Hierarchical Hybrid Search Structure for High Performance Packet Classification

Oğuzhan Erdem
Electrical and Electronics Engineering
Middle East Technical University
Ankara, TURKEY 06510
Email: ogerdem@metu.edu.tr

Hoang Le, Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, USA 90007
Email: {hoangle, prasanna}@usc.edu

***Abstract*—Hierarchical search structures for packet classification offer good memory performance and support quick rule updates when implemented on multi-core network processors. However, pipelined hardware implementation of these algorithms has two disadvantages: (1) backtracking which requires stalling the pipeline and (2) inefficient memory usage due to variation in the size of the trie nodes.**

We propose a clustering algorithm that can partition a given rule database into a fixed number of clusters to eliminate backtracking in the state-of-the-art hierarchical search structures. Furthermore, we develop a novel ternary trie data structure (T_ϵ). In T_ϵ structure, the size of the trie nodes is fixed by utilizing ϵ-branch property, which overcomes the memory inefficiency problems in the pipelined hardware implementation of hierarchical search structures. We design a two-stage hierarchical search structure consisting of binary search trees in Stage 1, and T_ϵ structures in Stage 2. Our approach demonstrates a substantial reduction in the memory footprint compared with that of the state-of-the-art. For all publicly available databases, the achieved memory efficiency is between 10.37 and 22.81 bytes of memory per rule. State-of-the-art designs can only achieve the memory efficiency of over 23 byte/rule in the best case. We also propose a SRAM-based linear pipelined architecture for packet classification that achieves high throughput. Using a state-of-the-art FPGA, the proposed design can sustain a 418 million packets per second throughput or 134 Gbps (for the minimum packet size of 40 Bytes). Additionally, our design maintains packet input order and supports in-place non-blocking rule updates.

I. Introduction

With the rapid growth of the Internet, the design of high speed packet forwarding engines has been a major challenge. Advances in optical networking technology are pushing link rates beyond OC-768 (40 Gbps). Such high rates demand that packet processing in routers must be performed in hardware [6].

Most hardware-based solutions for high speed packet classification fall into two main categories: ternary content addressable memory (TCAM)-based and dynamic/static random access memory (DRAM/SRAM)-based solutions. Although TCAM-based engines can retrieve search results in just one clock cycle, their throughput is limited by the relatively low speed of TCAMs. They are expensive and offer little adaptability to new addressing and routing protocols [1]. Since SRAM-based solutions utilize some kind of tree traversal, multiple cycles are required to process a single packet. Therefore, pipelining techniques have been employed to improve the throughput. As a result, the new incoming packet does not need to wait for the previous one to finish its whole search process. The key issues in designing an architecture for IP packet header classification are (1) size of supported ruleset, (2) high throughput, (3) scalability, and (4) incremental update. To address these challenges, we propose and implement a high-throughput and memory-efficient SRAM-based linear pipelined architecture for packet classification. This paper makes the following contributions:

- A clustering algorithm that partitions a given rule database into a fixed number of sets to eliminate backtracking in the state-of-the-art hierarchical search structures (Section IV-B).
- A special type of ternary trie data structure (T_ϵ) and a two-stage hierarchical search structure that achieve substantial memory saving in hardware implementation (Section IV-C).
- A linear multi-pipelined SRAM-based architecture using the proposed hierarchical search structure that can be easily implemented on hardware (Section VI). Our design achieves a memory efficiency between 10.37 and 22.81 bytes of memory per rule, and sustains a high throughput of 418 million packets per second on a state-of-the-art FPGA device (Section VII).

The rest of the paper is organized as follows. Section II covers the background and related work. Section III presents the definitions and notations used in the paper. Section IV details the proposed data structures and clustering algorithm. Section V describes the design methodology and optimizations. Section VI introduces the proposed architecture and presents implementation results. Section VIII concludes the paper.

II. Background

A. Packet classification overview

Packet classification is an essential part of a full-featured network router. It enables the router to support firewall processing, Quality of Service (QoS) differentiation, virtual private networks, policy routing, and other value-added services.

Supported by the U.S. National Science Foundation under grant No. CCF-1018801. Equipment grant from Xilinx is gratefully acknowledged.

An IP packet can be classified based on 5-tuple header rules (i.e. Source IP Address, Destination IP Address, Protocol, Source Port, and Destination Port), in which fields are generally specified by prefixes and ranges. When a packet arrives at a router, its header is compared against a set of rules, often known as a ruleset or filter database. Each rule can have one or more fields and an associated action to be taken if matched. A packet is considered matching a rule only if it matches all the fields within that rule. In a network router, the matching result guides the forwarding decision. In a network firewall, packets are dropped (silently discard) or rejected (discard with "error responses" sent to the sources) when a match is detected. A sample ruleset is shown in Table I. The terms *ruleset* and *filter database* are used interchangeably in this paper.

TABLE I: Sample 5-field ruleset

Rule	SA	DA	SP	DP	PRTCL	Priority	Action
R_1	0*	10*	80	*	TCP	1	Act0
R_2	0*	01*	17	17	UDP	2	Act1
R_3	0*	1*	44	*	TCP	2	Act2
R_4	00*	1*	17	44	UDP	3	Act3
R_5	00*	11*	*	100	TCP	4	Act4
R_6	10*	1*	*	*	*	5	Act5
R_7	*	00*	*	*	TCP	5	Act6
R_8	0*	10*	*	100	TCP	6	Act7
R_9	0*	1*	*	*	TCP	7	Act8
R_{10}	0*	10*	17	17	UDP	7	Act9
R_{11}	111*	000*	80	*	TCP	8	Act10

B. Prior work

Existing packet classification approaches can be classified into four main groups: (1) exhaustive search, (2) decomposition, (3) decision tree, and (4) hierarchical-trie (H-trie).

