FPGA-Based Updatable Packet Classification Using TSS-Combined Bit-Selecting Tree

Yao Xin, Wenjun Li, Guoming Tang, *Member, IEEE*, Tong Yang, Xiaohe Hu, and Yi Wang

***Abstract*— OpenFlow switches are being deployed in SDN to enable a wide spectrum of non-traditional applications. As a promising alternative to brutal force TCAMs, FPGA-based packet classification is being actively investigated. However, none of the existing FPGA designs can achieve high performance on both search and update for large-scale rule sets. To address this issue, we propose TcbTree, an FPGA-based algorithmic scheme for packet classification. Specifically, at the algorithmic side, i) a two-stage framework consisting of heterogeneous algorithms is proposed, where most rules can be mapped into several balanced trees without rule replications, ii) for the remaining few rules, a centralized TSS (Tuple Space Search) architecture together with a real-time feedback scheme is designed to enhance the efficiency of TSS search on FPGA, and iii) a tree dilution method is designed to equalize rule distribution in trees, so that the latency of tree search can be reduced. At the hardware side, i) an efficient data structure set is designed to convert tree traversal to addressing process, which breaks the constraints of limited tree depth and imbalanced node distribution, and ii) distinct from fully pipelined designs, multiple levels of parallelism are efficiently explored with multi-core, multi-search-engine and coarse-grained pipelines herein. Experimental results using ClassBench show that, with the implementation of TcbTree on FPGA, the average classification throughputs for 1k, 10k, 32k and 100k rule sets achieve 788.8 MPPS, 404.3 MPPS, 237 MPPS and 41.8 MPPS, respectively, and the update throughput for all benchmark rule sets is above 1 MUPS.**

***Index Terms*— OpenFlow, packet classification, FPGA, algorithmic.**

Manuscript received June 29, 2021; revised January 13, 2022 and May 16, 2022; accepted June 1, 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. Giaccone. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101130003, in part by NSFC under Grant 62102203 and Grant 61872212, in part by the National Key Research and Development Program of China under Grant 2020YFB1806400, in part by the Basic Research Enhancement Program of China under Grant 2021-JCJQ-JJ-0483, in part by the China Postdoctoral Science Foundation under Grant 2020TQ0158 and Grant 2020M682825, in part by the International Postdoctoral Exchange Fellowship Program of China under Grant PC2021037, in part by the National Key Research and Development Program of China under Grant 2019YFB1802600, in part by the Major Key Project of Peng Cheng Laboratory (PCL) under Grant PCL2021A02 and Grant PCL2021A08, and in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2019B1515120031. This paper was presented in part at the ACM/IEEE ANCS, Cambridge, U.K., September 25, 2019 [1] [DOI: 10.1109/ANCS.2019.8901884]. *(Corresponding author: Wenjun Li.)*

Yao Xin and Guoming Tang are with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: xiny@pcl.ac.cn; tanggm@pcl.ac.cn).

Wenjun Li is with the School of Engineering and Applied Sciences, Harvard University, Allston, MA 02134 USA, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: wenjunli@seas.harvard.edu).

Tong Yang is with School of Computer Science, Peking University, Beijing 100871, China (e-mail: yang.tong@pku.edu.cn).

Xiaohe Hu is with Department of Automation, Tsinghua University, Beijing 100084, China (e-mail: hxhe@mail.tsinghua.edu.cn).

Yi Wang is with the Institute of Future Networks, Southern University of Science and Technology, Shenzhen 518055, China, also with the Peng Cheng Laboratory, Shenzhen 518055, China, and also with the Heyuan Bay Area Digital Economy Technology Innovation Center, Heyuan 517000, China (e-mail: wangy@pcl.ac.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2022.3181295, provided by the authors.

Digital Object Identifier 10.1109/TNET.2022.3181295

I. INTRODUCTION

AS ONE major building block of Software-Defined Networking (SDN), OpenFlow has been widely deployed for a wide spectrum of non-traditional applications, such as flexible resource partitioning and real-time migration [2]. An OpenFlow switch applies multiple flow tables for packet *match-action* lookups and forwarding policies, which essentially falls into a multi-field packet classification problem [3]. Although having been investigated for two decades, this problem encounters new challenging requirements with OpenFlow switches nowadays, including large-scale rule set support and dynamic rule update. Furthermore, in support of the search function, OpenFlow puts forward higher requirements than that in traditional 5-tuples, e.g., with more than 12 fields in OpenFlow 1.x standard [4] or arbitrary number of fields in P4 [5].

For the implementation of packet classification in commercial OpenFlow switches, Ternary Content Addressable Memory (TCAM) is a widely adopted solution due to its line-speed classification [6]. However, TCAM is expensive, area inefficient, and power hungry, because of its dense and parallel circuitry in hardware design [7]–[12]. Worse still, in function design it does not support *range match* which is required in many packet classification scenarios [13]–[15]. Also, modern TCAM suffers from a high complexity of rule update [16]–[18].

Under such circumstances, efficient algorithmic solutions using generic memories instead of TCAMs to facilitate efficient packet classification are becoming revitalized, such as decision tree and Tuple Space Search (TSS) [19]–[27]. Among the platforms of algorithm execution, Field Programmable Gate Arrays (FPGAs) have been actively investigated for line-speed packet classification over the past decade, due to its ability to reconfigure and to offer massive parallelism [28]–[37].

The existing FPGA-based packet classifications can be roughly divided into two categories: decomposition approaches and decision tree approaches. For the first category, bit-vector (BV) decomposition can achieve decent performances at both packet classification and rule updating [28], [38]–[40]. Nevertheless, the scale of applied vectors in BV decomposition is restricted by the FPGA logic resource, as it consumes a large amount of distributed RAMs. Thus only *small-scale rule sets* can be supported in these solutions. For the second category of decision tree approaches, although they do not have the restriction of rule set scale, two major problems affect their scalability to OpenFlow applications:

i) they could *hardly support dynamic rule update* (i.e., without pre-computing the memory content or rebuilding the tree/subtree) due to the serious rule replications; ii) most decision tree approaches are with imbalanced and unbounded depth, which not only results in *inefficient optimizations* on FPGA but also causes the fully pipelined design *a high degree of dependence* on specific rule sets. If setting equally allocated memories along the pipeline stages to deal with the imbalanced depth, it would result in a waste of storage resources.

In this paper we present **TcbTree**, an FPGA-based updatable packet classification solution using **T**SS-**c**ombined **b**it-selecting **Tree**. TcbTree could achieve high performances on both table lookups and rule updates with our particular algorithmic and hardware designs. **From the algorithmic aspect**, TcbTree adopts a two-stage framework for packet classification. In the first stage, several balanced bit-selecting trees are constructed from rule subsets grouped with respect to their *small fields*. This grouping eliminates wildcard (*) at a set of most significant bits in *small fields*, thereby getting rid of the trouble of rule replications. The second stage handles the terminated nodes from the trees constructed in the first stage, where wildcards may lead to serious rule replications. Thanks to the efficient rule grouping and balanced bit selection, the number of rules in the last layer of tree leaves is significantly reduced, where linear search can be well applied to these rules to facilitate tree constructions. Besides decision tree, Priority Sorting Tuple Space Search (PSTSS) [26] is adopted to deal with the remaining few rules.

Nevertheless, when implementing TcbTree on FPGA **from the hardware aspect**, we are faced with two challenges. Firstly, the PSTSS scheme is hardware unfriendly and resource consuming. Secondly, the *big leaf* problem (i.e., the number of rules within a leaf is much larger than a predefined threshold value) causes over-dispersion of rules along a tree for some specific rule sets, leading to inefficient searches. To address the above issues, we propose a centralized and uniform PSTSS together with a tree dilution scheme, following the hardware properties of FPGA. Then, a dynamically updatable hardware architecture for the upgraded algorithm is designed and implemented on a state-of-the-art FPGA. Instead of implementing a fully pipelined design, the patterns of multi-core, multi-engine and coarse-grained pipelines are efficiently explored.

Compared with the well-known decomposition-based design proposed in [39], the TcbTree can support rule sets that are more than an order of magnitude larger; compared with existing decision tree based designs, the overall structure of TcbTree is independent of rule sets and fully supports dynamic update without pre-computing updated memory content. Particularly, the average throughputs of TcbTree for 1k, 10k, 32k and 100k rule sets can achieve 788.8 MPPS, 404.3 MPPS, 237 MPPS and 41.8 MPPS, respectively, in terms of packet classification, and the update throughput for all benchmark rule sets is above 1 MUPS. Overall, the major contributions of this work are as follows.

- A novel two-stage framework consisting of heterogeneous algorithms: decision tree, linear search and TSS, which can build trees in linear memory footprint and avoid rule replications simultaneously.
- A centralized and uniform hash-table based PSTSS approach is proposed, which is tailored to and implemented on the FPGA hardware. A real-time result feedback scheme between trees and PSTSS is also presented to accelerate the tuple space search.

TABLE I
AN EXAMPLE OF 2-FIELD CLASSIFIER

Rule id	Priority	Field X	Field Y	*Action*
R_1	6	111*	*	*action1*
R_2	5	110*	*	*action2*
R_3	4	*	010*	*action3*
R_4	3	*	011*	*action4*
R_5	2	01**	10**	*action5*
R_6	1	*	*	*action6*

- A search tree dilution method is devised for the TSS-combined bit-selecting tree, to address the problem of *big leaf*. The method could largely homogenize the distribution of rules within a tree and reduce the latency of search in leaf nodes.
- An FPGA-based updatable packet classification architecture in supporting large-scale rule sets is designed. Specifically, efficient data structures for the TSS-combined tree are constructed and stored in large pieces of RAMs (instead of distributed small ones). Such a design breaks the constraints of limited tree depth and imbalanced node distribution, making the architecture independent of rule sets.
- With our proposed data structure and the revised PSTSS scheme, fast dynamic rule update is fully supported for rule sets in various scales, including creating, deleting and modifying tree nodes in real time aided by dynamic storage allocation.

The rest of the paper is organized as follows. Section 2 summarizes the background and related work. Section 3 elaborates the TcbTree algorithmic design. Section 4 presents the FPGA architecture. Section 5 provides experimental results. Finally, Section 6 draws the conclusion.

II. BACKGROUD AND RELATED WORK

In this section, the background and classic packet classification approaches, especially the algorithmic approaches of tuple space and decision tree, are first reviewed. After that, related FPGA-based designs are introduced. Finally, the summary of prior art is given.

A. The Packet Classification Problem

The purpose of packet classification is to enable differentiated packet treatment in fine granularity according to multi-field packet header information and a pre-established classifier which consists of a set of rules. Each rule r has d components each represented by r_i. r_i is a regular expression on the i field of the packet header, which could be a prefix, a range or an exact value. A packet $p = (p_1, p_2, \ldots, p_d)$ is said to match rule r if $\forall i, p_i \in r_i$. Table I shows an example of 2-field rule set in [41], and the default priority is the order of the rules. Priority indicates the degree of importance, meaning that if a packet conforms to more than one rule, the low priority rules would give way to the highest priority rule. Next, we will give more details for packet classification from the aspects of algorithmic designs and FPGA designs respectively.

