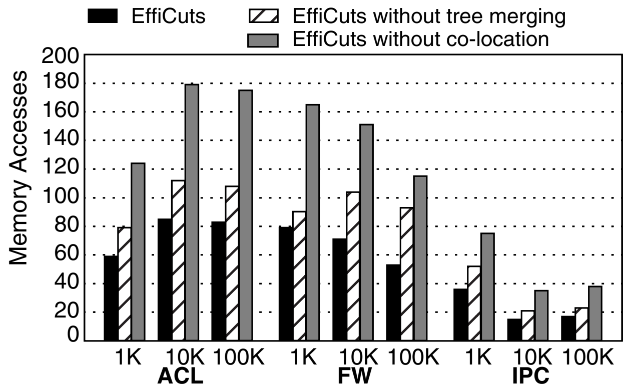

FIGURE 10. Impact of Tree Merging & Co-location

5.4.1 Isolating Separability and Equi-dense Cuts

In Figure 9, the Y axis shows in log scale, from left to right, the memory per rule for the original EffiCuts, EffiCuts without separable trees, EffiCuts using a static distribution of rules into trees based on size instead of separability, and EffiCuts without equi-dense cuts (i.e., all cuts are equi-sized).The static distribution sorts the rules by size and then groups the largest 1% of the rules, the next 2%, the nest 4%, the next 10% and the remainder of the rules (83%) into distinct trees (as described in Section 3.1). Similar to Figure 7, the X axis shows one typical classifier for each type (ACL, FW, and IPC) while varying the classifier size as 1000, 10,000, and 100,000 rules.

Figure 9 shows that the original EffiCuts is better than EffiCuts without separable trees and EffiCuts without equi-dense cuts by several factors and factors of 2-4, respectively (recall that the Y axis is log scale). While the impact of equi-dense cuts is smaller than that of separable trees or the static distribution scheme, the former's impact is obscured by an interplay with node co-location. EffiCuts without equi-dense cuts does not employ node co-location which modestly increases memory for the original EffiCuts and offsets the memory reduction achieved by equi-dense cuts. Thus, both separability and equi-dense cuts are important. Though the static distribution scheme falls behind separability, the scheme achieves about 10x memory reduction over HyperCuts. Thus, while simply separating small and large rules without considering the rule-space's dimensions reduces rule replication, separating small and large rules on a per-dimension basis nearly eliminates it.

5.4.2 Isolating Selective Tree Merging & Co-location

Figure 10 shows our representative classifiers along the X axis and shows memory accesses along the Y axis. The bars, from left to right, show the base EffiCuts algorithm, EffiCuts without tree-merging, and EffiCuts without co-location. Both variants perform considerably worse than EffiCuts. Co-location effectively halves the number of accesses from two accesses per node to one. The reduction is of course, not quite half, because unfused nodes are not co-located (Section 3.4). Tree merging, though not as effective as co-location, also contributes significantly to access reduction, 30% on average. In many cases, tree merging reduces the number of trees by 50-60% compared to the unmerged trees, thereby linearly searching the rules in only half as many leaves, while only slightly increasing tree depth.

6 RELATED WORK

Packet classification is a well-studied problem addressed by algorithmic and TCAM approaches. Previous algorithms to address this problem can be classified as being based on bit vectors, cross producting, tuple search, and decision trees. Bit-vector approaches [8] match incoming packets to rules on each dimension to produce per-dimension bit vectors which are then ANDed. These approaches require large bit vectors that grow quadratically with the number of rules. Aggregration schemes [2] compress the bit vectors by exploiting the vectors' sparseness. However, the compression achieved is modest in the presence of wildcards.

Cross-producting approaches [5] look up per-dimension tables for all the dimensions in parallel and combine the results of the per-dimension look-ups via one or more look-ups of cross-product tables. Further, each table look up can be sped up by hashing [7]. While the number of look-ups is deterministic, the cross-product tables grow rapidly with the classifier size. Grid-of-tries schemes [1] prune the cross-product space by first considering the cross product of only the two dimensions of source IP and destination IP, and then linearly searching the matching rules. However, the initial two-dimensional tables grow rapidly with the classifier size. Another approach shrinks the cross-product tables by storing only the valid cross products [15]. However, the tables have to be searched, instead of being indexed, incurring many accesses.

Tuple-search approaches [14] exploit the fact that the number of unique prefix lengths for each dimension is far fewer than the number of rules. The approaches partition the classifier on the basis of these lengths allowing incoming packets to construct a hash index for each partition. The partitions are looked-up in parallel to find the matching rules which are then pruned based on their priority. However, the partition size is unconstrained and there may be hash collisions, resulting in unpredictable performance.

We have extensively discussed the decision-tree approaches HiCuts and HyperCuts throughout the paper. Another decision tree approach, called Common Branches [3], reduces rule replication by placing the about-to-be replicated rules at the node where replication would occur. The rules at a node are searched linearly. However, the amount of linear search is unconstrained and often results in long searches which degrade performance. A recent work, HyperSplits [11], attempts to balance the tree by splitting a dimension into two unequal parts containing nearly equal number of rules. Because of being restricted to splitting only one dimension into two parts, HyperSplits incurs worse depth than HiCuts. As mentioned in Section 1, Modular packet classification [17] reduces rule replication by binning the rules into multiple trees based on a selected set of prefixes. In contrast, EffiCuts bins rules based on separability, drastically reducing rule replication.