In exhaustive search, all the entries in a ruleset are analyzed [18], [20]. The two basic approaches in this group are linear search and TCAM based parallel search. Linear search is performed by comparing the header of a packet with all the entries in a ruleset sequentially. Since linear search is a slow process for large rulesets, it is popular for the final stage of a search when the set of possible matching rules has been reduced to a bounded constant [2], [5]. In TCAM-based approach, the header of a packet is compared with all the entries in parallel. TCAMs has several disadvantages such as high cost, storage inefficiency, high power consumption, and limited scalability to long input keys.

In decomposition based solutions, independent searches on each header field are performed, then the results are combined [4], [11], [3], [19], [23]. These approaches offer high throughput but require high amount of storage space in order to aggregate the results of single searches efficiently. The primary challenge for these approaches is how to efficiently aggregate the results of the single field searches.

Most of the proposed packet header classification algorithms and architectures are based on decision trees, which take the geometric view of the packet classification problem. HiCuts [5] and its enhanced version HyperCuts [16] are representatives of such algorithms. At each node of the decision tree, the search space is cut based on the information from one or more fields in the rule. HiCuts builds a decision tree using local optimization decisions at each node to choose the next dimension to test, and how many cuts to make in the chosen dimension. The HyperCuts algorithm allows cutting on multiple fields per step, resulting in a fatter and shorter decision tree. The common problem of these approaches is that it is difficult to support incremental updates.

Hierarchical-trie (H-trie) is built using the source address (SA) and destination address (DA) prefixes. Initially, a SA trie is constructed using all the SA prefixes. For each prefix node in SA trie, a DA trie is constructed using DA prefix(es) associated with that SA prefix. Thus, the structure consists of a large SA trie and hierarchically connected multiple small DA tries. Search starts from the SA trie. If a prefix node of the SA trie is visited, then the corresponding DA trie connected to that prefix node is traversed. Even though a match can be found at any node in the DA trie, search has to **backtrack** to the SA trie and continue the search to find other possible matches. The search terminates after a leaf node in the SA trie is visited. Set-pruning trie eliminates the backtracking by replicating the rules [22]. Grid-of-tries (GoT) [19] data structure for 2-field packet classification eliminates the backtracking by introducing switch pointers to some trie nodes and hence each rule is stored in only one node. Despite of good memory efficiency, it is not clear how to extend the GoT to support multiple fields.

The authors in [2] presented the Extended Grid-of-tries (EGT) to improve the previous idea by supporting multiple fields without using many instances of the data structure. The authors replaced the switch pointers with jump pointers. All matching filters are traversed during the search. A node in an EGT has a pointer to the list of rules. Whenever a matching node is reached, a linear search is performed in the final rule list. Although EGT has good memory performance, high number of worst-case memory accesses decreases the search time performance. These designs suffer from backtracking, which makes hardware implementation difficult and inefficient.

III. Definitions and Notations

Definition ***Prefix node*** in a trie is any node to which a path from the root of the trie corresponds to a SA or DA prefix of a rule. Other nodes are called ***non-prefix node***. If a prefix node is a leaf node then it is called ***leaf prefix node***, otherwise ***non-leaf prefix node***.

Definition Two prefixes *x*, *y* are said to be ***disjoint*** if *x* is not a proper prefix of *y* and *y* is not a proper prefix of *x*. A set of prefixes in which all the prefixes are pairwise disjoint is called a ***disjoint prefix set***.

Definition Two rules are said to be ***overlapped*** if they have the same SA and DA prefixes. For instance, R_1, R_8 and R_{10} in Table I are overlapped. A node that stores overlapped rules is called ***supernode***.

Definition ***Memory efficiency*** is defined as the average amount of memory (in bytes) required to store a filter rule.

Table II shows the list of notations used throughout the paper.

TABLE II: List of notations used in the paper

Notation	Meaning
SA	Source IP address
DA	Destination IP address
SP	Source port number
DP	Destination port number
$PRTCL$	Protocol name
S	Set of SA prefixes
N	Number of prefixes in S
R	Number of clusters
S_i	Set of SA prefixes in cluster i, $0 \leq i < R$
P_{trie}	Upper bound for the number of rules per trie node
SV	Skip value used in path compression
BS	Bit string used to store missing bits in path compression
ϵ_b	Upper bound for the number of consecutive ϵ transitions
α	Ratio of the number of rules in secondary memory
α_T	Upper bound for α

IV. Data Structure and Algorithms

A. Motivation

While hierarchical trie structures can be efficiently deployed on multi-core network processors, their hardware implementations have two issues: (1) backtracking and (2) memory inefficiency.

As previously mentioned, in Hierarchical-trie (H-trie), each prefix node of SA trie is hierarchically connected to a destination address prefix trie (DA trie) in the second stage. An H-trie is composed of a large SA trie and multiple small DA tries. If there is a match at a prefix node X in the SA trie then the corresponding DA trie is traversed. Note that search in the SA trie is paused during this time. Once completed the DA trie, search has to **backtrack** to the prefix node X in the SA trie to find other possible matches. All matches must be found, and the match with a highest priority is returned. In hardware implementations, backtracking requires stalling the pipeline and feedback loops to return the intermediate results from each DA trie search. Figure 1 illustrates the backtracking while searching a 5-tuple IP packet header ($SA = 000$, $DA = 110$, $SP = 44$, $DP = 100$, $PRTCL =$ TCP) in a H-trie structure constructed using the rules in Table I. Although the packet header initially matches R_5, search needs to backtrack to the SA trie to find the highest priority match. In the figure, the red lines show the backtracking paths. Consequently, R_9 is selected as the highest priority match among all the matches (R_5, R_3 and R_9).