B. Algorithmic Designs for Packet Classification

As an extensively studied problem [3], a lot of algorithmic approaches have been proposed over the past two decades, such as decision tree [42], [43], decomposition [44]–[47] and TSS [48]. Since TSS and decision tree are related to our proposed architecture, more detailed reviews about these two approaches would be given, as well as their latest progress in the last decade.

TABLE II
TSS BUILDS 4 TUPLES FOR RULES IN TABLE I

Tuple	*Rule id*	Rule Priority	Tuple Priority	Field X	Field Y	*Action*
(3, 0)	R_1	6	6	111*	*	$action_1$
	R_2	5		110*	*	$action_2$
(0, 3)	R_3	4	4	*	010*	$action_3$
	R_4	3		*	011*	$action_4$
(2, 2)	R_5	2	2	01**	10**	$action_5$
(0, 0)	R_6	1	1	*	*	$action_6$

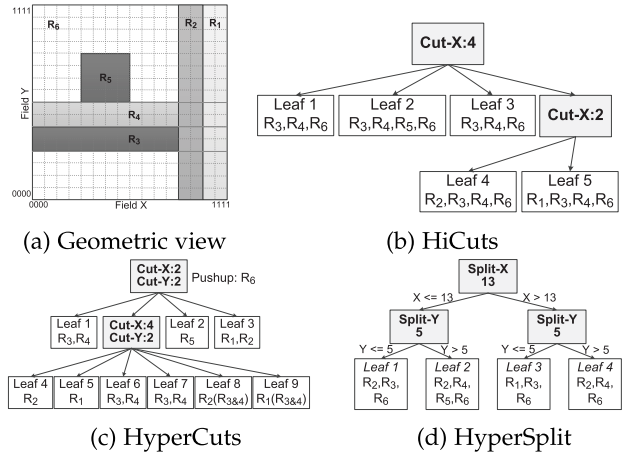

Fig. 1. Review on classic decision trees (*binth* = 4).

1) Tuple Space Search (TSS): A tuple space (tuple for short) is the combination of prefix lengths of different fields [48]. All the rules belonging to a tuple are maintained in an independent hash table. Upon receiving a packet, it simply looks up all the hash tables by probing them with the keys formed by concatenating the known set of bits in each field corresponding to that tuple. Table II shows the four tuples built for rules given in Table I.

In TSS-based solutions, rules can be inserted and deleted from hash tables in amortized one memory access, resulting in a high update performance. But in order to find the best matching rule for each packet, all these partitioned hash tables have to be searched exhaustively, resulting in a low lookup performance. As an improvement, PSTSS [26] reduces average table lookups by introducing a pre-computed priority for each tuple (as shown in the fourth column of Table II), so that each search can terminate as soon as a match is found. However, its worst-case performance is still the same as TSS. TupleMerge [27], a recently proposed TSS-based scheme, improves upon the original TSS by relaxing the restrictions on which rules may be placed in the same tuple. However, with more tuples merged, its performance may be affected due to hash collisions.

2) Decision Tree: Decision tree is a flowchart-like tree structure, where the root node covers the whole searching space containing all rules. From the geometric view, building a tree is to recursively partition the space covered by a node called *parent* into many smaller subspaces (equal-sized or not) until the rules covered by each subspace is less than the pre-defined bucket size (i.e., *binth*). These subspaces are the children of the parent node in the decision tree, and the terminal node is called leaf node. The geometric view of the example rules given in Table I is illustrated in Fig. 1(a), and Fig. 1(b)(c)(d) shows the decision trees generated by classic HiCuts [49], HyperCuts [43] and HyperSplit [19]. In case a rule spans multiple sub-spaces, the undesirable rule replication happens (e.g., R_3, R_4 and R_6 in Fig. 1(b)). When a packet arrives, the decision tree is traversed based on the key values in the packet header, to find a matching rule at a leaf node.

As reviewed in CutSplit [50], rule replication is the key trouble-maker for decision trees. To reduce rule replications, rule set partitioning has been recognized as a common practice and many novel partition based decision trees have been proposed in the past decade. The well-known cutting-based scheme EffiCuts [20] observes that real-world rules exhibit several inherent characteristics, and a good rule set partitioning can significantly reduce rule duplication. So instead of covering all rules with a single decision tree, EffiCuts divides the rules into subsets, each of which independently creates its own decision tree using a variant of HyperCuts. At most 2^d decision trees can be generated for d-field classifiers, resulting in a large number of memory accesses. By contrast, HybridCuts [21] separates rules based on a single field, so it achieves a significant reduction in the number of subsets (i.e., from 2^d to $d + 1$), which in turn reduces overall memory accesses. However, the worst-case search performance of HybridCuts is unbounded due to the adoption of HyperCuts. Worse still, as the number of rule fields and classifier size increases, the performance of HybridCuts drops dramatically due to the rule replication. Other partitioning methods are also emerging, such as ParaSplit [31], SmartSplit [22], PartitionSort [23], CutSplit [50], TabTree [1], NeuroCuts [24], CutTSS [41] and NeuvoMatch [25].

However, the real-time rule update is still facing challenges in most of these decision trees. Besides, all of these decision trees are not very friendly for FPGA implementations, because they are either not balanced enough, or the tree depth is uncontrollable, and these tend to cause FPGA resource waste and bottlenecks in the convergence of parallel results.

C. FPGA Designs for Packet Classification

Although software algorithms have been widely studied, there is a large bottleneck in the performance of software algorithms, and TCAM will not be able to achieve a great increase in capacity in the foreseeable future. Therefore, because of its programmability, high parallelism and large capacity, FPGA has become an ideal hardware platform for accelerating algorithmic packet classification, and has received extensive attention and research. Next, we will generally introduce three mainstream categories of FPGA packet classification technologies: 1) decision tree based, 2) decomposition based, and 3) RAM based TCAM on FPGA.

1) Decision Tree Based Designs on FPGA: The majority of decision tree based architectures have adopted a full pipeline design, which can benefit from the high frequency and high throughput. And the major concerns of this kind of methods are memory reduction and performance enhancement, such as node merging and leaf-pushing to reduce the number of pipeline stages and balance memory allocation [29], subsets partitioning with multiple trees to minimize rule duplication [33], and hybrid scheme combing different algorithms [51]. To further reduce memory consumption, Chang *et al.* [52] proposed a greedy bucket compression scheme to reduce the duplicated rules in the memory bucket pipeline, based on the observation that rule buckets associated with leaf nodes in decision trees consume a large portion of on-chip memory, and Kennedy *et al.* [53] reduce the amount of memory for large rule sets with the proposed pre-cutting

process that results in a shallow decision tree, also reducing the number of memory accesses.

2) Decomposition-Based Designs on FPGA: The principle of decomposition is to decompose a complex multi-domain search problem into multiple simple single-domain concurrent searches, which can make full use of the parallel characteristics of FPGA. Since most of the current decomposition implementations on FPGA are based on the classic BV algorithm [44], BV decomposition is used to represent this type of technology in this paper. In such a method, a two-dimensional pipelined architecture is commonly explored to achieve high performance in classification [39], [54]. In aspect of supporting range match, Chang *et al.* [36] proposed to either use specially designed codes to store the pre-computed results in memory, or perform subrange match operations sequentially. However, it does not support dynamic rule update. For efficient update in SDN switches, Li *et al.* [40] divides the ruleset into subsets and uses BV-based pipelines to match these sub-rulesets separately in parallel. To save stringent memory resources which are wasted to store relatively useless wildcards, Shi *et al.* [55] proposed a memory compression scheme which adopts a memory-shared homogeneous pipeline, together with a rearrange technology by utilizing a bit matrix to determine the potential of memory consumption.

3) TCAM-Based Designs on FPGA: This method utilizes RAMs to emulate the operation of TCAM, which can perform packet classification with few memory accesses and has uniform hardware organization for all rule sets. Instead of brute-force implementations to mimic the native TCAM architecture, Jiang *et al.* [56] presented a modular architecture consisting of arrays of small-size RAM-based TCAM units on FPGA, which scales well in implementing large TCAMs. While Yu *et al.* [57] takes a different approach which first encodes the rule header fields and maps them to SRAM-based match units using a bit-selection approach. Whereas, this may cause rule conflicts in the same bucket. Moreover, only 10k-scale rule sets can be accommodated by this method on Xilinx Virtex 7 and Ultrascale FPGA for above methods.

D. Summary of Prior Art

Although the FPGA designs based on decision trees could provide high throughput and efficient memory usage, the rule duplication problem still remains. The concerns for dynamic rule update in hardware, i.e., how to deal with rule duplication, how to update leaf node or ancestor node backward, and how to create a new node on the fly in a fully pipelined architecture, have not been addressed. Especially for some optimized and balanced memory algorithms, the real-time update becomes more complicated. In contrast, although BV-based designs can sustain a high throughput in packet classification and fully support dynamic rule update, the method is essentially exhaustive to list all possible matching combinations for each bit. Therefore, the consumption of hardware resources is large especially for rules with more wildcards. The scale of rule sets accommodated by FPGAs is always restrained by this feature. Similar to BV-based methods, TCAM simulation method exhaustively lists possible combinations by match vectors, which is also resource-intensive and difficult to scale to large-scale rules.

III. Algorithm Overview

The algorithmic framework of our proposed TcbTree consists of three key components, as shown in Fig. 2. The first key component is rule set partitioning, which divides rules into a few subsets, where rules in each subset share the similar characteristics. The second key component is tree construction for the partitioned rule subsets (except for the last subset), where several balanced trees are generated with linear memory consumption and without the trouble of rule replications. The last key component is a hardware-optimized PSTSS scheme (detailed in Section 3.6), which can efficiently handle the few rules expelled by rule set partitioning. Next, we give more details in terms of these three key components.

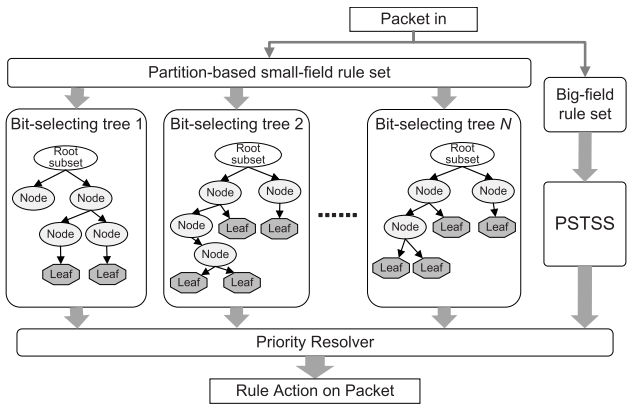

Fig. 2. The algorithmic framework of TcbTree.

A. Rule Set Partitioning

Before describing the partitioning mechanism, we first give the definitions of two concepts: *small field* and *big field*.