Previous work [13] optimizes TCAM power by partitioning the classifier into subsets each of which is identified by a prefix. However, evenly distributing the rules while achieving reasonably-sized sub-arrays is hard. Other work [9] reduces range expansion in TCAM by representing ranges as ternary values instead of prefixes. Our results in Section 5.2 assume a factor of 2 for range expansion which is well under that achieved in [9].

7 CONCLUSION

Previous decision-tree algorithms for packet classification, such as HiCuts and HyperCuts, incur considerable memory overhead due to two key issues: (1) Variation in rule size: The algorithms' fine cuts for separating the small rules replicate the large rules that overlap with the small rules. (2) Variation in rule-space density: The algorithms' equi-sized cuts to separate the dense parts needlessly partition the sparse parts resulting in many ineffectual nodes that hold only a few rules. We proposed *EffiCuts* which drastically reduces the overhead. To address the first issue, we eliminate overlap among small and large rules by separating small and large rules in each dimension into distinct *separable trees* so that each dimension can be cut finely or coarsely without incurring replication. To reduce the multiple trees' extra accesses which degrade throughput, we *selectively merge separable* trees mixing rules that may be small or large in at most one dimension. In the trade-off between memory size and bandwidth, HyperCuts considerably increases size to gain some bandwidth whereas EffiCuts modestly degrades bandwidth to reduce size drastically. To address the second issue, we employ unequal, *equi-dense cuts* which distribute a node's rules as evenly among the children as possible, avoiding ineffectual tree nodes at the cost of some small processing overhead in the tree traversal. Equi-dense cuts enable us to reduce the number of accesses per tree node by *co-locating* parts of a node and its children.

Using ClassBench [16], we showed that for classifiers with 1000 to 100,000 rules (1) EffiCuts drastically reduces the worst-case rule replication to less than a factor of nine as compared to HyperCuts' factor of several thousands; and (2) For similar throughput, EffiCuts needs a factor of 57 less memory than HyperCuts and a factor of 4-8 less power than TCAM. By reducing the total memory of decision tree-based algorithms by orders of magnitude, EffiCuts greatly lowers the barrier for their adoption. Schemes like EffiCuts will likely be valuable in the future when larger classifiers will need to be looked up at higher throughputs while consuming reasonable power.

ACKNOWLEDGMENTS

We thank our shepherd, Jon Turner, for his valuable comments. This work is supported, in part, by a Clare Booth Luce fellowship.

REFERENCES

[1] F. Baboescu, S. Singh, and G. Varghese. Packet Classification for Core Routers: Is there an alternative to CAMs? In *Proceedings of the 22nd IEEE Conference on Computer Communications (INFOCOM '03)*, pages 53 – 63 vol.1, 2003.

[2] F. Baboescu and G. Varghese. Scalable Packet Classification. In *Proceedings of the ACM SIGCOMM '01 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, pages 199 – 210, 2001.

[3] E. Cohen and C. Lund. Packet Classification in Large ISPs: Design and Evaluation of Decision Tree Classifiers. In *Proceedings of the '05 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, pages 73 – 84, 2005.

[4] A. Feldman and S. Muthukrishnan. Tradeoffs for Packet Classification. In *Proceedings of the 19th IEEE Conference on Computer Communications (INFOCOM '00)*, pages 1193 – 1202 vol.3, 2000.

[5] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proceedings of the SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99)*, pages 147–160, 1999.

[6] P. Gupta and N. McKeown. Classifying Packets with Hierarchical Intelligent Cuttings. *IEEE Micro*, 20(1):34 – 41, 2000.

[7] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar. Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*, pages 203 – 215, 2006.

[8] T. V. Lakshman and D. Stiliadis. High-speed Policy-based Packet Forwarding using Efficient Multi-dimensional Range Matching. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '98)*, pages 203 – 214, 1998.

[9] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for Advanced Packet Classification with Ternary CAMs. In *Proceedings of the ACM SIGCOMM '05 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '05)*, pages 193 – 204, 2005.

[10] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, HP Labs.

[11] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet Classification Algorithms: From Theory to Practice. In *Proceedings of the 28th IEEE Conference on Computer Communications (INFOCOM '09)*, pages 648 – 656, 2009.

[12] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification using Multidimensional Cutting. In *Proceedings of the ACM SIGCOMM '03 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '03)*, pages 213 – 224, 2003.

[13] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended TCAMs. In *Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP '03)*, page 120, 2003.

[14] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proceedings of the ACM SIGCOMM '99 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99)*, pages 135 – 146, 1999.

[15] D. Taylor and J. Turner. Scalable packet classification using distributed crossproducing of field labels. In *Proceedings of the 24th IEEE Conference on Computer Communications (INFOCOM '05)*, pages 269 – 280 vol. 1, 2005.

[16] D. E. Taylor and J. S. Turner. Classbench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking (TON)*, 15(3):499 – 511, 2007.

[17] T. Woo. A Modular Approach to Packet Classification: Algorithms and Results. In *Proceedings of the 19th IEEE Conference on Computer Communications (INFOCOM '00)*, pages 1213 – 1222 vol.3, 2000.