Hardware implementation of an H-trie structure also suffers from the **memory inefficiency**. This is due to the variable number of rules stored in each node. Analysis of the available rulesets shows that the number of rules stored in a trie node varies from 0 to 73 [21]. In hardware implementation, the size

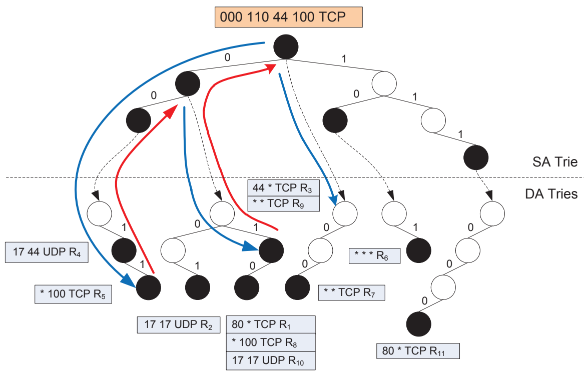

Fig. 1: Backtracking in H-trie structure

of a node is determined by that of the largest node, leading to inefficient memory and resource usage. For instance, the hardware implementation of the H-trie shown in Figure 1 requires a memory space for 17×3 rules (the largest node has 3 rules and DA tries includes 17 nodes). However, only 21.5% (11 rules) of the total space is used.

In this paper, we propose a clustering algorithm, named *recursive leaf extraction*. The algorithm eliminates the backtracking from DA tries to the SA trie in the hierarchical structure, while generating a fixed number of clusters. Furthermore, we propose a 2-stage memory-efficient hierarchical data structure that can easily be implemented in hardware. The proposed architecture is called $Tree - Trie_\epsilon$ (TT_ϵ) hierarchical search architecture.

B. Recursive Leaf Exraction (RLE) for Fixed-Set Rule Clustering

The *recursive leaf extraction* algorithm partitions a given ruleset based on the SA field to eliminate backtracking from a DA structure to the SA structure. The algorithm takes a set of prefixes S and the number of clusters R as its inputs, and generates a collection of non-empty subsets $\{S_i\}$, $(0 \leq i < R)$. Each subset is called a *cluster*. In each cluster, the SA prefixes are pairwise disjoint.

Initially, RLE algorithm builds a uni-bit trie using the SA prefixes. Then, the recursive leaf extraction process is performed. Each recursive Step i of the algorithm includes two sub-steps: (1) the leaf nodes are removed from the trie and moved into set $\{S_i\}$ and (2) the non-prefix leaf nodes are trimmed off the trie.

In the RLE algorithm, input R is the number leaf extraction steps, and it also specifies the number of clusters. Our algorithm repeats the leaf extraction process $R - 1$ times. In the last step, leaf pushing is applied to the remaining trie. The pseudo-code is given in Algorithm 1.

Figure 2 demonstrates the recursive leaf extraction process using the SA prefixes from Table I. In our example, R is chosen as 3. In the last step of the algorithm, only the default prefix (P=*) remained; hence, leaf pushing is not required.

Time Complexity: In the first step of the algorithm, a binary trie is built from a given prefix set S consisting of N prefixes.

Algorithm 1 Clustering algorithm

```
Input: Prefix set $S$, Number of clusters $R$.
Output: A partition of $S$ into a collection of non-empty prefix
    subsets such that within each subset all the prefixes are
    pairwise disjoint
1: $i = 0$
2: Construct a binary trie using prefix set $S$.
3: while $i < R - 1$ do
4:     Move the leaves of the trie into $S_i$
5:     Trim the leaf-removed trie
6:     $i = i + 1$
7: end while
8: Leaf-push the trie and move the leaf-pushed leaves into
   $S_i$
9: return $\{S_i\}$, $0 \leq i < R$
```

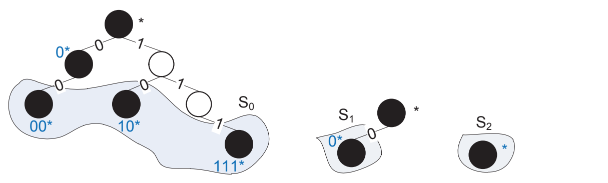

Fig. 2: Recursive leaf extraction using SA prefixes from Table I

The complexity of this step is $O(N)$. In each step of the algorithm, the leaf prefixes are moved into Set S_i, and the remaining trie is trimmed. This step is executed $R-1$ times. The complexity of each step is $O(N)$. In the final step, the trimmed trie is leaf-pushed and the leaves are moved into Set S_{R-1}. Since all the steps are sequential and R is an input constant, the overall complexity of recursive leaf extraction algorithm is $O(N)$.

C. $Tree - Trie_\epsilon$ (TT_ϵ) Search Structure

We propose a hierarchical search structure which consists of 2 stages:

Stage 1 **(SA tree)**: A binary search tree (BST) is built for each cluster using the SA prefixes. In the BST, each node includes: (1) a value (prefix), (2) a prefix length, (3) left pointer, and (4) right pointer. The left subtree of a node contains only values that are less than or equal to the value stored in that node. The right subtree contains values that are greater than the value. Prior to constructing the BST, prefixes are sorted into an array in ascending order. Given the sorted array of prefixes, the BST is constructed by picking the correct prefix (pivot) as root, and recursively building the left and right subtrees. Note that short prefixes are extended to the length of the longest prefix by appending with '0' bits. In our structure, the prefixes in any cluster are pairwise disjoint; hence, extending the prefixes to a same length should not cause overlapping.

Figure 3 illustrates a TT_ϵ data structure for the given set of rules in Table I. In this example, $R = 3$. Algorithm 1

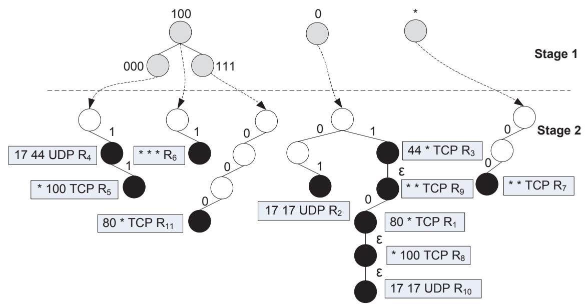

Fig. 3: TT_ϵ data structure for the ruleset in Table I

partitions the SA prefixes into 3 sets (or clusters). For each set S_i, a binary search tree is constructed in the first stage of hierarchical data structure. To further improve memory efficiency, each set has its own maximum prefix length, depending on the length of the longest prefix in the set. In our example, the lengths in each cluster are chosen as 3,1 and 0 for Set S_0, S_1 and S_2, respectively. The shorter prefixes are extended by appending with 0s.