1) Small Field: Given an d-field rule $R = (F_1, ..., F_i, ...F_d)$ and a threshold value vector $T = (T_1, ..., T_i, ...T_d)$, we give a definition for field F_i as follows: if the range span length of field $F_i \leq$ threshold value T_i, F_i is defined a *small field*.

2) Big Field: Similarly, if the range span length of field $F_i >$ threshold value T_i, we say that F_i is a *big field*.

Based on the observations revealed in CutSplit [50], even under very demanding thresholds, most rules still have at least one *small field*. Thus, we can partition the vast majority of the rules into a very limited number of subsets without duplicates among each other, where rules in each subset all share the common characteristic of *small field* in the selected fields. The heuristic of rule set partitioning is as follows:

- Distinct fields selecting. Pick up a few distinct *small fields*, where the vast majority rules contain at least one *small field* in selected fields. The remaining rules are divided into big-field rules.
- Fields-wise partitioning. Assume m fields have been selected for d-field rule sets. We categorize rules based on field length (i.e., *big* or *small*) in all selected fields, leading to at most $2^m - 1$ subsets.
- Selective subset merging. The subsets containing a very few rules can be merged into other subsets with fewer *small fields*.

Based on the above rule set partitioning method, rules are firstly separated into a few number of rule subsets based on their *small fields* in order to eliminate rule overlapping at large scales. The subsets of *small field* for typical 5-tuple rule sets are selected to be source address (SA), destination address (DA), SA and DA combined (SA_DA) in this work. Since only few big-field rules are left after partitioning, the tuple space structure is then utilized to process these rules.

TABLE III
EXAMPLE RULE SET WITH TWO IPV4 ADDRESS FIELDS

rule id	src_addr field	dst_addr field	*rule id*	src_addr field	dst_addr field
R_1	228.128.0.0/9	0.0.0.0/0	R_{11}	0.0.0.0/1	226.0.0.0/7
R_2	223.0.0.0/9	0.0.0.0/0	R_{12}	128.0.0.0/1	120.0.0.0/7
R_3	0.0.0.0/1	175.0.0.0/8	R_{13}	128.0.0.0/2	120.0.0.0/7
R_4	0.0.0.0/1	225.0.0.0/8	R_{14}	128.0.0.0/1	38.0.0.0/7
R_5	0.0.0.0/2	225.0.0.0/8	R_{15}	0.0.0.0/1	120.0.0.0/7
R_6	128.0.0.0/1	123.0.0.0/8	R_{16}	128.0.0.0/2	160.0.0.0/5
R_7	128.0.0.0/1	37.0.0.0/8	R_{17}	16.0.0.0/4	36.0.0.0/6
R_8	0.0.0.0/0	123.0.0.0/8	R_{18}	200.0.0.0/5	168.0.0.0/5
R_9	178.0.0.0/7	0.0.0.0/1	R_{19}	196.0.0.0/2	96.0.0.0/5
R_{10}	0.0.0.0/1	172.0.0.0/7			

By examining rules grouped with respect to their *small fields*, we identify a useful observation on rule fields, which is the key basis of the following proposed bit-selecting trees: For the *small field* of grouped rules with the type of prefix or exact value, there are a set of most significant bits which are indicated by either bit-0 or bit-1. More specifically, for a *W*-bit wide field F_i with the threshold value of 2^K, we can draw the conclusion that, F_i is a *small field* if and only if there is no wildcard (*) at its most significant $W-K$ bits. In this paper, we call these $W-K$ bits as ***selectable bits***. For the *small field* of grouped rules with range type, we give a novel encoding scheme called *False Range Encoding*, and show that the observation is still valid for its encoded prefixes, which is detailed in the Appendix.

B. Bit-Selecting Tree Construction

By grouping rules that are narrow in the same fields, we get a set of *selectable bits* among grouped rules without wildcard value. Thus, each *selectable bit* can map (i.e., partition) rules into at most two subsets without any rule replications. To exploit this favorable property, we build a multi-way tree by selecting a few *selectable bits* in each tree node recursively.

In order to build shallow and balanced decision trees, the heuristic greedy strategy bit-selecting algorithm is utilized to select most distinguishing *selectable bits* in the process of tree construction. To control the width of the tree, we assume that at most *b* bits are allowed to be selected in each tree node.

Greedy Strategy: The greedy algorithm tries to find a local optimal solution, where the "good" bits are selected one by one recursively. We assign an *imbalance* value for each current selectable and unused bit by using the formula (1), where *#ruleLChild*/*#ruleRChild* is the number of rules mapped into the left/right child node (i.e., #bit-0/1s in *v*-th bit). The greedy algorithm is to choose at most *b* bits one by one, where each selected single bit is with the smallest *imbalance* value among current selectable and unused bits.

$$imbalance(bit\ v) = |\#ruleLChild - \#ruleRChild| \quad (1)$$

In Fig. 2, each decision tree is corresponding to a specific field or a combination of fields, the rules contained in which conform to the small-field characteristics of the relative field. The decision tree structure is composed of nodes which are divided into internal nodes and leaves.

In order to bound tree depth, avoid rule replication and support fast rule updates, the approach stops its bit-selecting progress in one of the following cases: 1) the tree depth achieves the predefined maximum value; 2) the number of rules in the mapped tree node is less than a predefined threshold value (i.e., *binth*); 3) the remaining unselected rule bits share same values and cannot separate rules from each other; 4) the further bit-selecting will lead to rule replications due to the wildcards. Due to the above termination mechanism, some terminal nodes might not be further split in which #rules $>$ *binth*. These nodes are called *big leaf* nodes in this work.

TABLE IV
PARTITIONED RULES WITH SMALL DST_ADDR FIELD

rule	src_addr (T_{src_addr}= 2^{25})	dst_addr (T_{dst_addr}= 2^{25})	
id	1-32th bits	**33-39th**	40-64th bits
R_3	0*********************************	**1010111**	1*********************
R_4	0*********************************	**1110000**	1*********************
R_5	00********************************	**1110000**	1*********************
R_6	1*********************************	**0111101**	1*********************
R_7	1*********************************	**0010010**	1*********************
R_8	*********************************	**0111101**	1*********************
R_{10}	0*********************************	**1010110**	*********************
R_{11}	0*********************************	**1110001**	*********************
R_{12}	1*********************************	**0111100**	*********************
R_{13}	10********************************	**0111100**	*********************
R_{14}	1*********************************	**0010011**	*********************
R_{15}	0*********************************	**0111100**	*********************

TABLE V
PARTITIONED RULES WITH *Big Field*

rule	src_addr (T_{src_addr}= 2^{25})	dst_addr (T_{dst_addr}= 2^{25})	
id	1-32th bits	33-39th	40-64th bits
R_{16}	10********************************	10100**	*********************
R_{17}	0001*********************************	001001*	*********************
R_{18}	11001*********************************	10101**	*********************
R_{19}	11**********************************	01100**	*********************

C. A Working Example

To illustrate the algorithm more clearly, we give a working example for rules presented in Table III. Assume that each internal tree node is allowed to select a maximum of two bits for rule mapping and the *binth* of leaf is one, the threshold value vector is T = (T_{src_addr}= 2^{25}, T_{dst_addr}= 2^{25}). Only source address (SA) and destination address (DA) are merely selected for subset partitioning in this example.

The scheme first partitions the 19 rules into three rule subsets: $(arbitrary_{SA}, small_{DA})$ = {R_3, R_4, R_5, R_6, R_7, R_8, R_{10}, R_{11}, R_{12}, R_{13}, R_{14}, R_{15}}, $(small_{SA}, arbitrary_{DA})$ = {R_1, R_2, R_9} and $(big\_field\ rules)$ = {R_{16}, R_{17}, R_{18}, R_{19}}. Thus, for the *small field* of grouped rules in the former two subsets, there are 32−25=7 *selectable bits*. After rule set partitioning, a decision tree is constructed for each small-field rule subset. For example, Table IV shows the partitioned rules in $(arbitrary_{SA}, small_{DA})$ with the representation of ternary strings, while Table V shows the big-field rules. Clearly, the middle 7 bits (i.e., 33-39th) in Table IV are *selectable bits*. Fig. 3 illustrates the DA decision tree constructed together with the PSTSS for the rules shown in Table IV and Table V, based on the proposed bit-selecting and TSS-combined method. Although this example only shows the decision tree of DA, the construction mechanism of SA tree is the same as that of DA tree, which will not be detailed here.

D. Challenges for Hardware Design

In TabTree [1] and CutTSS [41], both the linear search (#rules $\leq$ *binth*) and the PSTSS (#rules $>$ *binth*) for rules in the terminal nodes (i.e., leaf nodes) are employed to facilitate tree constructions. The big-field rules are processed by PSTSS as well. Each TSS structure consists of multiple tuples each associated with an independent hash table and search

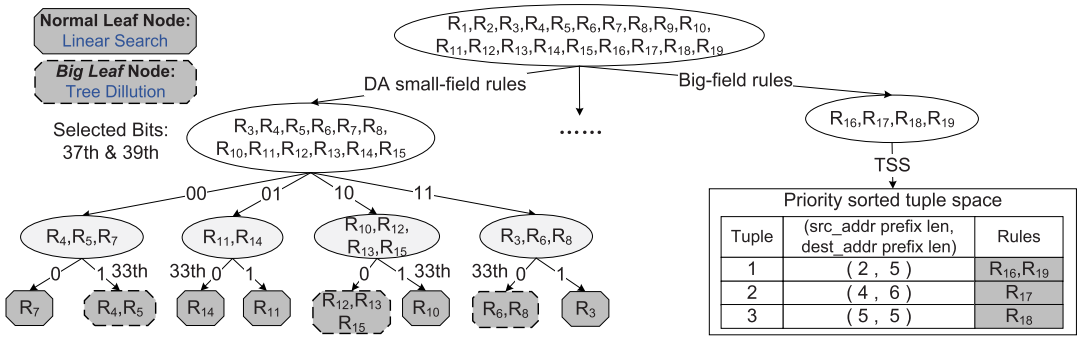

Fig. 3. TSS-combined decision tree for rules in Table IV and Table V (*binth* = 1).

mechanism. If this structure is implemented in hardware, there would be dozens of PSTSS instantiated together with hundreds of thousands of tuples and hash tables, and thus hardware resource consumption is unimaginable. Moreover, the number of TSS and tuples is unpredictable which is dependent on the specific rule set and configurations of tree construction. To address this issue, TcbTree adopts linear search for all leaf nodes while the TSS structure merely targeting big-field rules is remained, which is called TSS-combined bit-selecting tree algorithm. In spite of the above modifications, designing scalable and efficient architecture for this algorithm still faces two major difficulties and challenges:

1) Hardware-Unfriendly TSS Structure: Even for the only one left TSS to process big-field rule subset, the multiple-tuple multiple-hash-table structure is still hardware unfriendly due to the aforementioned reasons.

2) Inefficient Search Caused by Big Leaf: Enforcing linear search for all leaf nodes would aggravate and magnify the problem of *big leaf*, especially for the inseparable rules containing the same values in selected small fields. Searching a large number of rules sequentially in leaf nodes is time-consuming and inefficient.

