TreeCAM: Decoupling Updates and Lookups in Packet Classification

Balajee Vamanan and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{bvamanan, vijay}@ecn.purdue.edu

ABSTRACT

Packet Classification is a key functionality provided by modern routers. Previous approaches — TCAM and algorithmic — perform well in either lookup efficiency (power and number of accesses) or update effort but not both. To perform well in both, we propose *TreeCAM,* which employs three novel ideas. (1) *Dual versions* of TreeCAM's decision tree to decouple lookups and updates: A coarse version with a few thousand rules per leaf achieves efficient lookups and a fine version with a few tens of rules per leaf reduces update effort. (2) *Interleaved layout* of the rules in the TCAM: Combined with the fine version's few rules per leaf, the layout enables us to bound our worst-case update effort. (3) *Path-by-path updates* to enable update work to be interspersed with packet lookups (i.e., non-atomic updates), eliminating packet buffering or packet drops during update. Using simulations of 100,000-rule classifiers, we show that TreeCAM performs well in both lookups and updates: (1) 6-8 TCAM subarray accesses per packet, matching modern TCAMs. (2) close to an idealized TCAM in worst-case update effort while requiring little buffering of packets.

Categories and Subject Descriptors:

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

General Terms:

Algorithms, Design, Performance

Keywords:

Packet Classification, Updates

1 INTRODUCTION

Packet classification involves determining the highest-priority rule to which each network packet matches out of a set of rules (i.e., a classifier). Each rule specifies a desired action on matching packets identified by a combination of the packet fields (e.g., source/destination IP, source/destination port, and protocol). Packet classification is vital for QoS, security, and traffic monitoring and analysis. While line rates continue to improve, classifiers also grow in size due to rule customization for virtual private networks (VPNs) and quality of service (QoS). Thus, larger classifiers need to be searched at higher rates (e.g., several tens of thousands of rules every few nanoseconds).

The key metrics for packet classification schemes are number of accesses needed per packet (a proxy for packet throughput), power, and update effort. While the first two metrics are self-evident, we elaborate on update effort by extrapolating from [2]. Updates add or remove either (1) a few thousands of rules as virtual interfaces are created or destroyed (e.g., ten interfaces per minute with 5000 rules per interface), or (2) a handful of rules at time granularities of flows for access control or QoS (e.g., 10 rules every ms). Though either update flavor's net rates are much lower than packet rates (ms versus ns), most schemes move or change a large fraction of the rules for at least a small fraction of the updates, leading to three problems: (1) significant loss of memory bandwidth to updates; (2) long hold-ups of memory to achieve a consistent state, causing packet drops or incurring cost and complexity for buffering packets; and (3) failure to meet latency requirements of a few milliseconds for QoS (e.g., to stay well below human perception times). For example, assume that the update rate is 10 per ms (either flavor), packet rate is 1 per 10 ns (~40Gbps), and 20% of 100,000 rules are moved (read and written) for 5% of the updates. Then, updates overall would impose 20% bandwidth overhead, and those 5% of the updates would each (1) require 40,000 packets to be buffered and (2) take 1 ms assuming each move takes 50 ns. Moreover, current trends of OpenFlow [7] and on-line classification require updates at flow granularities (i.e., milliseconds), potentially increasing the update rates in the future.

Previous packet classification schemes, which fall in two broad categories — TCAM and algorithmic, perform well in either lookup efficiency (power and number of accesses) or update effort but not both. In unoptimized TCAMs, each packet searches all rules in one fast access at the cost of high power. Modern TCAMs reduce power by partitioning rules into smaller subarrays (e.g., 4K rules), and searching only a small subset of rules instead of searching all the rules. Because partitioned TCAM details are not publicly available, we consider the Extended TCAM [12] as a representative. While partitioning reduces power, modern TCAMs still suffer from high update complexity. Because TCAMs physically order the rules based on priority for fast determination of the highest-priority match, updates require moving rules to achieve proper ordering. Previous optimizations reduce update effort either (1) only in some favorable cases while resorting to moving or changing all the rules in the other cases [9], or (2) in many cases but may require significant

ACM CoNEXT 2011, December 6-9, 2011, Tokyo, Japan.

effort in finding and maintaining [18], and updating [11] the rules' relative priority. While the Extended TCAM paper does not address updates, the approach inherits TCAM's update overheads. The update effort may be reduced by using finer distribution which would substantially increase the number of accesses for each lookup, as our results show. As such, modern TCAMs incur high update effort.

Algorithmic schemes, such as HiCuts [4], HyperCuts [10], and EffiCuts [17], build a decision tree to prune the search by partitioning the rules into small subsets. However, there are two problems: (1) Lookup requires many tens of accesses per packet which degrades throughput (e.g., 80 accesses for 100,000 rules). (2) None of the decision tree papers address updates. Nevertheless, the algorithms do not order the rules to determine the highest-priority match and instead sequentially compare all the rules in the matching leaf, avoiding the rule re-ordering effort incurred by TCAMs. However, updates would deepen the tree in an unbalanced manner and increase lookup accesses, requiring high re-balancing effort. Other approaches [1] [3] support fast updates but require prohibitively large memory. In summary, the previous schemes do not perform well in both lookups and updates.

To address this limitation, we propose *TreeCAM,* which employs three novel ideas.

(1) Dual Tree Versions: To reconcile the tension between lookups and updates, we propose our central idea of *dual versions* of TreeCAM's decision tree to decouple lookups and updates: a coarse tree version with a few thousands of rules per leaf (e.g., 4K) which reduces the number of accesses per packet and power; and a fine tree version with a few tens of rules per leaf (e.g., 16) which reduces the update effort. Our coarse tree is similar to a partitioned TCAM where each leaf in the tree maps to a TCAM subarray, and our fine tree is similar to those in previous algorithmic schemes. Because lookup rates are high and update rates are low, and the coarse tree is for lookups and the fine tree is for updates, we maintain the coarse tree in TCAM and the fine tree in slow control memory. The fine tree is logically superimposed on the coarse tree to keep the trees consistent upon updates. While Extended TCAM with smaller partitions to reduce the update effort would require considerably more accesses, TreeCAMs' dual tree versions avoid this problem by decoupling lookups and updates.

(2) Interleaved Rule Layout in TCAM: TreeCAM inherits the update overheads of both TCAMs and decision trees: TCAM's rule re-ordering and decision trees' re-balancing. We address this challenge by reducing the overhead of one part by leveraging the other. For the first overhead, we exploit the well-known idea that because a packet cannot match non-overlapping rules, only the overlapping rules need to be priority-ordered in the TCAM [9]. We combine this idea with decision trees via the key observation that the rules in a leaf may overlap with each other but not with rules in other leaves because leaves cover non-overlapping subspaces. Consequently, a leaf's rules have to be ordered only among each other but not with other leaves' rules. With only a few rules per leaf, the fine tree significantly reduces the per-leaf ordering effort. While previous work on decision trees reduce packet lookup effort, this paper is the first to exploit decision trees for reducing update effort.

For the second overhead of tree re-balancing, we optimize the rule layout in the TCAM. Our re-balancing simply displaces excess rules from a leaf to its neighbor. While the per-leaf priority-ordering effort is low, making room for an incoming rule would require moving the rules in many fine-tree leaves if each leaf's rules are contiguous in the subarray (e.g., 256 16-rule leaves in a 4k-entry subarray). Instead, we propose an *interleaved layout* in which the rules at each priority level from all the leaves within a subarray are contiguous. This interleaving achieves flexible sharing of a subarray's free entries among all its leaves, and is fundamental to bounding the update effort to moving the few rules per leaf in the fine tree.