Stage 2 **(DA trie)**: Each node of the SA trees connects to a DA trie. Therefore, in each cluster, the number of DA tries equals to the number of SA prefixes. Each prefix node of a DA trie stores at least one rule. For each rule, only the SP, DP, PRTCL, and Priority fields are stored.

In our approach, the overlapped rules are stored in the DA trie nodes rather than pointing to a list of these rules as in [2]. Therefore, the matching results can be resolved at each node. However, the number of rules stored at each node is not constant. This results in memory inefficiency for hardware implementation. To improve memory efficiency, we propose a special type of ternary trie data structure, called T_ϵ.

1) T_ϵ Trie Structure: A single node in T_ϵ may have (1) a single *epsilon* (ϵ) branch for which no input bit is consumed or (2) '0' and/or '1' branch (same as binary trie), but can not have (1) and (2) at the same time. The main goal of utilizing the ϵ transition is to split a supernode in a trie into multiple small and fixed size nodes. These nodes are sequentially connected by the ϵ branches.

In the T_ϵ data structure, we set a limit on the number of overlapped rules per node, denoted by P_{trie}. If the number of overlapped rules in a supernode is M, then this supernode can be represented by $\lceil M/P_{trie} \rceil$ nodes. Each node contains P_{trie} rules, except for the last node. In Figure 3, Stage 2 illustrates the T_ϵ trie structures generated using the DA prefixes of the rules in Table I. In our example, $P_{trie} = 1$. Thus, a supernode, which has overlapped rules R_1, R_8 and R_{10}, is represented by three nodes connected by the ϵ transitions. The pseudo-code to construct T_ϵ is given in Algorithm 2.

2) Path Compression (PC) Algorithm: Our T_ϵ data structure can also be used with path compression technique [13]. Each non-prefix node of a trie can be removed if it has only one child. Path compression helps shorten the path to a prefix node from the root. In order to keep the record of the removed internal nodes, each node must store a skip value (*SV*) and a

Algorithm 2 T_ϵ construction algorithm

```
Input: Prefix table $T$ consisting of prefixes $\{P_i\}$, $0 \leq i < N$
    with associated nexthop info $NHI_i$
Input: Root node $P_{root}$ of $T_\epsilon$, $P_{root}.left = NULL$,
    $P_{root}.right = NULL$, $P_{root}.size = 0$
Input: Maximum node size, $P_{trie}$
Output: $T_\epsilon$ trie structure
 1: $i = 0$
 2: while $i \leq N$ do
 3:     Binary_trie_insert ($P_i$, $P_{root}$)
 4:     Let $P_{node}$ be a node where $P_i$ is stored
 5:     if $P_{node}.size < P_{trie}$ then
 6:         Store $NHI_i$ to $P_{node}$, $P_{node}.size = P_{node}.size + 1$
 7:     else
 8:         if $P_{node}$ has $\epsilon$ branch ($P_{node}.left = \epsilon$ branch,
            $P_{node}.right = 0$) then
 9:             $P_{node} = P_{node}.left$
10:             Go to Step 5.
11:         else
12:             Create a node $P_{new}$
13:             $P_{new}.left = P_{node}.left$
14:             $P_{new}.right = P_{node}.right$
15:             $P_{node}.left = P_{new}$
16:             $P_{node}.right = 0$
17:             Store $NHI_i$ to $P_{new}$, $P_{new}.size = P_{new}.size + 1$
18:         end if
19:     end if
20:     $i = i + 1$
21: end while
22: return $T_\epsilon$ $trie$
```

bit string (*BS*). The skip value stores the number of bits to be skipped on the path. The bit string stores the missing bits from the last skip operation. The memory efficiency performance of the algorithm highly depends on the trie structure. The performance increases when the trie is sparse. Otherwise the algorithm may not save memory because of the overhead, which comes from the book-keeping of the removed internal nodes. Figure 4 illustrates the path compressed TT_ϵ data structure for the rule set shown in Table I.

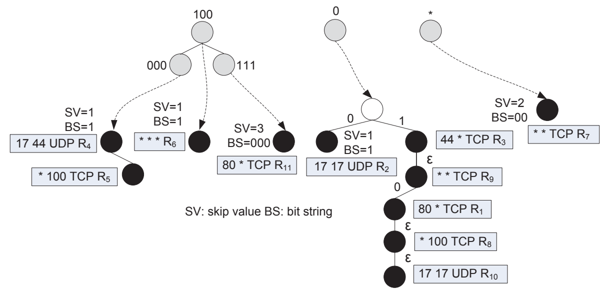

Fig. 4: Path-compressed TT_ϵ data structure for the ruleset in Table I

D. Packet Classification Algorithm

For each incoming packet, all the 5 fields (SA, DA, $PRTCL$, SP, DP) are extracted from the header and forwarded to all the clusters. A traversing frame includes seven fields: (1) source address (*SA*), (2) destination address (*DA*), (3) source port (*SP*), (4) destination port (*DP*), (5) protocol id (*PRTCL*), (6) the latest matched rule Flow ID (R_{ID}), and (7) the latest matched rule priority number (R_{pri}). Search is performed in all the clusters in parallel, and is carried out as follows:

SA tree search: Search in SA tree starts from the root node and is simply performed by traversing left or right, depending on the comparison at each node. Search ends when the source address matches with the value in a node, or when a leaf node is reached. If a match is found in any node, then search continues to traverse the corresponding T_ϵ structure in the next stage, otherwise it is terminated without a match.