Regarding the issues above, this work gives solutions by proposing **a hardware-optimized PSTSS** and **a tree dilution scheme** to achieve a complete and hardware-friendly framework. The details are elaborated in the following two subsections.

E. Hardware-Optimized PSTSS

The framework of hardware-optimized PSTSS is illustrated in lower part of Fig. 5. Two major modifications have been made to original PSTSS approach according to the hardware characteristics:

1) Incorporation of Port Fields: In prior work, each tuple is composed of source IP mask and destination IP mask. In many cases, however, big-field rules contain the same masked value in these two fields making them indistinguishable, which consequently leads to serious collisions in the hash table. The linear bucket length would be consequently long. To address this issue, source port LCP and destination port LCP are precomputed and added to tuple space to make the tuple combination more diversified and thereby reduce the conflict possibility.

2) Centralized Hash Table: Each tuple in the software algorithm is assigned a hash table, which is not friendly to hardware that needs to fix resources in advance, and causes a waste of resource. We use a centralized hash table to overcome this problem, which combines the hash tables of all tuples into one and is shared by all tuples. When traversing each tuple, each packet header field value is first ANDed with the hash key converted by the corresponding fields and prefix lengths and then hashed. With the hashed value as index address, one entry of the centralized hash table is accessed to perform a linear lookup. Conflicting values within each entry are stored in a bucket in the format of a linked list.

The hash table is a two-level query structure which consists of a bucket table and a big rule (short for big-field rule) table. The bucket table serves as the mapping between the hashed value and its corresponding rule subset address in big rule table. Its each address corresponds to a hashed value, and each content is the address of the first rule in the rule subset. The big rule table records all consecutive big-field rule subsets.

Here is a working example for classifying a 2-field incoming packet P=<19.0.0.0, 38.0.0.0> with the constructed TSS shown in Fig. 3. P is first searched in the tuple $(2, 5)$ by $hash$ (19.0.0.0 & C0.0.0.0, 38.0.0.0 & F8.0.0.0) (prefix length of 2 and 5 corresponds to $C0.0.0.0$ and $F8.0.0.0$) and after checking bucket table with the hashed value as the address, the rule table entry containing R_{16} and R_{19} (maybe more rules in the entry since centralized table would have more collisions than separated ones) is entered without no match. Similarly, in the tuple $(4, 6)$, the rule table entry containing R_{17} is entered with $hash$(19.0.0.0 & F0.0.0.0, 38.0.0.0 & FC.0.0.0), in which R_{17} is matched. There is no match in the last tuple $(5, 5)$. Thus R_{17} is the final best matching rule for P.

It is worth noting that the rules in both leaf nodes and TSS buckets are sorted by priority in advance. The advantage is that when a rule is matched, there is no need to continue searching in the subset. Tuples are also sorted according to the priority of rules which rank highest in the respective tuples. In order to maintain the priority structure, the rule fast update scheme in our architecture is in accordance with the priority principle as well.

F. Tree Dilution Mechanism

For some specific rule sets, the values of many rules in the same *small field* are exactly the same. As a result, it is impossible to divide the rules further with a very deep bit-selection during tree construction, resulting in so called *big leaf* problem.

This issue is addressed by the proposed tree dilution mechanism which can be illustrated by Fig. 4. Two *binth* values are set, which are *tree binth* and *big leaf binth* respectively. The main tree is firstly constructed with *tree binth*, while the *big leaf binth* is a threshold of rule number associated with a *big leaf* node, which thereby determines the number of dilution trees. Part of rules that exceed *big leaf binth* in all

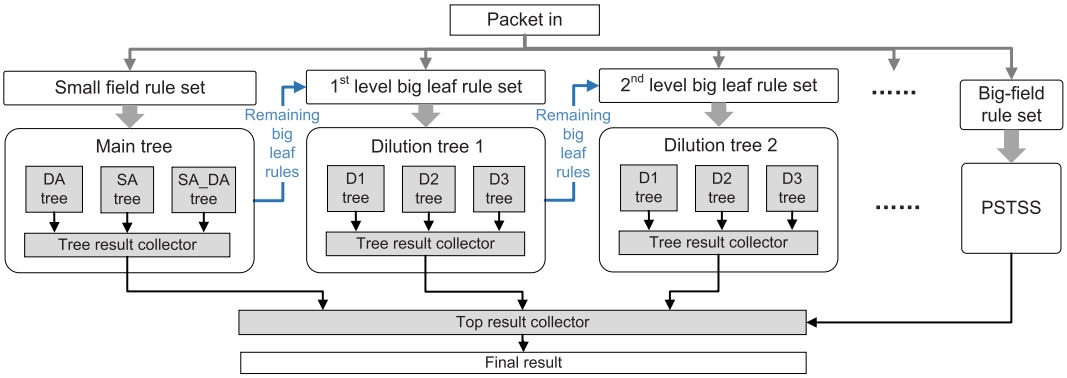

Fig. 4. Dilute tree architecture.

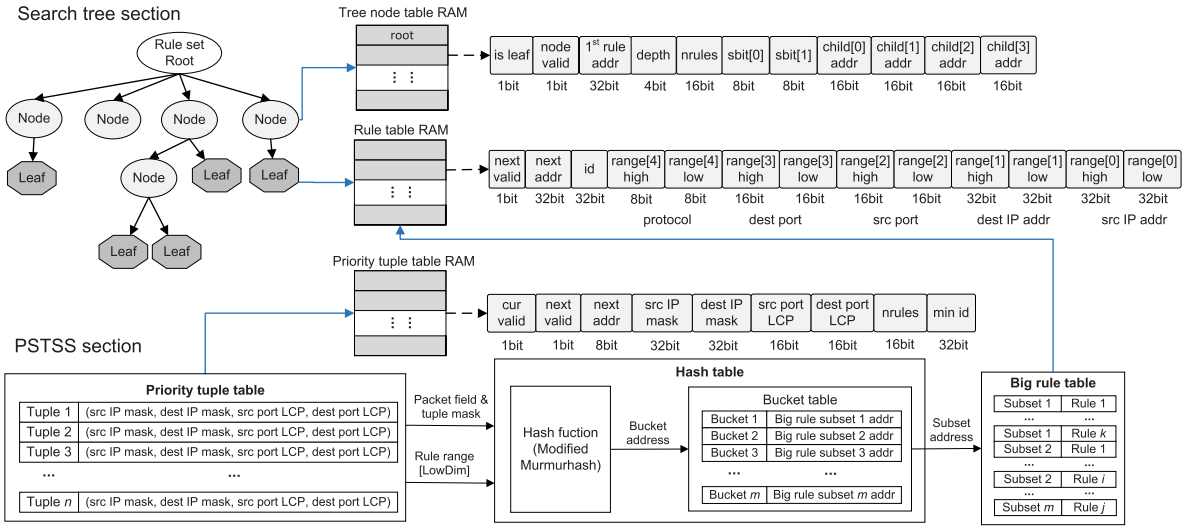

Fig. 5. The data structure design for search tree and PSTSS.

leaf nodes are screened out to build the first-level dilution tree in the manner of bit-selecting. A dilution tree consists of three sub-trees which can correspond to any fields which need to dilute. Next, the second-level dilution tree would be constructed recursively if the rule number of a leaf in dilution subtrees exceeds *big leaf binth*. This process could continue until the number of rules in all leaf nodes is less than or equal to *big leaf binth*. The results from the main tree and dilution trees would be processed hierarchically level by level. Smaller *big leaf binth* would bring more dilution trees and better performance for each tree. The side effect is that the system complexity becomes higher, the result resovler levels becomes more, and the working frequency decreases as a result. Hence choosing a balanced *big leaf binth* is more favoured.

IV. Hardware Architecture Design

A. Existing Problems

Although extensive research has been conducted on decision-tree algorithm implementations, fully pipelined architectures on FPGA are still facing many challenges due to the following facts. First, most actively investigated algorithmic decision trees are imbalanced which causes the allocation of memory along the pipeline difficult especially when the pipeline is long. As a result, decision tree architectures would highly depend on rule sets. Second, only a limited number of tree levels is supported due to the explosive growth of high-level nodes. Furthermore, the overly distributed storage structure hinders real-time rule update, because the pipeline is one-way flow, dynamic update requires reverse flow to renew node information.

To address the above issues, an efficient set of data structures and hardware architecture on FPGA for TcbTree are proposed, which are described in details in the following subsections.

B. Data Structure

As shown in Fig. 5, three chain-table-like data structures are constructed for search tree nodes, priority tuples, and rules, respectively.

1) Node Table: Every node in a search tree, including internal node and leaf node, is associated with a table of 134 bits (*selectable bit* number is 2) which is called node table. The first bit *is_leaf* indicates current node is internal node or leaf node. The second bit *node_valid* indicates if current node is valid, which could be modified during update. This bit being 0 means there is no rule associated with this node. Following bits *depth* and *nrules* represent node level depth and number of related rules for current node. The *sbit* is the position of selection bit in the field prefix, while the *child_addr* is the RAM address for next-level node.

If the current node is an internal node, two selection bit positions $sbit[0]$ and $sbit[1]$ are utilized to select the corresponding bits in *selectable bits* of current prefix and join them, so the next-level child node address would be determined by the specific value of spliced 2 bits. If current node is a leaf

node, then only the field of $1^{st}\_rule\_addr$ is referred to locate the rule table RAM address of the first rule within the leaf subset.

2) Priority Tuple Table & Bucket Table: Similarly, every tuple is corresponding to a priority tuple table. The first two bits *cur_valid* and *next_valid* indicate if current and next tuple are valid. In the tuple search process, these bits would be firstly checked to make sure starting from valid tuples. Rule number is recorded in *nrule* dynamically for each tuple. The minimum rule ID (equivalent to highest priority) is also recorded in tuple table. The tables are sorted in advance according to the highest priority of respective rule subset in each tuple, and then connected sequentially through the *next_addr* field. Following words are the *IP masks* and *port LCPs* for 4 different fields. The bucket table serves as the mapping between the hashed value and its corresponding rule subset address. Its each address corresponds to the hashed value, while each content is the rule table address of the first rule in rule subset.m

3) Rule Table: Each rule in tree leaf nodes or TSS has a 273-bit rule table structure for the 5-tuple format. The mask is transferred to ranges with two endpoints in advance and recorded in this table. The rule tables in the rule subset associated with each leaf node or tuple are stored consecutively, and the bit *next_valid* indicates whether the next rule is valid. A *next_valid* value of 0 indicates that this rule is the last one in the current subset. The rule tables of all rule subsets in each tree or TSS are also stored continuously, and the starting address of each subset is indicated by the $1^{st}\_rule\_addr$/*Big_rule_subset_#_addr* field in the corresponding node table/bucket table.