Assuming about 25 subarrays in a 100,000-entry TCAM, TreeCAM would move around 500 rules in the worst case for re-balancing as compared to conventional TCAMs which would move a large fraction of the rules. Because of its low update effort, TreeCAM does not require any batching to amortize the update effort as in [18], and therefore does not incur the latency penalties of batching.

(3) Path-by-path Updates: Because holding up packet lookups for the 500 moves in the worst case may be problematic, we make our updates non-atomic by performing our tree re-balancing path-by-path which involves multiple singleton rule movements which can be interspersed with packet lookups. By updating the tree path corresponding to the rule being moved, we ensure that the TreeCAM is in a consistent state after each singleton movement. Because the coarse tree's paths are short, the path updates require minimal effort. While CoPTUA [18] provides non-atomic updates for conventional TCAMs, TreeCAM addresses decision tree re-balancing. CoPTUA worsens the common-case update effort (e.g., up to 50x more than conventional TCAMs) to achieve non-atomic updates. In contrast, TreeCAM leverages the fine tree to enable non-atomic updates without worsening the common-case effort.

To summarize, TreeCAM's contributions are:

- *Dual Tree Versions* which decouple lookups and updates to achieve both efficient lookups and low-effort updates;
- *Interleaved Rule Layout* which is fundamental to reducing update effort; and
- *Path-by-path Updates* which enable efficient, non-atomic updates.

Using simulations of 100,000-rule classifiers, we show that TreeCAM performs well in both lookups and updates: 6-8 TCAM subarray accesses per packet (matching modern TCAMs) and close to the worst-case update effort of an idealized TCAM while requiring little buffering of packets.

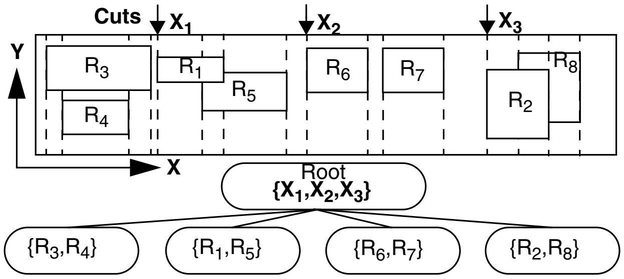

FIGURE 1. Sort-based Heuristic

The rest of the paper is organized as follows: We describe the coarse tree version for packet lookups and the fine tree version for updates in Section and Section 3, respectively. 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 COARSE TREE VERSION

We build a coarse tree that partitions the rules among TCAM subarrays, similar to modern partitioned TCAMs. Our coarse tree has a depth of only two, requiring two memory accesses per packet.

2.1 Decision Tree Algorithm

Decision tree algorithms [4][10][17] partition the multi-dimensional rule space (e.g., source IP, destination IP, source port, destination port, protocol) by building a decision tree so that the rules are separated evenly into the tree's leaves. The algorithms successively partition the resulting subspaces to prune down to some *binth* [4]. For our coarse tree, we choose a *binth* of a few thousands of rules so that the entire leaf node can fit in a TCAM subarray.

Extended TCAM, our representative for partitioned TCAMs, uses complicated heuristics to choose which dimension(s) to cut at every tree node and how many cuts to perform with the goal of reducing the tree depth without prohibitively increasing rule replication. Because these heuristics may need to be computed for a significant fraction of the updates, such complicated heuristics significantly increase update effort. Therefore, we opt for a simple sort-based heuristic where we choose the dimension with the most unique projections of the rules. We sort the start points of the projections and evenly partition the rules by introducing cuts after a fixed number of the start points in the sorted order (e.g., 4K rules). Each rule partition is placed in a leaf. Figure 1 shows a two-dimensional rule space with rules R_1 through R_8 and the corresponding tree. Sorting the projections of the rules on the X dimension and equally partitioning the rules into two per leaf leads to cuts at X_1 through X_3. Because our cuts are at the start points, our cuts naturally reduce rule replication.

Naively applying our heuristic may result in considerable rule replication which, as observed in EffiCuts [17], arises mainly due to overlap between small and large rules. Fine cuts to partition small rules (i.e., non-wildcard rules) needlessly cut and replicate large rules (i.e., wildcard rules). To address this issue, we employ EffiCuts' separable trees which segregate small and large rules into distinct trees. However, our sort-based heuristic is simpler than, and different from, *equi-dense cuts* used by EffiCuts. EffiCuts groups rules with wildcards (or almost wildcards) in the same dimensions into a separable tree. Though there could be up to 31 trees for five dimensions, we obtain only 5-6 trees for our 100,000-rule classifiers while nearly avoiding replication. Overall, TreeCAM requires 6-7 accesses per packet for our classifiers (many trees contain only one node which fits within a single subarray, requiring one access).

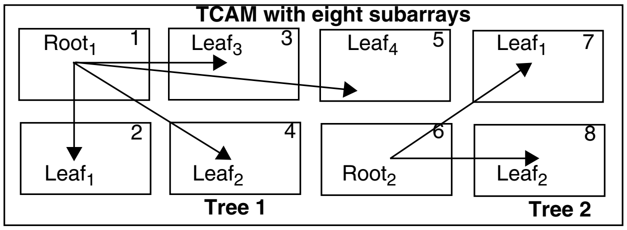

FIGURE 2. TCAM Organization Example

2.2 TCAM Organization

We first group rules into separable trees and then apply our sort-based heuristic to each tree. Each of our coarse trees is at most two deep with a root and leaves. The root node holds the interval, or range, in the cut dimension corresponding to each leaf while each leaf holds its rules. For each separable tree, we map the root and each leaf into a distinct TCAM subarray, similar to partitioned TCAMs. Figure 2 shows two trees mapped to eight TCAM subarrays. We place the rules in priority order in each subarray (we describe the layout details in Section 3.2). TreeCAM incurs some range expansion for the ranges in the root, as do all TCAMs for the classifier fields that specify ranges (e.g., the source port and protocol fields). Fortunately, because the total number of ranges in the root, which equals the number of leaves, is far fewer than the rules (e.g., 25 leaves for 100,000 rules assuming a *binth* of 4K), this expansion adds little to the total space. Further, because our cuts are at the start points. which fall at prefix boundaries, our cuts naturally avoid further range expansion (thus, no compounding of rule replication and range expansion).

An incoming packet looks up each tree's root subarray where the matching entry provides the corresponding leaf's subarray number (see Figure 2). The packet then looks up the leaf's subarray to find the highest-priority match in the tree. An external hardware logic, then determines the highest-priority match over all the trees. Modern partitioned TCAMs may pipeline the tree traversal for high bandwidth and our approach does not hinder such pipelining.

Finally, using separable trees introduce some fragmentation in TreeCAM. The root of each separable tree typically has far fewer entries than a subarray. However, to achieve high packet throughput, multiple packets must be able to lookup different trees at the same time. Therefore, we cannot map multiple root nodes of different trees to the same

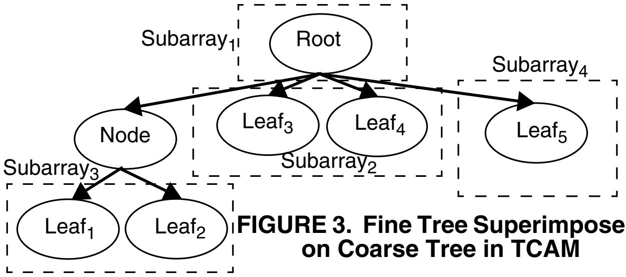

FIGURE 3. Fine Tree Superimposed on Coarse Tree in TCAM

TCAM subarray. Instead, we place each root node in a separate subarray, and incur some modest internal fragmentation in the subarrays (Section 5.3).