DA trie search (Path compressed T_ϵ structure): Search uses the destination IP address to traverse the DA trie, similarly to that in a traditional binary trie. However when a node with an ϵ branch is reached, no address bit is inspected and the search passes directly to the next node. A node in T_ϵ has two pointer fields (left and right child pointers) as in the binary trie; however, in case of the ϵ branch, only one of pointer fields is used and the other can be filled by all '0' bits. If a node has no ϵ branch, then the direction of traversal is determined by the most significant bit of destination address (left if 0 or right otherwise). In each node, the source port number *SP*, the destination port number *DP* and the protocol *PRTCL* of the packet header are compared with the corresponding fields of the rules stored in that node, in parallel. If a match occurs and the priority value of the matched rule is higher than that of the last match (R_{pri}), then the corresponding action (R_{ID}) and priority (R_{pri}) fields of a traversing frame are updated.

If path compression is employed, then the skip value (SV) and the bit string (BS) in each trie node are also examined. The destination address is left-shifted accordingly by $SV + 1$ bits. Search operation in the T_ϵ structure terminates when a leaf node or a null pointer is reached.

The search results from all the clusters are compared based on their priority value. The matching rule with the highest priority is returned as the final result.

Claim: *Backtracking* is not needed in the search using the proposed TT_ϵ data structure.

Proof: The proposed clustering algorithm ensures that all the SA prefixes within any cluster are disjoint. Thus, at most one match is possible in each SA tree. Therefore, once the search leaves Stage 1 (SA tree) for Stage 2 (DA tries), it does not have to jump back to the SA tree.

E. Rule Updates

Rule updates include (1) rule insertion, (2) rule deletion, and (3) action ID changes. To simplify updating operations, a shadow SA trie is maintained. This shadow trie is used to find the correct cluster on which the update operation will be performed and thus rebuilding the full trie is not necessary.

Each update operation requires modifications in SA tree and DA tries separately. Prefix insertion and deletion in a binary search tree are covered in detail in [12]; hence, these details are skipped in this paper. We only focus on the update operations in the DA tries (T_ϵ structure).

Rule insertion: The rule insertion requires a prefix insertion into the SA tree and DA tries. The complexity of update operations in a binary trie, is $O(W)$, where W is the maximum length of a prefix (32 for IPv4 and 64 for IPv6). However, the prefix insertion in a T_ϵ structure needs to check the size of the prefix node before placing. As described in Algorithm 2, if a node, in which the new rule will be stored, has more than P_{trie} rules, then the rule is inserted into the next node which is connected by an ϵ branch. If such a node does not exist, a new node with ϵ transition is created. Even though generation of such a node in TT_ϵ is easy in multicore implementations, it requires shifting of trie levels in a pipelined implementation. Once the correct prefix node is located, the associated DP, SP, PRTCL, Priority and the Action ID of a new rule are finally stored in this node.

Rule deletion: In rule deletion, the target SA and DA prefixes are located and their valid bits are reset. However, the prefix deletion in T_ϵ structure may also lead to shifting of prefixes, which is a performance bottleneck in pipelined hardware implementations.

Action ID change: The target rule is located and its Action ID is simply updated.

V. Design Methodology and Optimizations

A. Design Methodology

The TT_ϵ data structure for packet classification can easily be implemented in hardware. Our analysis using the publicly available real rulesets shows that the number of pipeline stages (SA tree + DA trie) in the TT_ϵ architecture is at most 55 (Section VII). However, the size of a single supernode may increase due to rule updates. If the size of a supernode exceeds the threshold P_{trie}, then new trie node(s) with ϵ transition(s) must be inserted to the trie data structure. This also leads to increase in the number of pipeline stages of the architecture. Any changes in the hardware configuration while searches are running may decrease the performance of the architecture substantially. To solve this problem, we propose two methods:

1) A limit on the number of consecutive ϵ transitions (ϵ_b) is set while the number of overlapped rules allowed per node (P_{trie}) is relaxed. If M is the number of overlapped rules in a supernode, then this supernode can be represented by at most $(\epsilon_b + 1)$ nodes, which are sequentially connected using ϵ transitions. In this case, any supernode is allowed to store at most $\lceil M/(\epsilon_b + 1)\rceil \geq P_{trie}$ rules. This approach sets an upper bound for the number of stages in a pipeline in return for increase in the memory requirement since P_{trie} is relaxed for some stages.
2) To use the available memory space efficiently and fix the number of stages, the two parameters P_{trie} and ϵ_b can be used together. However, in this case a secondary search structure is required to store the rules which exceed the predefined limit. In hardware implementation, SRAM is used to store the large primary data structure and the small secondary structure can be implemented using TCAM. If M denotes the number of overlapped rules in a supernode, then this supernode can be represented by at most $(\epsilon_b + 1)$ nodes. Each node can contain at most P_{trie} rules. In this case, if $M > (P_{trie} \times (\epsilon_b + 1))$ then the number of excessive rules, $M - (P_{trie} \times (\epsilon_b + 1))$, will be moved to the secondary storage.