4) Storage Mechanism: These multiple types of tables are stored in bulks of on-chip memories: node table RAM, tuple table RAM, bucket table RAM, tree rule table RAM and big rule table RAM. More specifically, all node tables throughout one tree are stored in one RAM, and the same is true for tuple tables, bucket tables and rule tables. In other words, the RAM management is centralized rather than distributed. Therefore, the tree traversal to search a rule is converted to the addressing process. Such design has three major advantages:

- The overall architecture does not depend on specific rule sets, since there is no need to allocate specific memory size for each level of nodes.
- The level of tree nodes could not be restricted, as the searching down to next-level is essentially a recursive addressing controlled by a finite state machine (FSM). The tree reconstruction is equivalent to the overwriting of RAM data which is facilitated by the centralized memory scheme.
- It motivates the support for large-scale rules, as the scalability for different sizes of rule sets is achieved merely through configuring the depth of different RAMs in the architecture.
- It facilitates real-time rule update. Information on adjacent levels is interrelated, thus rules could be traced back to upper-level nodes with cached node information.

C. Top-Level Architecture

The top-level block diagram of proposed FPGA architecture is shown in Fig. 6. Instead of a purely pipelined design where performance scales linearly with operating frequency, the scheme of multiple cores computing simultaneously is adopted for outermost architecture to explore data-level and task-level parallelism. This scheme has high memory capacity requirements, therefore the high-density on-chip memory UltraRAM in the most advanced FPGA is leveraged in this architecture.

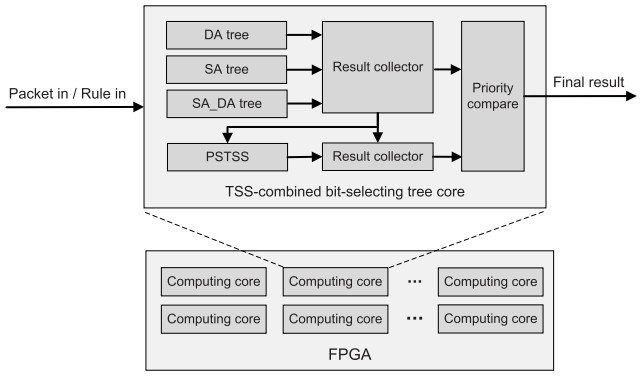

Fig. 6. Top-level architecture design of the system.

Taking typical 5-tuple rule set as an example, each computing core consists of PSTSS and three tree structures based on the *small field* rule subsets according to the algorithm: source IP address (SA), destination IP address (DA), SA and DA combined (SA_DA). Every packet/rule would go through all these four modules to make a match/update. The results of packet classification from search trees and PSTSS would be collected and compared by priority to select the final rule ID.

D. Search Tree Architecture

The architecture for each tree structure is composed of Node Search module and Rule Processor module, which is illustrated in Fig. 7. The former one is in charge of traversing the tree node by node from the root, finding the leaf possibly containing matched rules, and locating the start address of the subset associated with this leaf. The address and the node information of last two levels are transferred to the later module. The Rule Processor module searches rules linearly through looking up the rule table RAM and makes actions of search, delete, or insert according to the operation code (OP code).

The rules to be added or deleted are processed in the same way as the packets in Node Search stage with the lower endpoint of range as input, but they are processed by rule delete and insert engines separately in Rule Processor. With cached complete information from the Node Search module, upper-level nodes can be traced back to support real-time update of internal nodes, as well as deletion and addition of leaf nodes.

In order to manage the available space in node table RAM and rule table RAM for dynamic rule update in real time, two empty address allocators are designed interacting with rule delete and insert engines to recycle and reallocate empty RAM addresses. The associated FIFOs record the addresses of the emptied content after deletion and provide available addresses for insertion.

These two modules operate independently and constitute a two-stage coarse-grained pipeline, although they are not pipelined internally. The memories in the tree and PSTSS architecture are dual-port to facilitate independent lookup and update, at the cost of a little extra logic consumption that hardly affects the overall design. Furthermore, to achieve parallel lookup, the rule table RAM is implemented as a true

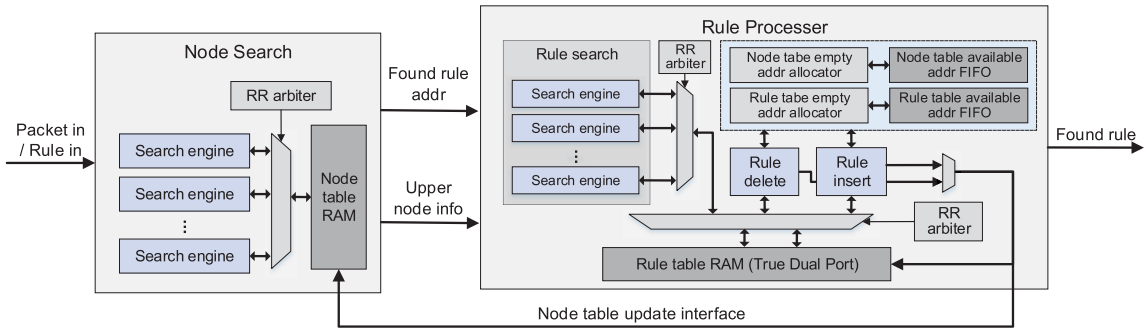

Fig. 7. The hardware architecture of search tree.

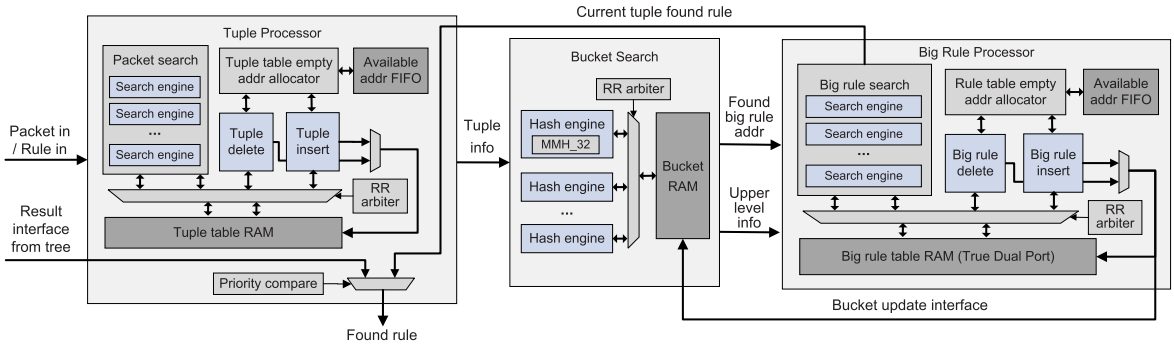

Fig. 8. The hardware architecture of PSTSS.

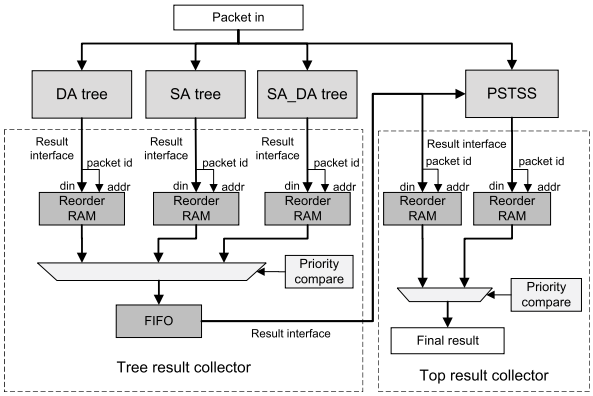

Fig. 9. Result collector module.

dual-port RAM. In Node Search and Rule Search in Rule Processor, multiple engines are implemented to speed up the search process. What is more, because the memory access has a latency and each engine only takes a limited time to query the storage, increasing the number of engines could significantly improve memory utilization. However, excessive engines will encounter the bottleneck of memory access, in other words, when the number reaches a certain value, the search performance will not be further improved by continuing to increase engines. Thus we configure the number that can maximize memory access efficiency in the actual implementation. The arbitration between multiple engines is ensured by Round Robin, as each engine has the same priority.

E. PSTSS Architecture

The PSTSS module has three submodules in accordance with the phase of tuple search, hash and bucketd table query, big rule processing, respectively. The detailed block diagram is shown in Fig. 8. Similar to the hardware design idea of search tree, PSTSS architecture adopts a coarse-grained pipeline with three separate modules. And dual-port RAM, multiple search engines in Tuple Processor and Big Rule Processor as well as multiple hash engines in Bucket Search module are implemented. The Tuple Processor leads the entire process by: reading each tuple in sequential order, sending packet/rule to the hash engine, receiving search/update feedback from Big Rule Processor, and determining whether to continue to search from the next tuple. The hash function in PSTSS architecture is a modified MurmurHash.

In packet classification, if the priority of matched rule is higher than the maximum priority of next tuple, the search would be terminated. However, the TSS-based method still needs to traverse multiple tuple tables averagely, resulting in low efficiency. To take advantage of hardware parallel property, we propose a real-time interaction scheme between decision trees and PSTSS. When each set of search trees has found a match, the rule would be promptly transferred to Tuple Processor and compared with the current matched rule priority in tuple (if any) and maximum priority of next tuple. The feedback from trees would be the final result if it has a higher priority and the tuple space search would also be terminated, otherwise the search in tuple space would continue. Since the vast majority of rules can be partitioned into small-field trees, this mechanism can dynamically save redundant search in the vast majority of tuple space.

F. Hierarchical Result Resolver

The proposed TcbTree faces a difficulty, that is, there are multiple tree structures for different *small field* combinations, and the speed of different trees producing results is not consistent, thus the results need to be collected separated, aligned and compared. Moreover, there are multiple search engines in a tree structure, so multiple packets are processed simultaneously at different speeds, resulting in non-sequential

Algorithm 1 Insertion of Rule for Search Tree

Input: $1^{st}\_rule\_addr$, rule to insert: ins_rule
if $no\_rule\_empty\_addr$ **then**
  return INSERT_FAIL
else if $(no\_leaf\_node \mid\mid invalid\_rule\_node)$ **then**
  if $(no\_rule\_empty\_addr \mid\mid no\_node\_empty\_addr)$ **then**
    return INSERT_FAIL
  else
    obtain new $rule\_table\_empty\_addr$
    write ins_rule to rule table RAM
    if no_leaf_node **then**
      obtain new $node\_table\_empty\_addr$
      write leaf node to node table RAM
      update parent tree node table
else
  $read\_addr = 1^{st}\_rule\_addr$
  while 1 **do**
    read rule table from RAM with $read\_addr$
    if $(ins\_rule\ priority\ >$ *read rule priority*$)$ **then**
      obtain new $rule\_table\_empty\_addr$
      write ins_rule to rule table RAM
      if *is the first rule* **then**
        update $1^{st}\_rule\_addr$ in parent node
      else
        update $next\_addr$ in previous rule table
      break
    else if $(next\_addr == Null)$ **then**
      obtain new $rule\_table\_empty\_addr$
      write ins_rule to rule table RAM
      update $next\_addr$ in ins_rule to $Null$
      return INSERT_SUCCESS
    else
      $read\_addr = next\_addr$ of rule table

Algorithm 2 Deletion of Rule for Search Tree

Input: $1^{st}\_rule\_addr$, rule to delete: del_rule
if $(no\_leaf\_node \mid\mid invalid\_rule\_node)$ **then**
  return DELETE_FAIL
else
  $read\_addr = 1^{st}\_rule\_addr$
  while 1 **do**
    read rule table from RAM with $read\_addr$
    if $rule_match$ **then**
      put $read\_addr$ to empty addr list
      if $(read\_addr = 1^{st}\_rule\_addr)$ **then**
        // first rule in chain
        if $(next\_addr == Null)$ **then**
          update $1^{st}\_rule\_addr = Null$ in parent node
        else
          update $1^{st}\_rule\_addr = next\_addr$ in parent node
      else if $(next\_addr == Null)$ **then**
        // last rule in chain
        update $next\_addr = Null$ in previous rule table
      else
        update $next\_addr$ in previous rule table
      return DELETE_SUCCESS
    else if $(next\_addr == Null)$ **then**
      return DELETE_FAIL
    else
      $read\_addr = next\_addr$ of rule table

results. This issue is also encountered in other multiple-tree based FPGA implementations, and the common solution is utilizing pipeline bubbles which is inefficient and not suitable for our scenario.