3 FINE TREE VERSION

While the coarse tree achieves lookups with only a few accesses and low power, the fine tree reduces the update effort. However, TreeCAM inherits the update overheads of both the TCAM's rule re-ordering and the decision tree's tree re-balancing. Recall that our strategy is to leverage the fine tree to reduce the TCAM's overheads and vice versa. Also recall our key observation that because leaves and rules in a leaf are non-overlapping, a leaf's rules have to be ordered in the TCAM only among each other but not with other leaves' rules. To exploit this idea, we build the fine tree with only a small *binth* (e.g., 16).

3.1 Building and Mapping the Fine Tree

We apply our sort-based heuristic from Section 2.1 to build the fine tree version of each separable coarse tree. Because of overlapping rules, partitioning the rules just once at the root, like the coarse tree, may result in the *binth* being exceeded. In such cases, we partition once more, possibly in a dimension different than the first cut. In practice, we found that two cuts are sufficient even for our 100,000-rule classifiers due to EffiCuts' separable trees.

The fine tree is used only for updates which occur at a much lower rate than packet lookups — once every few milliseconds versus once every few nanoseconds (Section 1). Consequently, we maintain the fine tree details in the slow control memory. To ensure that the coarse tree in the TCAM is updated correctly, we logically superimpose the fine tree on to the coarse tree in the TCAM ensuring that the fine tree's leaves are aligned with the coarse tree's leaves in the TCAM subarrays. Thus, each TCAM subarray, which corresponds to a coarse tree leaf, holds a set of fine tree leaves (i.e., fine tree leaves are not split across multiple coarse tree leaves). Figure 3 shows a fine tree and its superimposition on the coarse tree. *Subarray$_1$* is the coarse tree root and *Subarrays$_{2\text{-}4}$* are the coarse tree leaves. *Subarray$_2$* holds *Leaf$_1$* and *Leaf$_2$* of the fine tree; similarly, *Subarray$_3$* holds *Leaf$_3$* and *Leaf$_4$*, and *Subarray$_4$* holds *Leaf$_5$*, Book-keeping structures hold the range for each first-level node and leaf in the fine tree (if the tree is 3-deep) and for each leaf, a pointer to the TCAM subarray that holds the leaf.

A key point is that because of using separable trees, our rule replication is low despite the fine cuts to achieve the small *binth*. However, some rule replication is inevitable which raises a correctness issue. All replications in the fine tree must be reflected in the TCAM which has to hold the replicas even if the fine-tree leaves causing the replication map to the same TCAM subarray (in which case the replicas would be present in the same subarray). Even though this redundancy is unnecessary from the point of view of packet lookups, the redundancy is needed for correct updates. This correctness constraint follows from the fact that we assume that rules in different leaves are non-overlapping and therefore, need not be ordered with respect to each other in the TCAM. If the redundancy were eliminated resulting in only one instance of a rule being stored, then the rule may overlap with other rules in many leaves, violating our assumption and requiring the rule to be ordered with the rules of many leaves. As part of re-balancing the tree, if a rule is moved into a fine-tree leaf that already holds another replica, then the redundancy can be eliminated.

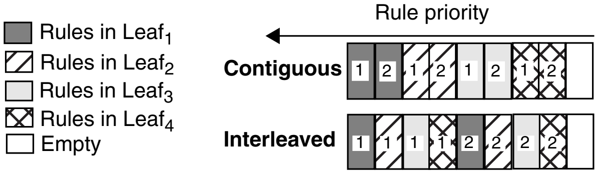

FIGURE 4. Contiguous versus Interleaved Layout

3.2 Interleaved Rule Layout in TCAM

Recall from Section 1, that conventional TCAMs physically order the rules based on priority for fast determination of the highest priority match. Therefore, any newly-added rule must be ordered correctly with respect to the other rules. In conventional TCAMs, rules are placed contiguously, leaving empty entries at the bottom. However, if we contiguously place the rules belonging to each fine tree leaf in a subarray, adding a rule may require as many moves as there are rules in the entire subarray. The well-known optimization [9] of placing some empty entries after every so many rules reduces the number of moves as long as the empty entries are not exhausted (e.g., free entries after every 10 rules would require at most 10 moves to gather one empty entry). Upon exhaustion, however, the problem resurfaces. Alternately, replenishing an empty entry as soon as it is used up would also face the problem. Thus, contiguous layout of the rules, as is done in today's TCAMs, leads to high update effort in the worst case.

We observe that contiguous layout forces many rules to be moved under the pessimistic assumption that *all* the rules in the subarray need to be ordered. Instead, we leverage the fact that a packet cannot match non-overlapping rules and that rules in different leaves cannot overlap. Therefore, *only* the rules in the same leaf need to be ordered. Accordingly, we interleave the rules from different leaves in the TCAM subarray such that the rules with the highest priority in all the leaves are in a contiguous *batch* before the batch of rules with the next priority level, and so on. Figure 4 shows contiguous and interleaved layouts for four leaves with two

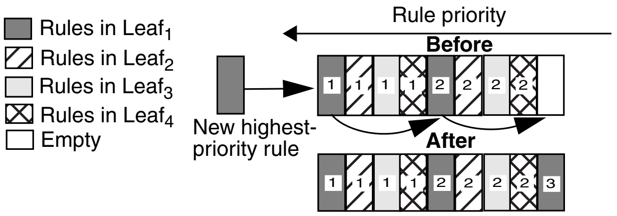

FIGURE 5. Rule Addition in Interleaved Layout

rules each where each leaf's rules are shaded differently. The numbers show the relative priority of each rule in its leaf. In interleaved layout, the four leaves result in a batch size of 4. The book-keeping data structures hold pointers to the start of each priority-level batch.

Making room in the interleaved layout for adding a rule amounts to moving an empty entry to the right priority-level batch. Assume that a new rule's priority is higher than that of an empty entry and that higher-priority rules are at the top of the subarray. Accordingly, we move each empty entry to the desired batch by first swapping the empty entry with the top-most rule of the empty entry's batch (such swapping is well known [9]). Because the top-most rule stays in its batch, in which all the rules are at the same priority level, the swap does not violate the top-most rule's priority. Then, we successively swap the empty entry with the top-most rule of every batch between the empty entry's batch and the desired batch. Continuing with Figure 4's example, Figure 5 shows the addition of a rule to *Leaf$_1$* at priority 1 by successively swapping the top-most rules of priority 1 and priority 2 batches, and an empty entry. These swaps move the top-most rule of a higher priority level to the top-most position of the next lower priority level. The moved rule abuts its batch from below, producing the effect of the rule moving from the top of its batch to the bottom and of essentially sliding the batch down by one position (as seen in Figure 5). Because the rules in a batch are at the same priority level, these swaps do not cause any priority inversion. At the same time, the number of swaps *for each empty entry* is bound by the number of batches i.e., *binth*. For the case where the new rule's priority is lower than that of an empty entry, we analogously swap the bottom-most rules of the relevant batches. Thus, interleaving is fundamental to bounding the number of rule movements to the small *binth* of the fine tree.

We briefly note that to prevent the accumulation of new leaves to hold the displaced rules, we opportunistically merge leaves that become empty upon rule deletions. Because the new leaves are in the fine tree and are recorded only in the book-keeping structures and not in the TCAM, the merging of such leaves do not affect the TCAM.