Several design problems can be formulated to choose the design parameters P_{trie} and ϵ_b based on the given SRAM and secondary memory (TCAM) sizes. The best SRAM memory efficiency can be achieved by choosing the smallest values for these parameters. However, in this case, the largest amount of secondary memory is required. Let α denote the ratio of the number of rules stored in the secondary storage over the total number of rules of the given ruleset S. The following design strategies are possible:

1) Given a limit for the number of consecutive ϵ transitions ϵ_b and an $\alpha_T \geq 0$ as the input threshold, choose P_{trie} such that $\alpha \leq \alpha_T$. Let P_{trie_max} be the maximum number of rules that can be stored at each node over the entire structure. First, P_{trie_max} can easily be determined by building the TT_ϵ for the given ruleset with $\alpha = 0$, and finding the maximum number of rules per node over the entire TT_ϵ. If the number of rules in the largest node is M_{max}, then $P_{trie\_max} = \lceil M_{max}/(\epsilon_b+1)\rceil$. Secondly, we independently vary P_{trie} from 1 to P_{trie_max}. For each $(P_{trie},\ \epsilon_b)$ pair, a TT_ϵ is generated, and the resulting memory efficiency and α are calculated. A design such that $\alpha \leq \alpha_T$ is selected. In the case that more than one design satisfies the constraint, the design with the highest memory efficiency (or lowest memory requirement) is chosen.
2) Alternatively, ϵ_b can be optimized such that $\alpha \leq \alpha_T$ for a given P_{trie} and an $\alpha_T \geq 0$ as the input threshold. Let ϵ_{b_max} be the maximum number of consecutive ϵ transitions that are allowed in our structure. First, ϵ_{b_max} can easily be determined by building the TT_ϵ for the given ruleset with $\alpha = 0$. Then, we vary ϵ_b from 1 to ϵ_{b_max}. Similar to above design approach, for each $(P_{trie},\ \epsilon_b)$ pair, a TT_ϵ is generated and the resulting memory efficiency and α are calculated. The $\alpha \leq \alpha_T$ providing the highest memory efficiency is selected.
3) P_{trie} and ϵ_b can also be optimized together to minimize the size of the secondary memory for a given SRAM size M_{SRAM}. First, P_{trie_max} and ϵ_{b_max} can be determined for $\alpha = 0$. Secondly, we decrease these parameters and calculate the required SRAM size for each $(P_{trie},\ \epsilon_b)$ pair. When M_{SRAM} is reached, then the α which guarantees the smallest size of the secondary memory for the given SRAM size is returned.

We utilized design strategy 2 in this paper while describing the architecture and implementation. In our design, we set

$\alpha_T = 0$ and optimize ϵ_b for given P_{trie} values. By setting $\alpha_T = 0$, we aimed to find the smallest ϵ_b value which guarantees no secondary storage required.

B. Optimization in clustering algorithm

Our clustering algorithm partitions a given prefixes into a collection of non-empty clusters $\{S_i\}$, $0 \le i < R$ (Section IV-B). In each cluster, the prefixes are pairwise disjoint. Our clustering algorithm can further be extended to reduce the negative effect of some large sized supernodes over the performance. The algorithm can be relaxed to allow some prefixes to be resided in multiple clusters as long as the rule of pairwise disjoint prefix set is not violated. As seen in Figure 2, the prefixes 10* and 111* in S_0 can also be replicated in S_1. By doing so, the prefixes in each cluster will still remain pairwise disjoint. By using this extension, some large sized supernodes can be represented by smaller sized supernodes in multiple clusters.

VI. Architecture and Implementation

A. Architecture

Figure 5 describes the overall architecture of the proposed TT_ϵ engine for packet classification. Pipelining is used to improve the throughput. SA and DA denote pipelines for the source address prefix trees and destination address prefix tries, respectively. The number of pipelines depends on the number of clusters. The number of stages in each pipeline is determined by the height of the data structure used. Delay blocks are added at the end of the shorter path to match the latency of the pipelines.

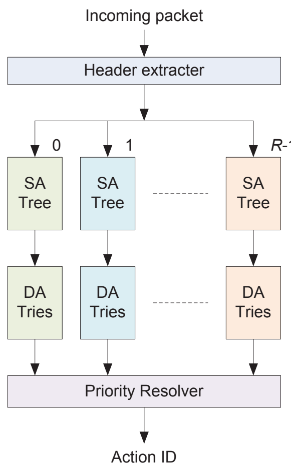

Fig. 5: Multi-pipeline TT_ϵ architecture

The header fields of packet is extracted from the incoming packet. These values are then routed to all pipelines and searches are performed in parallel. The results are fed through a priority resolver to select the highest priority match.

B. Implementation

The architecture is configured as dual linear pipelines to double the search rate. At each stage, the memory has dual Read/Write ports so that two packets can be input every clock cycle.

Figure 6 presents a single pipeline stage for a DA trie with $P_{trie} = 1$. Each stage includes a match module and a rule table stored on SRAM. There are seven inputs: (1) destination address (*DA*), (2) destination port number (*DP*), (3) source port number (*SP*), (4) protocol number (*PRTCL*), (5) action ID of latest matched rule, (6) priority value of latest matched rule and (7) the memory address, which is used to retrieve the node stored in SRAM. The memory address is forwarded from the previous stage.

Each entry in SRAM consists of 11 data fields: (1) destination port number higher range value (DP_{high}), (2) destination port number lower range value (DP_{low}), (3) source port number higher range value (SP_{high}), (4) source port number lower range value (SP_{low}), (5) protocol number ($prtcl$), (6) priority value of rule, (7) action ID associated with the rule, (8) left child node address, (9) right child node address, (10) skip value used for path compression, and (11) bit string to keep the record of removed nodes while path compression. If any stage is optimized for $P_{trie} > 1$, then a SRAM entry in that stage includes total of $P_{trie} \times 11$ data fields.

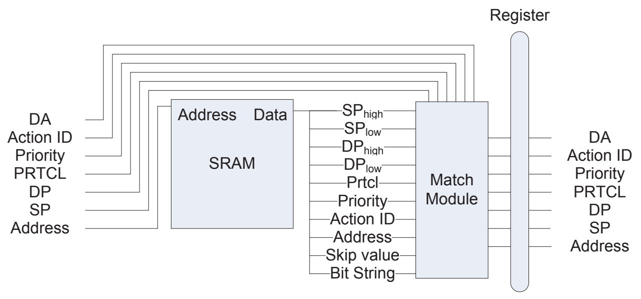

Fig. 6: A basic stage of the DA pipeline

At each stage, the *DA* bits are used to determine the direction of traversal. Then, the memory address is updated and the *DA* is left-shifted (by $SV+1$ times if path compression is employed). If ϵ transition is encountered, then search is forwarded to the next stage without checking and shifting *DA* bits. If there is a match in any stage and the priority value of the current rule is higher than the priority value of the last matched rule, then the action ID and priority value are updated by the match module.