A hierarchical result resolver is proposed in this work which is shown in Fig. 9. Each tree has an independent reorder RAM to sequentialize out-of-order results. The write address is the remainder of packet ID divided by the depth of reorder RAM. Right after a certain number of results are written, the results of different trees are read in order at the same time and the priorities are compared. The rule with the highest priority is selected and output to a FIFO controlled by the bus interface to PSTSS. The results read from the FIFO are sent to PSTSS module and another reorder RAM in two separate channels. After the last priority comparison, the final result is output.

The processes of result collection and readout in every level are carried out simultaneously and do not interfere with each other, which is enabled by dual-port RAMs. Furthermore, the threshold of the amount of results for reorder RAMs in each tree result collector is set to minimum for the purpose of outputting results to PSTSS as soon as possible to finish the tuple space search.

G. Dynamic Update Scheme for Hardware

The hardware dynamic update schemes for decision tree and PSTSS are shown in Algorithm 1, 2 and 3 respectively. The update types include deletion and insertion of one rule once a time. The modification is achieved by a combination of consecutive operations of deletion and addition. Facilitated by the Node/Rule update interface in the architecture and the support of reverse data flow, the update scheme of search tree effectively supports real-time leaf node creation or invalidation, as well as parent node table renewal. Similarly, the PSTSS also supports creation or invalidation of tuples and buckets. The priority order of rule subset within a leaf node or a tuple is also maintained after update to facilitate subsequent lookups.

In order to prevent rule duplication, a rule to be inserted or deleted will first determine the corresponding tree or PSTSS to be operated according to the prefix length of its *small field*. In the case of tree dilution, the insertion operation is only performed on the main tree or PSTSS, and the deletion is performed on all trees or PSTSS.

Although the scheme can guarantee the priority order of big rule tables after update in PSTSS, the order of tuples could not be maintained after the deletion or insertion of rules with the highest priority in any tuples. If the tuple tables change a lot, it is more preferred to reconstruct the complete tuple table set. Nevertheless, the rules that are ranked behind in tuple tables usually have a relatively lower overall priority, and the actual impact of real-time updates to the actual order is thus trivial. A mechanism to guarantee no missing matches is designed:

1) Deletion: Even the highest priority rule in a tuple is deleted, the highest priority for the tuple is kept unchanged. This would prevent the match found by the previous tuple from missing a comparison with the priority of next tuple of the currently updated tuple, thus can make sure the rule being searched with enough number of tuples at the cost of one extra tuple search at most.

2) Insertion: If the priority of inserted rule ranked highest in a tuple, the max priority of the tuple would be renewed, and

Algorithm 3 Rule Update for PSTSS

```
Input: operation code: OP_code, rule to update: up_rule, rescode from
       Big Rule Processor
if (OP_code == DELETE) then
    read_addr = 1st_tuple_addr
    while 1 do
        read tuple table from RAM with read_addr
        if tuple_match then
            // rule belongs to current tuple
            HASH(up_rule) and search in big rule table
            if (rescode == DELETE_SUCESS) then
                update rule_num in current tuple table
                if (rule_num == 0) then
                    cur_addr_val = 0 in tuple table
                return DELETE_SUCCESS else
                    return DELETE_FAIL
        else if (next_addr == Null) then
            return DELETE_FAIL
        else
            read_addr = next_addr of rule table
else if (OP_code == INSERT) then
    read_addr = 1st_tuple_addr
    while 1 do
        read tuple table from RAM with read_addr
        if tuple_match then
            // rule belongs to current tuple
            HASH(up_rule) and insert in big rule table
            if (rescode == INSERT_SUCESS) then
                update rule_num and max_priority in current tuple table
                return INSERT_SUCCESS
            else
                return INSERT_FAIL
        else if
        (next_addr == Null || up_rule priority > max_priority)
        then
            if no_tuple_empty_addr then
                return INSERT_FAIL
            else
                // create a new tuple
                if (read_addr = 1st_rule_addr) then
                    // new tuple on top is not allowed
                    return INSERT_FAIL
                else
                    hash up_rule
                    inset in big rule table and update bucket table
                    if (rescode == INSERT_SUCESS) then
                        obtain new tuple_table_empty_addr
                        write new tuple to tuple table RAM
                        update next_addr in current tuple table
                        return INSERT_SUCCESS
                    else
                        return INSERT_FAIL
        else
            read_addr = next_addr of rule table
```

this priority and tuple are recorded. When searching in preceding tuples, the recorded priorities will be complementarily compared with the priority of already matched rule to prevent omission. If such kind of update happens multiple times, it is better to reconstruct the tuple tables.

V. Experimental Result

A. Experiment Setup

Three types of rule sets are generated by ClassBench [58] with the first seed file to make the performance evaluation: Access Control List (ACL), Firewall (FW) and Internet Protocol Chain (IPC), each of which has four sizes from small scale to large scale: 1k, 10k, 32k, 100k. Corresponding synthetic packet traces are also generated along with the rule sets by ClassBench. The key RTL codes corresponding to Algorithm 1 and 2 can be downloaded from the website (http://www.wenjunli.com/TabTree).

The evaluation platform is Xilinx Virtex UltraScale+ VU9P FPGA, which is equipped with a large amount of UltraRAMs. Through taking advantage of this property, multiple computing cores can be instantiated to explore a high performance, since each core contains a complete set of search trees and PSTSS structure. The design and implementation tool is Vivado 2021.2, and the strategies for synthesis and implementation are Defaults and Performance_ExtraTimingOpt, respectively. The RTL language used is System Verilog HDL.

The *small field* threshold, i.e., the selectable prefix length for IP addresses, and the selection bit number in decision trees are configured to 14 and 2 respectively. The number of search engines in Node Search and Rule Processor in search tree architecture is set to 6 uniformly.

TcbTree is independent of rule sets and the configuration can be unitized for the rule sets in a same size. Nevertheless, in order to explore the characteristics of various rule sets and achieve the optimal performance for a specific rule set, hardware configurations without dilution for different sizes and types of rule sets have been customized and finely tuned, which are listed in Table VI. The parameters can be customized include *binth*, RAM depth and RAM type for tree node table, tree rule table and PSTSS big rule table. The *binth* is determined through extensive experiments. The RAM depth is determined by the specific numbers of nodes and rules for each tree. Based on the principle of resource balance, Block and Ultra RAMs are reasonably allocated for different node and rule tables.

It is discovered through comprehensive experiments that the trees constructed for the rule set of fw_100k have a serious *big leaf* problem, so tree dilution approach is applied herein to evaluate its effectiveness. Table VII lists the detailed configurations for three dilution modes with different dilution subtree numbers of 2, 3, and 6 respectively.

B. Resource Utilization

Table VIII summarizes the configurations, resource usage and maximum frequency of hardware designs for various rule sets after synthesized, placed and routed. The average, minimum and maximum latency for classifying a packet is also provided. The number of search engines in search trees maintains the same, while the optimal numbers of search engines and hash engines in PSTSS modules vary which are determined through trial and error with extensive experiments. Besides, according to resource constraints, different maximum computing core numbers are set to pursue maximum possible throughputs. It can be noted that the memory including UltraRAM and BlockRAM is the most consumed FPGA resource. Thus an FPGA equipped with UltraRAM is more suitable for our architecture.

TABLE VI

HARDWARE CONFIGURATIONS FOR DIFFERENT RULE SETS

Ruleset	Binth	Number of nodes/rules							RAM depth (bit)							RAM type						
		SA tree		DA tree		SA_DA tree		PSTSS	SA tree		DA tree		SA_DA tree		PSTSS	SA tree		DA tree		SA_DA tree		PSTSS
		node	rule	node	rule	node	rule	rule	node	rule	node	rule	node	rule	rule	node	rule	node	rule	node	rule	rule
ipc_1k	10	36	95	45	126	320	782	4	6	7	6	8	10	12	12	Block	Block	Block	Ultra	Ultra	Ultra	Ultra
acl_1k	10	9	16	1	1	316	916	16	4	5	2	2	10	12	12	Block	Ultra	Block	Block	Ultra	Ultra	Ultra
fw_1k	10	203	684	24	134	74	209	10	8	12	5	8	7	8	12	Ultra	Ultra	Block	Block	Block	Ultra	Ultra
ipc_10k	20	109	882	142	1276	1243	7327	77	8	10	8	11	11	13	6	Block	Block	Block	Ultra	Ultra	Ultra	Block
acl_10k	20	21	113	0	0	1702	9022	28	5	8	0	0	11	14	9	Block	Block	Block	Block	Block	Ultra	Block
fw_10k	15	341	2304	1365	5828	225	998	354	12	12	12	13	10	12	9	Block	Ultra	Ultra	Ultra	Block	Ultra	Block
ipc_32k	20	341	2557	557	3966	4423	25116	64	9	12	10	12	13	15	12	Block	Block	Block	Block	Ultra	Ultra	Ultra
acl_32k	16	85	542	1	5	5194	31196	335	7	10	3	4	13	15	12	Block	Block	Block	Block	Block	Ultra	Ultra
fw_32k	20	1281	7251	1845	18720	435	3122	751	11	13	11	15	10	12	12	Block	Ultra	Block	Ultra	Block	Block	Ultra
ipc_100k	8	2631	7912	4673	12888	24587	78359	155	12	13	13	14	15	17	12	Block	Block	Block	Block	Ultra	Ultra	Ultra
acl_100k	20	332	1813	1	15	17027	97377	335	9	11	3	5	15	17	12	Block	Block	Block	Block	Block	Ultra	Ultra
fw_100k	20	1357	21339	5461	58251	1564	9853	1540	11	15	13	16	11	10	12	Block	Ultra	Ultra	Ultra	Block	Block	Ultra