3.3 Updating the TreeCAM

Updates add or delete rules (changing a rule can be emulated by a delete followed by an add). Deletions simply invalidate the TCAM entry or entries, if there are replicas. Such invalidations leave empty entries scattered throughout the TCAM and are tracked in the book-keeping structures.

```
[1] Identify separable tree
[2] Query TCAM to get matching coarse tree leaf nodes and target
      subarrays
[3] For each target subarray, identify fine tree leaf nodes by com-
      paring rule boundaries with those of fine tree nodes
[4] If there is no space in the entire TCAM, rule cannot be added,
      else there are 3 cases:
Case I: No re-balancing: space in the target leaf
[5] Create space for the new rule by repeated swaps to move
      empty entry in to the desired batch for the new rule
[6] Add the new rule at the newly created batch
[7] Update batch boundaries
Case II: Local re-balancing: space in the target subarray
[8] Create a new leaf node for the added rule
[9] Adjust boundaries of the neighboring node and identify rules
      that are to be removed from the neighboring node
[10] Repeat steps [5] - [7] for each added rule in the new leaf and
      delete the rule from the neighboring leaf
Case III: Global re-balancing: space in some other subarray
[11] Identify subarray with empty entries and adjust boundaries
      of its neighbor that is closer to the target subarray to create
      space; Repeat [8] - [10] to add rules into the subarray
[12] Repeat [11] until there is space in the target subarray
[13] Repeat [8] - [10] at the target subarray
```

FIGURE 6. Update Algorithm for Rule Addition

We show our complete algorithm for adding a rule in Figure 6. For additions, we first identify the separable tree to which the new rule belongs. This identification is static based on fields in which the rule has wildcards. We then identify the coarse tree leaf in which the rule falls. We use a similar process to identify the target fine tree leaf. While these pre-processing steps are best done in software, we can also leverage our TCAM to optimize the steps by querying the root node in the TCAM to find the target subarray and then comparing (in software) the rule with the fine-tree leaf boundaries in the subarray (lines 1-3 in Figure 6). If the rule falls in multiple leaves then we repeat the rest of the process for each leaf. The simplicity of our sort-based heuristic makes these pre-processing steps reasonably low in effort.

Once the target fine-tree leaf is known, there are three cases in the order of increasing difficulty: (1) the target leaf has some empty entries (i.e., no re-balancing), (2) the target leaf is full but some other leaf within the same subarray has some empty entries, requiring re-distribution of the rules among the subarray's leaves (i.e., local re-balancing), and (3) the target leaf and its subarray are full but some other subarray has some empty entries, requiring re-distribution of the rules among the subarrays (i.e., global re-balancing). These cases can be ascertained by examining the book-keeping structures. The case where there are no empty entries in the entire TCAM requires a larger TCAM and cannot be addressed by re-balancing (line 4 in Figure 6).

3.3.1 No Re-balancing

In the first case, even if there are some empty entries in the leaf, the newly added rule has to be placed in a position that correctly orders the rule with respect to the other rules in the target leaf. We employ repeated swaps to move an empty entry to the desired batch as described in Section 3.2 (lines 5-7 in Figure 6). As discussed before, the number of swaps in this case is bound by the *binth*. Because each swap with an empty entry entails a read and a write, the update effort in this case is bound by 2**binth* TCAM operations.

3.3.2 Local Re-balancing

In the second case of local re-balancing, to make room for the newly-added rule in the target leaf, we choose to displace the rule closest to a boundary of the target leaf to a neighboring leaf (i.e., adjacent in the sorted order). If any other rules in the target leaf fully or partially overlap with the displaced rule then those rules are also displaced (partially overlapping rules are replicated in the new leaf). Our simulations showed that at most only two rules are displaced. Because the subspace covered by a leaf must be contiguous, the displaced rule(s) must be added to a neighboring leaf and not to an arbitrary, non-neighboring leaf that is not full. However, if we were to move the rule(s) into a neighboring leaf, then in case the neighbor is full then we would have to ripple more displacements all the way to the leaf with some empty entries. Such rippling would incur many moves because there are many fine-tree leaves per subarray (256 16-rule leaves in a 4K-entry subarray).

We avoid such high effort based on the key observation that despite all the displacements the rules stay within the subarray and, as such, packets accessing the subarray are guaranteed to search the rules. Therefore, instead of displacing the rules from the target leaf to its pre-existing neighbor, we create a new leaf adjacent to the target leaf to hold the displaced rules. This creation involves updating the fine trees to adjust the boundaries of the target leaf and create the new leaf. In Figure 7, we add R_{new} by creating $Leaf_{new}$ which corresponds to the new cut at X_{new}, and displacing R_2 to $Leaf_{new}$.

One may think that because of our above observation, there is no need to create the new leaf and even this bookkeeping effort can be avoided because packet lookups would occur correctly even without the new leaf. Such an approach, however, would essentially imply that the target leaf now has rules in excess of its capacity. The excess would grow unboundedly with future updates, forcing update effort to grow unboundedly even for the simple first case of no re-balancing. Therefore, creating the new leaf is important for bounding the update effort.

To adjust the target leaf's boundaries in the fine trees, we shrink the leaf's boundary, vacating some subspace, and create the new leaf covering the vacated subspace. In Figure 7, $Leaf_1$'s boundary shrinks from X_1 to X_{new} and $Leaf_{new}$ covers the vacated space containing R_2. All of these changes are restricted to the fine trees whereas the TCAM remains unaffected because the coarse-tree leaf boundaries have not changed. There is, however, one change to the TCAM: To avoid unbounded increase in update effort as described above, the rules belonging to new leaf should be moved from TCAM entries corresponding to the target leaf to the entries corresponding to the new leaf. In doing so, we need to ensure that the rules are ordered correctly with respect to the other rules in the entire subarray. Fortunately, because the new leaf covers the subspace vacated by the original target leaf, the new leaf does not overlap with (1) the modified target leaf or (2) any other leaf in the subarray. While the first part of this claim is obvious, the second part is true because the other leaves do not overlap with the original target leaf which fully contains the new leaf. Therefore, the rules in the new leaf need be ordered only with respect to each other but not with rules in the other leaves (lines 8-10 in Figure 6). This ordering effort is low because our interleaved layout bounds the number of swaps needed for one rule to the *binth* which is small. Because we find that at most only two rules are displaced in our simulations, we require 2**binth* swaps to add the displaced rules in the new leaf, and another *binth* swaps to add the new rule. Further, we require two TCAM invalidations (two writes) to remove the displaced rules from the target leaf. Therefore, the update effort in this case is 6**binth*+2.

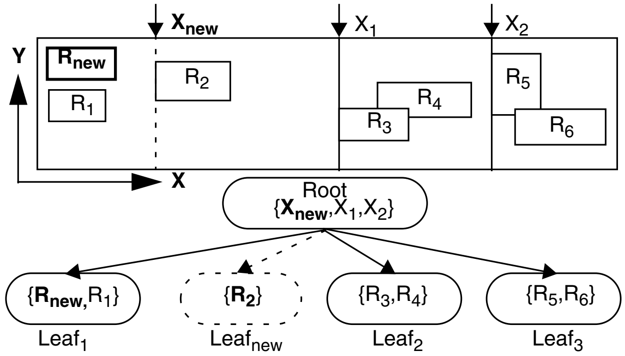

FIGURE 7. Local Re-balancing

3.3.3 Global Re-balancing

The third case of global re-balancing, where both the target leaf for the newly added rule and the target leaf's subarray are full but another subarray has some empty entries, requires the empty entries to be rippled to the target subarray. To this end, we displace rules from one subarray to the next, working backwards from the subarray with the empty entries to the target subarray. In Figure 8, to add R_{new} to $subarray_2$, we displace rules from $subarray_2$ to $subarray_3$ and so on. Similar to the second case, we displace rules closest to the boundaries of each subarray.At each subarray, we apply our second case to add the displaced rules (lines 11-13 in Figure 6).