VII. Performance Evaluation

A. Experimental Setup

Fifteen real rulesets were collected from class-bench [21]. There are three different types of rule sets: Access Control List (ACL), Firewall (FW), and IP Chain (IPC). Each group has 5 filter sets with different sizes. The size of the sets range from 100 to 10K rules.

B. Memory requirement

We first applied our proposed fixed-set clustering algorithm to each rule set. Our algorithm takes the SA prefixes of

TABLE III: Memory efficiency (Bytes per rule) for various rulesets

1	2	3	4	5	6	7	8	9	10	11
Ruleset	N	S_0	S_1	S_2	S_3	TT_ϵ	EGT [2]	HyperCuts [16]	BV [11]	Hybrid scheme [9]
ACL	752	679	16	55	2	22.08	25.41	32.58	71.80	N/A
ACL100	98	85	3	10	0	21.42	27.69	27.78	47.35	24.44
ACL1K	916	798	28	85	5	22.20	24.96	38.15	91.63	22.98
ACL5K	4415	3927	279	201	8	21.44	24.87	59.64	257.23	24.83
ACL10K	9603	8416	496	684	7	22.62	30.23	54.22	789.22	25.51
FW	269	93	6	1	169	10.48	25.31	399.18	40.72	N/A
FW100	92	28	2	2	60	10.37	23.42	113.37	27.46	56.63
FW1K	791	268	23	12	488	10.41	23.80	6110.58	67.08	215.06
FW5K	4653	1554	264	34	2801	13.27	39.04	16132.65	691.69	255.13
FW10K	9311	3611	28	0	5672	14.51	49.45	12554.18	1582.18	248.54
IPC	1550	1207	201	13	129	21.64	26.63	128.52	61.57	N/A
IPC100	99	65	18	3	13	18.34	31.60	24.57	69.16	23.65
IPC1K	938	730	101	47	60	20.05	29.95	61.34	176.03	25.63
IPC5K	4460	3763	218	147	332	21.56	27.62	406.80	358.61	49.46
IPC10K	9037	7659	491	331	556	22.81	28.92	2378.35	788.69	43.30

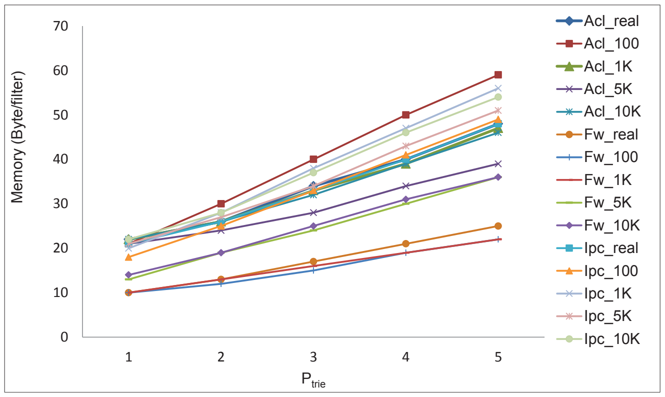

Fig. 7: Memory requirement for various P_{trie} values

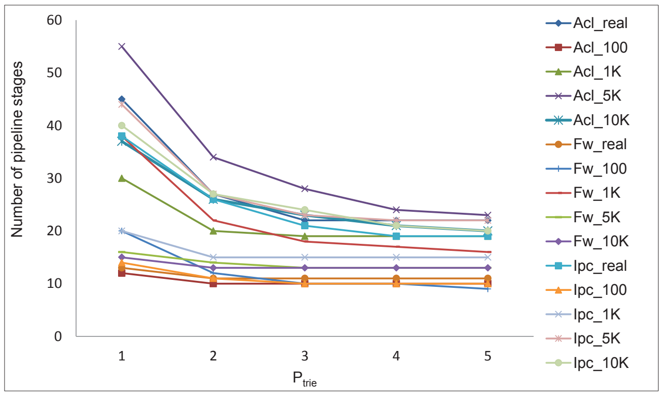

Fig. 8: The number of pipeline stages for various P_{trie} values

rules and the parameter R as inputs, and partitions the input ruleset into R clusters. We observed that leaf pushing is not required for any ruleset for $R = 4$, because set S_3 includes only default source address prefixes (SA=*) in the last step of our algorithm. Table III presents the numerical results of our clustering algorithm for the given 15 rulesets. The second column gives the total number of rules in each set. Columns 3-6 shows the resulting sets and the number of rules in each set.

P_{trie} provides the tradeoff between the memory requirement of TT_ϵ and the number of pipeline stages. If P_{trie} increases, then the memory requirement increases. However, the number of required pipeline stages decreases. On the other hand, small value of P_{trie} provides better memory efficiency for TT_ϵ, while increasing the number of stages. Figure 7 and 8 show the memory requirement (bytes per filter) and the number of pipeline stages for each ruleset [21] with various values of P_{trie}, respectively. Note that the number of pipeline stages observed in our experiments guarantees no TCAM is required ($\alpha_T = 0$).

In Table III, Column 7 shows the memory requirement for each rule set as $P_{trie} = 1$. Columns 8-11 show the results of the existing approaches for the same rulesets. All results are presented in the number of bytes per rule.

C. Throughput

We implemented our proposed hardware design in Verilog, using Xilinx ISE 12.4, with Xilinx Virtex-5 XC5VFX200T (-2 speed grade) as the target. The architecture supports the largest ruleset ACL10K consisting of 9603 rules. The post place and route results show a minimum clock rate of 4.785 ns, or a maximum frequency of 209 MHz. Using dual-ported memory, the design can support 418 million packets per second (MPPS), or 134 Gbps. Throughout this paper, throughput in Gbps is calculated based on a minimum packet size of 40 bytes (or 320 bits).