TABLE VII

DILUTION CONFIGURATIONS FOR FW_100K RULESET

Dilution mode	*Tree binth*	*Big leaf binth*	Tree type	Subtree field	Number		RAM depth (bit)		RAM type	
					node	rule	node	rule	node	rule
1 dilution tree 2 subtrees	20	183	main tree	SA	1357	18453	11	15	Block	Ultra
				DA	5461	58251	13	16	Ultra	Ultra
				SA_DA	1564	9853	11	10	Block	Block
			1st dilution tree	SA	78	2315	8	12	Block	Block
				SA	35	631	7	10	Block	Block
1 dilution tree 3 subtrees	20	137	main tree	SA	1357	16968	11	15	Block	Ultra
				DA	5461	58251	13	16	Ultra	Ultra
				SA_DA	1564	9740	11	10	Block	Block
			1st dilution tree	SA	183	2789	8	12	Block	Block
				SA	78	1236	7	11	Block	Block
				SA	35	406	6	9	Block	Block
2 dilution trees 6 subtrees	20	92	main tree	SA	1357	15335	11	14	Block	Ultra
				DA	5461	58251	13	16	Ultra	Ultra
				SA_DA	1564	9740	11	10	Block	Block
			1st dilution tree	SA	186	3136	8	12	Block	Block
				SA	78	1314	8	12	Block	Block
				SA	78	993	7	10	Block	Block
			2nd dilution tree	SA	35	449	7	10	Block	Block
				SA	16	172	6	8	Block	Block
				SA_DA	45	113	6	8	Block	Block

TABLE VIII

RESOURCE UTILIZATION AND LATENCY FOR DIFFERENT RULE SETS

Ruleset	Search engine number					Core num	CLB LUTs (1182240)	CLB Registers (2364480)	BRAM (2160)	URAM (960)	Max frequency (MHz)	Avg latency per packet (ns)	Min latency (ns)	Max latency (ns)
	search tree		PSTSS											
	node	rule	tuple	hash	big rule									
ipc_1k	6	6	8	4	4	55	525844	943611	1457.5	770	215.05	317.6	112	632
acl_1k	6	6	8	4	4	55	515932	923809	1457.5	770	225.53	565.8	155	1955
fw_1k	6	6	8	4	4	55	526254	938989	1457.5	770	225.07	298.6	107	546
ipc_10k	6	6	9	5	5	45	451450	828919	1192.5	810	184.945	646.1	189	1444
acl_10k	6	6	9	5	5	38	283673	558924	1520	608	214.7305	632.9	163	1243
fw_10k	6	6	9	4	5	42	426090	781384	1743	756	186.2197	554.2	188	8060
ipc_32k	6	6	9	4	5	20	201025	372394	1090	880	212.4495	485.8	179	1784
acl_32k	6	6	9	4	5	20	195917	365555	1310	720	201.2072	445.8	209	6844
fw_32k	6	6	9	4	5	20	200530	384555	1350	880	203.252	583.5	172	10421
ipc_100k	6	6	9	4	5	5	60181	99863	1342.5	740	198.4521	475.7	176	2449
acl_100k	6	6	9	4	5	6	66834	111719	990	792	190.15	668.9	200	12196
fw_100k	6	6	9	4	5	8	84839	154080	540	800	210.084	961.0	167	11424

The implementation results for the three dilution solutions are summarized in Table IX. The architecture configuration is the same for these solutions to make a peer-to-peer comparison. More dilution trees would cause a degraded working frequency due to the fact that the structure within a computing core is more complicated. More specifically, multi-level result collections can lead to excessive multiplexers resulting in large routing delays.

C. Data Structure Evaluation

The performance of our TcbTree data structure for various rule sets is calculated and shown in Table X. The tree depth is the maximum depth of the SA, DA, SA_DA trees, and the maximum and minimum values are obtained. The number of nodes refers to the sum of the three tree nodes. The above values are consistent before and after implementation. In aspect of memory usage, compared with the value before implementation, the value of after implementation includes the free space of the node table and the rule table for two reasons: 1) some more space needs to be reserved to support rule updates in case more rules are added than deleted; 2) the depth of on-chip memory can only be a power of 2. Moreover, the minimum size of BlockRAM and UltraRAM is 18Kb and 288Kb respectively, Therefore, in the actual FPGA implementation, many small data structures use BlockRAM

TABLE IX
RESOURCE UTILIZATION FOR DILUTION SCHEMES

Dilution mode	Search engine number					Core num	CLB LUTs (1182240)	CLB Registers (2364480)	BRAM (2160)	URAM (960)	Max frequency (MHz)
	search tree		PSTSS								
	node	rule	tuple	hash	big rule						
1 dilution tree, 2 subtrees	6	6	9	4	5	8	116039	210384	764	832	198.85
1 dilution tree, 3 subtrees	6	6	9	4	5	8	128668	233864	928	832	195.2
2 dilution trees, 6 subtrees	6	6	9	4	5	8	169710	308256	1388	704	181.13

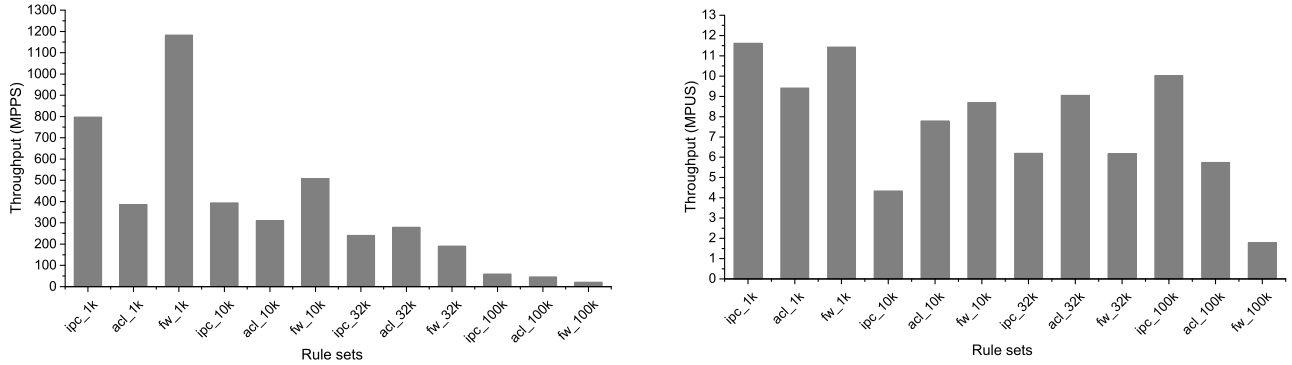

Fig. 10. The throughputs for packet classification and rule update.

TABLE X
IMPLEMENTATION RESULT OF DATA STRUCTURE

Ruleset	Memory usage (KB)		Tree depth		Number of nodes
	before implementation	after implementation	min	max	
ipc_1k	45.74	62.11	8	15	401
acl_1k	41.69	48.99	8	15	326
fw_1k	44.24	62.96	8	15	301
ipc_10k	369.66	444.43	8	15	1494
acl_10k	362.62	623.82	2	15	1723
fw_10k	367.91	541.68	4	5	1931
ipc_32k	1238.27	1616.75	4	6	5321
acl_32k	1241.27	1354.51	1	8	5280
fw_32k	1107.96	1702.70	6	15	3561
ipc_100k	4234.75	6274.25	6	9	31891
acl_100k	3896.17	5285.88	1	15	17360
fw_100k	3342.63	3781.70	6	15	8382

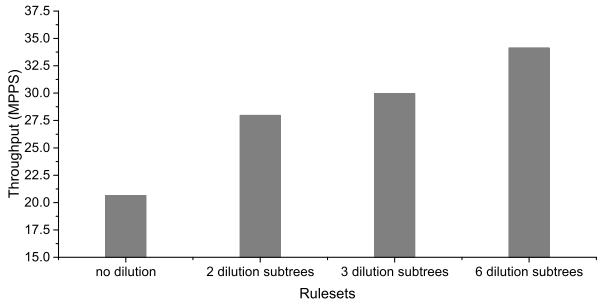

Fig. 11. Throughput in different dilution modes.

and UltraRAM which cannot be further split, so the actual on-chip resource consumption in Table VIII is much larger than the actual required capacity, especially for small-scale rule sets.

D. Performance Evaluation

In this evaluation, performance metrics consist of packet classification throughput and rule update throughput in units of MPPS (Million Packets Per Second) and MUPS (Million Updates Per Second) respectively. The classification throughput is an average value by processing all synthetic packet traces, while the update throughput is obtained by running randomly generated operations including deletion, insertion and modification for a long durance (i.e., 10ms) and calculating the average value.

The packet classification and rule update throughput are calculated by simulation. We first generate the data structure files of a specific rule set. Then we write the testbench which can load the benchmark trace file and rule file, and simulate our architecture with these data structure files at the maximum frequency obtained in Section V-B to perform classification/update with the packets/rules in the trace/rule file. In this way we get the average throughput for the rule set.

The left part of Fig. 10 shows the classification throughput with respect to various rule sets by employing corresponding customized architectures without tree dilution which are listed in Table VI. The performance varies depending on different types of rule sets. FW-related schemes have the best performance for 1k and 10k sizes with the value of 1182.9 MPPS and 508.5 MPPS respectively. ACL-related designs' performance is worst for these two sizes and the reason lies in the fact that most of the rules are distributed in the SA_DA subtree, and a large number of rules have the same SA and DA domain values which could not be further partitioned. In the aspect of large-scale rules, the throughputs for 32k rules are 240.6 MPPS, 279.7 MPPS and 190.9 MPPS separately, and those values for 100k rules are 59.1 MPPS, 45.6 MPPS and 20.6 MPPS with respect to IPC, ACL and FW separately.

The right part of Fig. 10 illustrates the rule update throughput for all kinds of benchmark rule sets. The dynamic update order sent to computing cores is in a broadcast manner since all cores should maintain the same copy of tree structures. The throughput fluctuates but the gap is not as large as that of packet classification. The values are all above 1 MUPS.