The update effort in the third case is bound by *update effort per subarray* times the *number of subarrays*. Thanks to our interleaved layout, the first term depends *only* on *binth* which is small for our fine tree. The second term grows slowly with classifier size because larger TCAMs employ larger subarrays to reduce area and power overhead. Recall from Section 3.3.2 that *update effort per subarray* is 6**binth*+2 (= 50 for *binth* of 8). Assuming 25 subarrays for

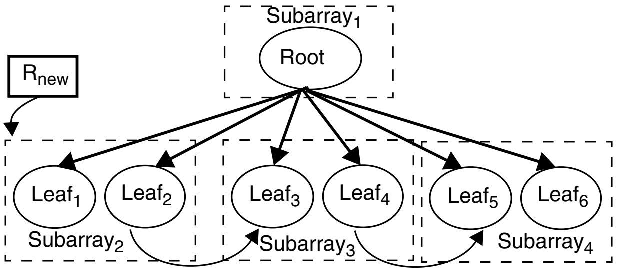

FIGURE 8. Global Re-balancing

100,000 rules, the overall update effort in the case of global re-balancing is 1250 TCAM operations. Thus, for a 100,000-rule classifier, both the common-case effort (first two cases) and the worst-case effort (third case) are low.

As we show in Section 5.2, TreeCAM's low update effort implies that lookup bandwidth lost to updates and the update latency are low (Section 1). We show in Section 5.3 that the storage overhead of our fine trees is low. In Section 3.3.5, we address the hold-up of the lookup accesses into the TCAM during the TCAM moves for updates.

3.3.4 Contrast to Other Schemes

Because of the lack of rule overlap information readily implied by decision trees (i.e., leaves do not overlap), previous TCAM schemes incur high update effort. As we discussed before, simply placing empty entries without exploiting rule overlap incurs numerous moves in the worst case of empty-entry exhaustion. While a previous TCAM scheme [11] exploits rule overlap, it requires significant effort — quadratic in the number of rules — to maintain and update overlap information. More importantly, being a TCAM-only scheme, it does not have the benefit of small, non-overlapping leaves afforded by the fine tree part of our scheme. Consequently, the sets of overlapping rules tend to be large in the scheme, which increases both the update effort in keeping the rules ordered and the book-keeping effort in maintaining overlap information. As mentioned before, Extended TCAM partitions the classifier space to reduce TCAM power but does not address updates. Applying our idea of fine partitioning (i.e., the fine tree) to reduce the scheme's update effort results in increased lookup accesses due to using the same partitioning for both lookups and updates (see Section 5.2). In contrast, our dual trees decouple lookups from updates to achieve low update effort without worsening lookup efficiency. Further, because TreeCAM does not require any batching of updates to amortize the update effort [18], TreeCAM does not incur the latency penalties of batching.

3.3.5 Path-by-path Updates

The remaining issue is that even though the update effort is low, a large classifier may incur enough TCAM moves to hold up packet lookups (e.g., 1250 TCAM operations in the worst case of global re-balancing across 25 subarrays with a binth of 8 for 100,000 rules). To address this issue, we make our updates non-atomic by balancing our tree path-by-path which involves multiple singleton rule movements with which packet lookups can be interspersed in time. To ensure that the TreeCAM is in a consistent state after each singleton movement, we update the tree path corresponding to the rule being moved.

We describe making non-atomic updates using our third case of global re-balancing, as the first two cases are subsumed by the third. In the third case, rules are displaced from one subarray to the next, rippling over multiple subarrays. We ensure that the TreeCAM is consistent after every set of rule displacements between a pair of subarrays (source and destination). We use a small, back-up TCAM to hold the set of to-be-displaced rules so that interspersed packets always search the back-up TCAM to avoid omitting the in-flight rules. The back-up TCAM holds at most as many rules as the *binth* though our simulations show that only two rules are displaced (Section 3.3.2). For each set of displacements, we perform the following steps one rule at a time:
(1) place the to-be-displaced rule into the back-up TCAM;
(2) shrink the boundary of the source subarray;
(3) remove (invalidate) the rule from the source;
(4) add the rule to the destination subarray;
(5) expand the boundary of the destination subarray; and
(6) invalidate the back-up TCAM.

Step 1 must occur first so that, because the rules have not been removed from the tree, the backing up of the rules may occur non-atomically one at a time (i.e., interspersed with packet lookups). Steps 2-3 fix the coarse tree path for the source subarray and Steps 4-5 for the destination subarray. Because the in-flight rules are backed up in Step 1, Steps 2-5 can occur in any order. Step 6 must occur after Steps 2-5.

Thanks to the back-up TCAM, most of the TCAM operations in these steps can be performed non-atomically, interspersed with packet lookups. The only atomic operations are in Step 4 where adding a rule may require moving an empty entry into the batch corresponding to the rule's priority level. This move requires a series of swaps of the empty entry and some rules in the subarray. The swapped rules are some arbitrary rules in the subarray, different from the displaced rules, and have not been backed up. Therefore, each such swap, entailing a TCAM read, a TCAM write, and a TCAM invalidation, must be atomic (i.e., cannot be interspersed with a packet lookup). Because the swaps comprise only three TCAM operations, this atomicity constraint can be satisfied easily by buffering the small number of intervening packets to avoid packet drops.

As mentioned before, CoPTUA provides non-atomic updates for conventional TCAMs. However, to provide non-atomic updates, CoPTUA worsens the common-case update effort (e.g., up to 50x more TCAM moves than conventional TCAMs). In contrast, TreeCAM leverages the fine tree to reduce the update effort enabling non-atomic updates without sacrificing the common case.

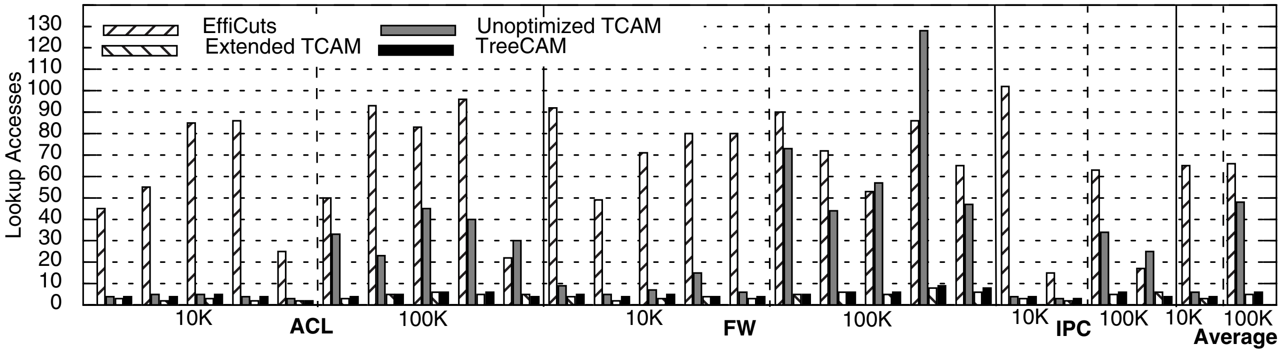

FIGURE 9. Lookup accesses for EffiCuts, Unoptimized TCAM, Extended TCAM, and TreeCAM

4 EXPERIMENTAL METHODOLOGY

Because TreeCAM uses both TCAMs and decision trees, we compare TreeCAM against modern partitioned TCAMs, and a recent decision tree proposal (EffiCuts). Because EffiCuts outperforms prior algorithmic schemes, we do not compare to those schemes. We use Extended TCAMs as our representative of partitioned TCAMs. We also show unoptimized TCAM (all the subarrays are accessed for every packet lookup) to validate our Extended TCAM implementation. We use software simulation for our evaluation. We implement EffiCuts capturing all the optimizations and heuristics. We set the parameters as per [17]: *binth* 16, *space_factor* 8, and l*argeness_fraction* 0.5 for all fields except source IP and destination IP which use 0.05.