D. Performance Comparison

In Table IV, the performance of TT_ϵ is compared with the state-of-the-art packet classification approaches in terms of the memory efficiency (byte/rule), throughput (Gbps), and throughput efficiency (Gbps/byte) (the ratio of the throughput to the memory efficiency). The results for the existing designs

TABLE IV: Performance comparison

1	2	3	4	5	6
Packet classification engines	Platform	# of rules	Memory efficiency (Bytes/rule)	Throughput (Gbps)	Throughput efficiency (Gbps/B)
TT_ϵ	FPGA	9603	22.62	134.0	5.92
Hybrid scheme [9]	FPGA	9603	25.51	80.00	3.14
Optimized HyperCuts [8]	FPGA	9603	63.73	80.23	1.26
Simplified HyperCuts [10]	FPGA	10000	28.60	10.84	0.38
BV-TCAM [17]	FPGA	222	72.07	10.00	0.14
2sBFCE [14]	FPGA	4000	44.50	2.06	0.05
Memory-based DCFL [7]	FPGA	128	1726.56	24.00	0.01
B2PC [15]	ASIC	3300	163.63	13.60	0.08

were reported in [9]. Note that all the designs have been implemented on a Xilinx Virtex-5 device for fair comparisons. Columns 7-11 in Table III show that, our scheme exhibits the superior memory efficiency, compared with the existing approaches for the same rulesets. Furthermore, the variation of the memory efficiency in TT_ϵ is smaller than that of the other solutions. Hence, the memory efficiency of the proposed TT_ϵ is less sensitive to the size and type of the supported rulesets. Table IV gives the throughput and throughput efficiency of the state-of-the-art hardware-based packet classification engines. Column 5 shows that our design achieves the highest throughput performance among all the existing architectures. Our scheme also outperforms all the existing schemes with respect to the throughput efficiency, as shown in Column 6.

VIII. Conclusion

In this paper, we proposed Hierarchical TT_ϵ structure for packet classification. Our approach eliminates the need for backtracking in hardware implementation and achieves substantial memory saving. Furthermore, we designed and implemented a high-throughput, linear pipelined architecture to support the proposed data structure on FPGAs. The number of memory accesses is fixed in contrast to the existing approaches. Using a state-of-the-art Field Programmable Gate Arrays (FPGA), the proposed architecture achieves a sustained throughput of 418 million lookups per second. One of the drawbacks of the design is the potentially long latency for rulesets with large number of overlapped rules. Another drawback is the memory efficiency of the design can be negatively affected by new rule updates. In this case, the design parameters need to be recalculated to improve the memory efficiency. In the future, we plan to explore the variable node size and other optimization techniques to further improve the memory efficiency. We also plan to reduce the power consumption and extend the data structure to support packet classification with more number of fields, such as OpenFlow.

References

[1] F. Baboescu, S. Rajgopal, L. Huang, and N. Richardson. Hardware implementation of a tree based IP lookup algorithm for oc-768 and beyond. In *Proc. DesignCon '05*, 2005.

[2] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to cams. In *In IEEE INFOCOM*, 2003.

[3] F. Baboescu and G. Varghese. Scalable packet classification. *IEEE/ACM Trans. Netw.*, 13:2–14, February 2005.

[4] P. Gupta and N. McKeown. Packet classification on multiple fields. *SIGCOMM Comput. Commun. Rev.*, 29:147–160, August 1999.

[5] P. Gupta and N. Mckeown. Packet classification using hierarchical intelligent cuttings. In *in Hot Interconnects VII*, pages 34–41, 1999.

[6] P. Gupta and N. McKeown. Algorithms for packet classification. *Network, IEEE*, 15(2):24 –32, April 2001.

[7] G. S. Jedhe, A. Ramamoorthy, and K. Varghese. A scalable high throughput firewall in fpga. In *Proceedings FCCM*, pages 43–52, Washington, DC, USA, 2008. IEEE Computer Society.

[8] W. Jiang and V. K. Prasanna. Large-scale wire-speed packet classification on fpgas. In *Proceedings of the FPGA 2009*, FPGA '09, pages 219–228, New York, NY, USA, 2009. ACM.

[9] W. Jiang and V. K. Prasanna. Scalable packet classification: Cutting or merging? In *Proceedings of the ICCCN '09*, ICCCN '09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.

[10] A. Kennedy, X. Wang, Z. Liu, and B. Liu. Low power architecture for high speed packet classification. In *ANCS'08*, pages 131–140, 2008.

[11] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28:203–214, October 1998.

[12] H. Le and V. Prasanna. Scalable high throughput and power efficient ip-lookup on fpga. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 167 –174, april 2009.

[13] D. R. Morrison. Patriciapractical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15:514–534, October 1968.

[14] A. Nikitakis and L. Papaefstathiou. A memory-efficient fpga-based classification engine. *Proceedings FCCM*, 0:53–62, 2008.

[15] I. Papaefstathiou and V. Papaefstathiou. Memory-efficient 5d packet classification at 40 gbps. In *Proceedings INFOCOM*, pages 1370 –1378, May 2007.

[16] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. ACM.

[17] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 238–245, New York, NY, USA, 2005. ACM.

[18] 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, pages 120–, Washington, DC, USA, 2003. IEEE Computer Society.

[19] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28:191–202, October 1998.

[20] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37:238–275, September 2005.

[21] D. E. Taylor and J. S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15:499–511, June 2007.

[22] P. Tsuchiya. A search algorithm for table entries with non-contiguous wildcarding. Unpublished report. Bellcore.

[23] J. van Lunteren and T. Engbersen. Fast and scalable packet classification. *Selected Areas in Communications, IEEE Journal on*, 21(4):560 – 571, may 2003.