We convert the source- and destination-port fields to prefixes in all the TCAM schemes (unoptimized TCAMs, TreeCAM, and Extended TCAM) using simple range expansion. To save implementation effort, we do not include other techniques to reduce range expansion [6]. However, our comparisons are fair because all TCAM schemes would benefit similarly from the techniques. The TCAM schemes use 4K-entry subarrays (1K-8K are common in today's products).

In Extended TCAM, a rule-grouping algorithm distributes rules to different TCAM subarrays to reduce the number of subarray accesses. We implement the algorithm and set α to 0.8 and β to 2.0, as per [12]. In addition, Extended TCAM provides support for range comparison in the TCAM cells to eliminate range expansion. Because such support will likely increase area power, and cost, we do not include this support in any of the TCAM schemes, and instead use simple range expansion. Because unoptimized TCAMs and TreeCAM would also be benefitted similar to Extended TCAM, our comparisons are fair.

We employ EffiCuts' separable trees and selective tree merging in TreeCAM. We expand the port fields during tree building. We use the same l*argeness_fraction* as EffiCuts. We set *binth* to 8 in our fine trees to achieve a good trade-off between update effort and rule replication, as we show in Section 5.4. Recall that fine-tree *binth* does not affect the number of accesses in TreeCAM, unlike EffiCuts.

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

5 EXPERIMENTAL RESULTS

Recall from Section 1 that the key metrics for packet classification are number of lookup accesses (a proxy for packet throughput), power, and update effort. We begin by comparing the number of lookup accesses per packet in TreeCAM, EffiCuts, unoptimized TCAM and Extended TCAM (Section 5.1). We use the number of accesses to compare the schemes' power. We compare the update efforts of TreeCAM and the other TCAM schemes using the number of TCAM operations per rule update (Section 5.2). To analyse the storage efficiency of TreeCAM, we present a breakdown of TreeCAM's storage overhead and compare those of with unoptimized TCAM and Extended TCAM (Section 5.3). Finally, we study TreeCAM's sensitivity to *binth* and subarray size (Section 5.4).

5.1 Lookup Accesses

Figure 9 compares TreeCAM against EffiCuts, unoptimized TCAM, and Extended TCAM in terms of the worst-case number of lookup accesses per packet. In the X axis, we show twelve classifiers — five *ACL*, five *FW*, and two *IPC*. We vary the classifier sizes as 10,000 and 100,000 within each category. The Y axis shows the worst-case number of lookup accesses per packet. For each classifier of a specific size, we show four bars, from left to right, one each for EffiCuts, unoptimized TCAM, Extended TCAM and TreeCAM. EffiCuts' and TreeCAM's lookup accesses are for all the separable trees (Section 2.1**).** We also show the average across the classifiers for each classifier size.

We see that EffiCuts requires many more lookup accesses than the TCAM schemes. Though SRAMs (used by EffiCuts) can achieve higher throughput and lower power than TCAMs, the sheer high number of accesses is likely to degrade EffiCuts' packet throughput and power without aggressive, customized pipelining which may increase complexity and cost. The number of lookup accesses in unoptimized TCAM increases linearly with the number of rules. In some of the 100K-rule classifiers, unoptimized TCAM requires as many lookup accesses as EffiCuts due to range expansion. In contrast, Extended TCAM and TreeCAM

Table 1: Update Effort of TCAM-basic, TCAM-ideal and TreeCAM for respective Worst-case Update Stream

Classifier Type	Classifier Size	TCAM-basic			TCAM-ideal			TreeCAM						
		# Empty Slots	Avg # TCAM Ops	Max # TCAM Ops	Max. Chain Length	Avg # TCAM Ops	Max # TCAM Ops	#Sub-arrays	Avg # TCAM Ops	Max # TCAM Ops	Avg rel. TCAM-basic	Max rel. TCAM-basic	Avg. rel. TCAM-ideal	Max rel. TCAM-ideal
ACL	10K	44K	954	30K	67	68	134	3	19	91	0.02	0.003	0.3	0.7
	100K	60K	5914	276K	166	166	332	19	179	684	0.03	0.002	1.0	2.5
FW	10K	68K	906	64K	90	90	180	3	25	112	0.03	0.002	0.3	0.6
	100K	82K	6774	567K	295	294	590	29	282	1069	0.04	0.002	1.0	1.7
IPC	10K	43K	582	22K	62	62	124	1	8	24	0.01	0.001	0.1	0.2
	100K	46K	6148	236K	137	134	274	11	99	385	0.02	0.002	0.7	1.3

require at most 8 accesses even for large classifiers (Extended TCAM's accesses relative to those of unoptimized TCAM are in line with [12]). These low numbers are due to the search pruning of TreeCAM's coarse tree and Extended TCAM's partitioning. While Extended TCAM is slightly better than TreeCAM on average, this difference is not significant as both schemes achieve low number of accesses. Both TreeCAM and Extended TCAM achieve 2x and 8x fewer accesses compared to unoptimized TCAM in classifiers with 10,000 rules and 100,000 rules respectively. Because TCAM power is directly proportional to the number of subarray accesses, TreeCAM and Extended TCAM achieve similar factors of power reduction over unoptimized TCAM. Because all the TCAM schemes pipeline the accesses across subarrays (Section 2.2) and employ the same subarray sizes for fair comparison, all of them would achieve the same packet throughput. While TreeCAM and Extended TCAM achieve similar number of accesses, the next section shows that reducing Extended TCAM's update effort results in high number of accesses. In contrast, both update effort and number of accesses remain low in TreeCAM.

5.2 Update Effort

We compare TreeCAM to two TCAM update schemes — *TCAM-basic* and *TCAM-ideal*. TCAM-basic places a set of empty entries for every so many non-empty entries. Because TreeCAM incurs some storage overhead over unoptimized TCAMs (Section 2.1), we include enough empty entries to make the total storage of TCAM-basic and TreeCAM equal. In TCAM-ideal, we adapt CAO_OPT which avoids TCAM priority-ordering for non-overlapping prefixes in longest prefix match [9]. We apply CAO_OPT to packet classification by using the algorithm from [11] to identify chains of overlapping rules. However, we make two ideal assumptions in TCAM-ideal: (1) Enough empty entries are always available at the end of every chain; and (2) Updates do not cause chains to merge (such merges would require all the merged chains' rules to be reordered in the TCAM which may move more rules than the longest chain [11]). While Extended TCAM does not discuss updates (the rule-grouping algorithm is not incremental), we configure the algorithm to use fine partitioning to reduce Extended TCAM's update effort (similar to our fine tree). We do not compare to EffiCuts for updates because the paper does not discuss updates and re-balancing EffiCuts' large trees is not straightforward. We do not compare to CoPTUA due to its high update effort.

Because of the lack of publicly-available typical update streams, we generate a worst-case rule update stream for a given classifier and update scheme. The worst-case update stream for TreeCAM adds rules to the left-most leaf of the largest separable tree (Section 2.1) and removes rules from the right-most leaf of the tree. The worst-case update stream for TCAM-basic adds high-priority rules and removes low-priority rules. The sizes of sets of empty and non-empty entries in TCAM-basic do not impact the update effort for such a stream [9]. The worst-case update stream for TCAM-ideal adds rules to the smallest chain of overlapping rules, and removes rules from the longest chain. For each stream, we compute the maximum and average number of TCAM operations per rule update.

In Table 1, we compare TCAM-basic, TCAM-ideal, and TreeCAM in terms of maximum and average number of TCAM operations per rule update for the worst-case update stream. In addition, we show the number of empty entries for TCAM-basic, the length of the longest chain of overlapping rules which is relevant for TCAM-ideal, and the number of subarrays in the largest tree for TreeCAM. All the schemes incur two operations (a read and a write) to move a rule. In TCAM-basic, (1) the average number of moves equals the length of contiguous non-empty entries and (2) the maximum number of moves equals the total number of rules with range expansion (as stated in [9]). In TCAM-ideal, (1) the average number of moves equals half of the longest chain length and (2) the maximum number of moves equals the longest chain length. Note that the maximum chain length is high in TCAM-ideal due to transitive dependencies (i.e., if rule *R1* overlaps with a lower-priority rule *R2*, and if *R2* overlaps with a much lower priority rule *R3*,

Table 2: Processing Overhead per Rule Addition

ACL (µs)	**FW** (µs)	**IPC** (µs)
37	57	38

then *R1* needs to be ordered with *R3*, even though *R1* does not overlap with R3). In TreeCAM, we compute the average number of moves by adding one rule which causes one rule to be displaced at each subarray for half the subarrays. Similarly, we compute the maximum number of moves by adding as many rules as the replication factor, which is two for TreeCAM, resulting in two rules to be displaced at each subarray for all the subarrays. In each subarray, a new or displaced rule takes at most 17 operations (8 swaps and 1 invalidation) (Section 3.3.2). At the right end of the table, we show TreeCAM's number of operations relative to those of TCAM-basic and TCAM-ideal.

Because TCAM-basic does not leverage non-overlapping rules, it incurs significantly more operations than TCAM-ideal and TreeCAM (see *Avg # TCAM Ops* and *Max # TCAM Ops* for TCAM-basic, TCAM-ideal, and TreeCAM in Table 1). TreeCAM performs close to TCAM-ideal, showing that TreeCAM's fine tree fully exploits non-overlapping rules (in some cases TreeCAM is better because of limiting the chain of overlapping rules to the *binth*).

Assuming update rate of 10 per ms and packet rate of 1 per 10 ns (Section 1) where each packet lookup makes 8 accesses in TreeCAM (Section 5.1) and each update makes 1069 TCAM operations (the highest among *Max # TCAM Ops* for TreeCAM in Table 1), TreeCAM consumes less than 2% of TCAM bandwidth for updates. Assuming each TCAM operation takes 25 ns, update latency is less than 40 µs. Because our non-atomic updates require only a single swap to be atomic (Section 3.3.5), TreeCAM requires only a few packets to be buffered.

The above analysis does not include processing overhead, including accesses to TreeCAM's fine trees. We show TreeCAM's processing overhead per rule addition for 100,000-rule classifiers of each type in Table 2. We run our update algorithm on AMD Opteron machines with 12 MB of L2 cache, and 8G of memory. After cache warm up, our fine trees easily fit in today's CPU caches (We show the storage overhead of our fine trees in Section 5.3). Also, because of the close correpondence between individual coarse tree (TCAM) and fine tree operations, the fine tree accesses are similar in number to TCAM operations in Table 1. Overall, our processing overhead fits well within the update interval of 100 µs (10 updates per 1 ms).

Finally, we run Extended TCAM's rule-grouping algorithm with a subarray size (and *binth*) of 8 rules, which limits makes the update effort same as that for TreeCAM. However, the finer partitions result in more accesses per packet. In Table 3, we show the number of accesses for this Extended TCAM variant and TreeCAM (repeated from Figure 9) averaged over 100,000-rule classifiers of each type. Because Extended TCAM's single partitioning for both lookups and updates does not decouple the two, reducing the update effort ends up degrading lookup efficiency. In contrast, TreeCAM's dual tree versions enable this decoupling to perform well in both lookups and updates.

Table 3: Lookup Accesses in Finely Partitioned Extended TCAM and TreeCAM

Scheme	ACL	**FW**	**IPC**	**Avg.**
Ext. TCAM	35	43	15	35
TreeCAM	5	7	5	6

5.3 Storage Overhead

We compare the storage overheads of TreeCAM and Extended TCAM over unoptimized TCAM. There are three sources of storage overhead in any TCAM scheme: range expansion, rule replication, and fragmentation of TCAM subarrays due to coarse-tree roots in TreeCAM (Section 2.2) and the index tables in Extended TCAM. While range expansion is common to all the TCAM schemes, rule replication and fragmentation occur only in Extended TCAM and TreeCAM. The overall overhead is the product of these three components. In Table 4, the columns show the relevant components for each scheme and the overall overhead of Extended TCAM and TreeCAM relative to unoptimized TCAM. The rows show the average across the classifiers of a type and size. At the bottom, we also show the average (geometric mean) across classifiers of the same size.

From Table 4, we observe that firewall classifiers incur higher range expansion than the other classifier types. All the schemes incur the same range expansion. Specifically, TreeCAM does not incur additional expansion because our sort-based heuristic aligns the cuts with prefix boundaries (Section 2.1). We see that both Extended TCAM and TreeCAM incur modest rule replication. Extended TCAM's rule-grouping algorithm and TreeCAM's separable trees reduce overlapping rules, thereby eliminating rule replication. Both Extended TCAM and TreeCAM incur higher fragmentation overhead in 10,000-rule classifiers than in 100,000-rule classifiers. TreeCAM's sort-based heuristic partitions the rules equally among subarrays without adding fragmentation overhead beyond that of Extended TCAM. The overhead is lower in 100,000 rule classifiers because the additional subarrays are a smaller fraction of all the subarrays. Lower overheads for larger classifiers is a good trend because the absolute TCAM sizes are larger for larger classifiers which is where overhead matters. Both Extended TCAM and TreeCAM incur 35% average storage overhead over unoptimized TCAM, which is reasonable given their 8x fewer accesses (lower power).

We show the storage overhead of TreeCAM's fine trees in Table 5 for 100,000-rule classifiers of each type. We require 16 bytes per rule, and 64 bytes per internal node for node boundaries and pointers. Assuming no rule replication and no range expansion, the size of our fine trees with 125

Table 4: Storage Overheads of Extended TCAM and TreeCAM

Classifier	Size	Range Expansion	Replication Overhead		Fragmentation Overhead		Overhead over UnOpt. TCAM	
			Ext. TCAM	TreeCAM	Ext. TCAM	TreeCAM	Ext. TCAM	TreeCAM
ACL	10K	1.61	1.05	1.21	3.30	2.99	3.47	3.62
	100K	1.59	1.04	1.08	1.40	1.30	1.46	1.40
FW	10K	3.13	1.12	1.08	3.02	3.16	3.38	3.41
	100K	3.32	1.13	1.04	1.40	1.24	1.58	1.29
IPC	10K	1.16	1.02	1.01	4.02	4.59	4.01	4.64
	100K	1.17	1.01	1.00	1.29	1.36	1.30	1.36
Average	10K	1.80	1.06	1.10	3.42	3.51	3.63	3.86
	100K	1.84	1.06	1.04	1.36	1.30	1.44	1.35

nodes for 100,000 rules with *binth* of 8 is 125*64+100,000*16 ~= 2 MB. Replication and range expansion increase the overhead further, as shown in Table 5. Firewall classifiers exhibit slightly higher overhead than the others due to range expansion. However, overall our fine tree sizes are small, and fit in the control memory of today's routers. The remaining overhead due to the bookkeeping data structures (for tracking empty entries and linking fine tree leaves to subarrays) is negligible.

5.4 Sensitivity

We analyse TreeCAM's sensitivity to *binth* and TCAM subarray size. We observe that *binth* and subarray size do not affect the number of accesses required for packet match. *Binth* affects storage overhead (rule replication) and update effort (number of overlapping rules). Smaller values of *binth* reduce update effort at the cost of increased rule replication. Rule replication does not change for *binth* greater than eight. However, there is a sharp increase in rule replication for *binth* less than eight. Subarray size affects update effort, but does not significantly impact storage overhead.

In Figure 10, we show how update effort varies with subarray size. Because the update effort for smaller classifiers is low, we only show the sensitivity results for 100,000-rule classifier from each classifier type along the X-axis. We vary subarray size as 2K, 4K and 8K. The Y-axis shows the number of TCAM operations per rule update, normalized to those for TreeCAM with 4K subarrays (default). We show both the average and maximum number of TCAM operations for the worst-case update stream described in Section 5.2. The number of operations decreases almost linearly with subarray size. Thus, the current trends of larger classifiers and TCAMs are favorable for TreeCAM.

In Figure 11, we show how update effort varies with *binth*. Along the X-axis, we show one typical 100,000-rule classifier from each category. We vary *binth* as 8, 16 and 32. Along the Y-axis, we show the number of TCAM operations per rule update normalized to those for TreeCAM, with *binth* = 8 (default). We show both the average and maximum number of operations. The number of operations increases with larger *binth*. Because we assume that all the rules in a fine-tree leaf are overlapping, increasing *binth* increases the update effort.

Table 5: Storage Overhead of TreeCAM's Fine Trees

ACL (MB)	**FW** (MB)	**IPC** (MB)
4	8	3

6 RELATED WORK

A plethora of TCAM and algorithmic approaches have been proposed for packet classification. Targeting updates, Chain Ancestor Ordering [9] exploits non-overlapping rules to reduce ordering overhead for longest prefix match. Another scheme [11], which we have previously discussed, proposes an algorithm to find chains of overlapping rules and to assign priorities to the rules on the basis of the position in the chain. During packet lookups, the scheme performs a binary search over priority levels, looking up the entire TCAM at each step, to find the highest-priority match. However, long chains in large classifiers incur many full-TCAM accesses per lookup. We have also discussed CoPTUA which provides non-atomic updates. Other work [6] reduces range expansion in TCAM by representing ranges as ternary values instead of prefixes.

Among the partitioned TCAM schemes, we have previously discussed Extended TCAM. CoolCAM [20] reduces power for longest prefix match by partitioning the classifier via either bit selection or prefix tries. These schemes do not address updates.

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 [5] produce large bit vectors encoding match on each dimension which are then ANDed. Cross-producting approaches [3]

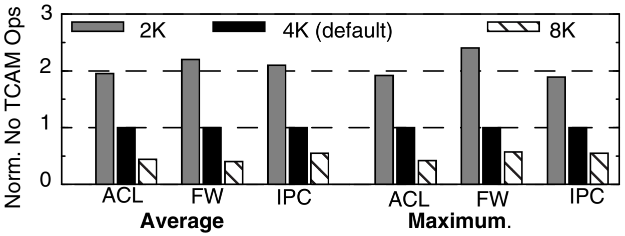

FIGURE 10. Update Effort versus Subarray Size

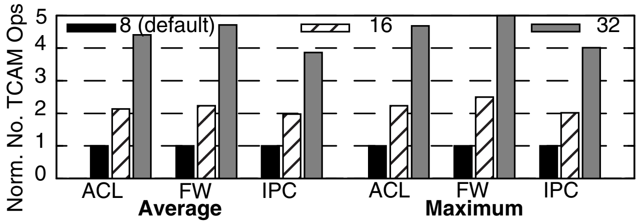

FIGURE 11. Update Effort versus *binth*

look up first per-dimension tables for all the dimensions in parallel and then one or more cross-product tables to combine the first step's results. Grid-of-tries schemes [1] employ tries to prune the cross product of source IP and destination IP fields and then linearly searches the matching rules. However, the bit vectors, the cross-product tables, and the tries grow rapidly with classifier size (prohibitive for 100,000 rules) [15]. Another approach [14] compresses the cross-product tables which requires them to be searched instead of being indexed, increasing accesses. Tuple-search [13] partitions the classifier based on prefix lengths. Packets hash on the prefix bits to determine the matching partition which is linearly searched, However, unconstrained partition size and hash collisions results in unpredictable performance. We have previously discussed EffiCuts which builds on HiCuts and HyperCuts. Other approaches [8][19] optimize decision trees but do not address updates.

7 CONCLUSION

Previous TCAM-based and algorithmic schemes for packet classification do not perform well in both lookup efficiency (number of memory accesses and power) and update effort. We proposed TreeCAM which performs well in both lookups and updates via three novel ideas. *Dual versions* of TreeCAM's decision tree decouple lookups and updates. A coarse version with a few thousand rules per leaf achieves efficient lookups and a fine version with a few tens of rules per leaf reduces update effort. Combined with the fine version's few rules per leaf, *interleaved layout* of the rules in the TCAM enables us to bound our worst-case update effort. *Path-by-path updates* enables update work to be interspersed with packet lookups (i.e, non-atomic updates), eliminating packet buffering or packet drops during updates.

Using simulations of 100,000-rule classifiers, we showed that TreeCAM performs well in both lookups and updates: (1) 6-8 TCAM subarray accesses per packet matching modern TCAMs, (2) close to the worst-case update effort of an idealized TCAM while requiring little buffering of packets. By performing well in both metrics of lookup efficiency and update effort, TreeCAM provides a scalable approach to packet classification. With increasing classifiers size and line rates, TreeCAM will likely be an attractive option.

ACKNOWLEDGMENTS

We thank our shepherd, Simon Leinen, and the anonymous reviewers for their valuable comments.

REFERENCES

[1] F. Baboescu, S. Singh, and G. Varghese. Packet classification for Core Routers: Is there an alternative to CAMs? In *Proceedings of IEEE INFOCOMM '03*, pages 53 – 63 vol.1, 2003.

[2] Cisco. Personal communication. Dec. 2010.

[3] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM '99*, pages 147 – 160, 1999.

[4] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *Micro, IEEE*, 20(1):34 – 41, 2000.

[5] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM SIGCOMM '98* , pages 203 – 214, 1998.

[6] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proceedings of ACM SIGCOMM '05*, pages 193 – 204, 2005.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[8] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *Proceedings of IEEE INFOCOMM '09*, pages 648 – 656, 2009.

[9] D. Shah and P. Gupta. Fast updating algorithms for TCAMs. *IEEE Micro*, 21:36–47, 2001.

[10] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proceedings of ACM SIGCOMM '03*, pages 213 – 224, 2003.

[11] H. Song and J. Turner. Fast filter updates for packet classification using TCAM. In *Proceedings of IEEE GLOBECOM '06*, pages 1 –5, 2006.

[12] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended TCAMs. In *Proceedings of IEEE International Conference on Network Protocols - 2003*, pages 120 – 131, 2003.

[13] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proceedings of ACM SIGCOMM '99*, pages 135 – 146, 1999.

[14] D. Taylor and J. Turner. Scalable packet classification using distributed crossproducing of field labels. In *Proceedings of IEEE INFOCOMM '05*, pages 269 – 280 vol. 1, 2005.

[15] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37:238–275, September 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] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In *Proceedings of ACM SIGCOMM 2010*, pages 207–218, 2010.

[18] Z. Wang, H. Che, M. Kumar, and S. Das. Coptua: Consistent policy table update algorithm for TCAM without locking. *Computers, IEEE Transactions on*, 53(12):1602 – 1614, 2004.

[19] T. Woo. A modular approach to packet classification: algorithms and results. In *Proceedings of IEEE INFOCOMM '00*, pages 1213 – 1222 vol.3, 2000.

[20] F. Zane, G. Narlikar, and A. Basu. Coolcams: power-efficient TCAMs for forwarding engines. In *Proceedings of IEEE INFOCOM 2003*, pages 42 – 52 vol.1, 2003.