The performance varies among different rule sets, as our architecture is not a pure pipeline implementation, therefore the throughput is not proportional to frequency. It is affected by many factors, including number of computing cores, maximum tree depth, number of rules in the leaf, number of TSS tuples, number of conflicting rules for entries in the hash table,

TABLE XI
COMPARISON WITH DECISION TREE AND TCAM BASED APPROACHES ON FPGA

Approaches	Type of ruleset	No.of rules	Device	Dynamic update implementation	Throughput (MPPS)
Proposed TcbTree	IPC	9485	Ultrascale+ VU9P	✓	394.2
	ACL	9163			310.7
	FW	9130			508.5
REC [52]	ACL	9603	Virtex-5 XC5VFX200T	×	323.5
	ACL	9603	Virtex-6 XC6VLX760		388.2
Modified Hypercuts [53]	ACL	10000	Stratix III EP3SE260H780C2	×	433
D^2BS [34]	ACL	9603	Virtex-5 XC5VSX240T	×	263.7
Hypercuts on FPGA [33]	ACL	9603	Virtex-5 XC5VFX200T	✓	250.7
CubeCuts [51]	ACL	9603	Virtex-5 XC5VFX200T	×	368.8
Hypersplit on FPGA [29]	ACL	10000	Virtex-6 XC6VLX760	×	230.5
TCAM on FPGA [56]	/	8192	Virtex-7 XC7V2000T	✓	154
Pseudo-TCAM [57]	/	10000	UltraScale XCVU080	✓	426

TABLE XII
COMPARISON WITH DECOMPOSITION-BASED APPROACHES ON FPGA

Approaches	Type of ruleset	No. of rules	Device	Range search	Dynamic update	Classification throughput (MPPS)	Update throughput (MUPS)
Proposed TcbTree	IPC	1007	Ultrascale+ VU9P	✓	✓	797.4	11.6
	ACL	949				386	9.4
	FW	1037				1182.9	11.4
Updatable Classifier1 [39]	5-tuple	1024	Virtex-6 XC6VLX760	×	✓	690	1
Many-field classifier [35]	15-tuple	1152	Virtex-7 XC7VX1140t	✓	×	500	/
Range-enhanced [36]	12-tuple	3000	Virtex-6 XC6VLX760	✓	×	566	/
Updatable Classifier2 [54]	5-tuple	1024	Virtex-6 XC6VLX760	✓	✓	690	1
StrideBV [38]	5-tuple	512	Virtex-6 XC6VLX760	×	×	390	/

etc. Each rule set has its own unique characteristics, and even for different types of rule sets of similar size, their performance can vary widely.

The effectiveness of tree dilution scheme is shown in Fig. 11. The throughput grows linearly as the number of dilution trees increases. However, this is not the case that more dilution trees are better, because further increasing dilution extent will cause the frequency drop sharply. In the case of two dilution trees with 6 subtrees, the throughput is increased by 65% compared with the original method without tree dilution.

E. Comparison With Related Work

The proposed approach is compared with previous state-of-the-art work based on decision trees and decomposition separately in this section. Since our approach relies on FPGAs equipped with URAMs, it cannot be accommodated by the previously adopted platforms such as Virtex-5 and Virtex-6. Furthermore, the majority of previous designs do not support large-scale rules. Therefore, in this context, we only select the implementation results for 1k and 10k rule sets on the VU9P FPGA for the following comparison.

The comparison with decision tree based approaches is summarized in Table XI. These implementations only adopt 10k ACL rule set as the benchmark, and we select three different types of 10k rule sets for comparison. Our architecture achieves the highest throughput on FW rules. The throughput for ACL rules is on average level. Most importantly, only TcbTree has realized dynamic rule update in the implementation without the need of pre-computing for the updated content of memories. Although [29] is claimed to be able to support on-the-fly rule update, the details about leaf node deletion/creation and internal node update is not discussed, and the corresponding hardware implementation is not proposed. Similarly, [52] only presents the rule deletion/insertion approach for the proposed algorithm, while the implementation of the update scheme on hardware could not be found in the paper. The work in [33] proposes the method of inserting *write bubbles* to pipeline memories to enable rule update. However, the new content of the memory is computed offline rather than changed dynamically according to on-the-fly update orders as our proposed method.

The TcbTree is also compared with the TCAM-simulation methods implemented on FPGA in Table XI. The method in [56] supports TCAM word update and range matching, but can only accommodate up to 16k 150bit words with 100% utilization of on-chip memory and slices. The implementation in [57] achieves high throughput in classification. Although it is claimed to be able to support dynamic update, the update mechanism and performance are not provided. Furthermore, the rule set scale supported by the FPGA implementation is also restricted to 10k.

The comparison with BV decomposition based FPGA designs is shown in Table XII. Since most of these kinds of approaches only support rules up to 1k, our evaluation results for ipc_1k, acl_1k and fw_1k are picked up for this comparison. It can be noted that the throughput for IPC and FW rule sets outperforms other designs while the performance for ACL is not comparable to other designs because of the unevenly distributed search trees. Nevertheless, the average throughput for 1k-scale rules of our architecture is 14.3% and 9.8 times higher than that of [39] in the aspects of classification and update, respectively. It is commendable that only our design and [54] have range search capabilities and support dynamic rule update. StrideBV [38] does not support range match or update. Although the design in [39] can update rule in real time, it only supports prefix match in the SA and DA fields, and exact match in all the other fields.

VI. Conclusion

In this paper, an FPGA-based high-throughput packet classification scheme supporting dynamic rule update called TcbTree is proposed. It is based on a hardware optimized TSS-combined bit-selection tree method. Various sizes of rule sets up to 100k are supported by adopting the scalable parallel method of multiple computing cores. Efficient data structures and centralized storage scheme enable the architecture to get rid of the dependence on specific rules. Moreover, a centralized and uniform hash table approach for PSTSS is proposed to replace the distributed ones which is extremely unfriendly to hardware. The *big leaf* problem in the algorithm is well addressed by the proposed tree dilution scheme. Experimental results on FPGA show that our solution can achieve a high performance in packet classification and real-time rule update. The average throughputs for 1k, 10k, 32k and 100k rule sets are 788.8 MPPS, 404.3 MPPS, 237 MPPS and 41.8 MPPS respectively in aspect of packet classification. The update throughput for all benchmark rule sets is above 1 MUPS. The tree dilution realization has increased the throughput by 65% for packet classification with 100k-scale rules. Compared with other decision tree based designs on FPGAs, only our architecture supports fast rule update without pre-computation of updated memory content. Compared with decomposition-based architectures, the proposed one can fully support range match and large-scale rule sets.

References

[1] W. Li, T. Yang, Y.-K. Chang, T. Li, and H. Li, "TabTree: A TSS-assisted bit-selecting tree scheme for packet classification with balanced rule mapping," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Sep. 2019, pp. 1–8.

[2] N. McKeown and *et al.*, "OpenFlow: Enabling innovation in campus networks," in *ACM SIGCOMM*, 2008, pp. 69–74.

[3] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[4] *ONF Website*. Accessed: 2022. [Online]. Available: https://opennetworking.org/sdn-resources/customer-casestudies/openflow

[5] P. Bosshart and *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[6] H. J. Chao and B. Liu, *High Performance Switches and Routers*. Hoboken, NJ, USA: Wiley, 2007.

[7] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010.

[8] C. R. Meiners, A. X. Liu, and E. Torng, "Topological transformation approaches to TCAM-based packet classification," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, pp. 237–250, Feb. 2011.

[9] A. X. Liu, C. R. Meiners, and E. Torng, "Packet classification using binary content addressable memory," *IEEE/ACM Trans. Netw.*, vol. 24, no. 3, pp. 1295–1307, Jun. 2016.

[10] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2017.

[11] E. Norige, A. X. Liu, and E. Torng, "A ternary unification framework for optimizing TCAM-based packet classification systems," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 657–670, Apr. 2018.

[12] W. Li *et al.*, "A power-saving pre-classifier for TCAM-based IP lookup," *Comput. Netw.*, vol. 164, Dec. 2019, Art. no. 106898.

[13] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *IEEE Trans. Netw.*, vol. 62, no. 6, pp. 1127–1140, Jun. 2013.

[14] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal in/out TCAM encodings of ranges," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 555–568, Feb. 2016.

[15] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 1251–1264, Apr. 2016.

[16] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast TCAM updates," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 217–230, Feb. 2018.

[17] Y. Wan, H. Song, Y. Xu, C. Zhang, Y. Wang, and B. Liu, "Adaptive batch update in TCAM: How collective optimization beats individual ones," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2021, pp. 1–10.

[18] R. Yao *et al.*, "MagicTCAM: A multiple-TCAM scheme for fast TCAM update," in *Proc. IEEE 29th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2021, pp. 1–11.

[19] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *Proc. IEEE 28th Conf. Comput. Commun. (INFOCOM)*, Apr. 2009, pp. 648–656.

[20] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *Proc. ACM SIGCOMM*, 2010, pp. 207–218.

[21] W. Li and X. Li, "HybridCuts: A scheme combining decomposition and cutting for packet classification," in *Proc. IEEE 21st Annu. Symp. High-Perform. Interconnects*, Aug. 2013, pp. 41–48.

[22] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, Oct. 2014, pp. 308–319.

[23] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng, "A sorted-partitioning approach to fast and scalable dynamic packet classification," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1907–1920, Aug. 2018.

[24] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 256–269.

[25] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "A computational approach to packet classification," in *Proc. ACM SIGCOMM*, 2020, pp. 542–556.

[26] B. Pfaff and *et al.*, "The design and implementation of open vSwitch," in *Proc. USENIX NSDI*, 2015, pp. 117–130.

[27] J. Daly *et al.*, "TupleMerge: Fast software packet processing for online packet classification," *IEEE/ACM Trans. Netw.*, vol. 27, no. 4, pp. 1417–1431, Aug. 2019.

[28] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs," in *Proc. ACM SPAA*, 2009, pp. 188–196.

[29] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna, "Multi-dimensional packet classification on FPGA: 100 Gbps and beyond," in *Proc. Int. Conf. Field-Program. Technol.*, Dec. 2010, pp. 241–248.

[30] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2009, pp. 219–228.

[31] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "ParaSplit: A scalable architecture on FPGA for terabit packet classification," in *Proc. IEEE 20th Annu. Symp. High-Performance Interconnects*, Aug. 2012, pp. 1–8.

[32] W. Jiang and V. K. Prasanna, "A FPGA-based parallel architecture for scalable high-speed packet classification," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jul. 2009, pp. 24–31.

[33] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 9, pp. 1668–1680, Sep. 2012.

[34] B. Yang, J. Fong, W. Jiang, Y. Xue, and J. Li, "Practical multituple packet classification using dynamic discrete bit selection," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 424–434, Feb. 2014.

[35] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna, "Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 87–98.

[36] Y.-K. Chang and C.-S. Hsueh, "Range-enhanced packet classification design on FPGA," *IEEE Trans. Emerg. Topics Comput.*, vol. 4, no. 2, pp. 214–224, Apr. 2016.

[37] M. Varvello, R. Laufer, F. Zhang, and T. V. Lakshman, "Multilayer packet classification with graphics processing units," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2728–2741, Oct. 2016.

[38] T. Ganegedara and V. K. Prasanna, "StrideBV: Single chip 400G+ packet classification," in *Proc. IEEE 13th Int. Conf. High Perform. Switching Routing*, Jun. 2012, pp. 1–6.

[39] Y. R. Qu and V. K. Prasanna, "High-performance and dynamically updatable packet classification engine on FPGA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 197–209, Jan. 2016.

[40] C. Li, T. Li, J. Li, Z. Shi, and B. Wang, "Enabling packet classification with low update latency for SDN switch on FPGA," *Sustainability*, vol. 12, no. 8, pp. 1–16, 2020.

[41] W. Li *et al.*, "Tuple space assisted packet classification with high performance on both search and update," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 7, pp. 1555–1569, Jul. 2020.
[42] T. Y. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. IEEE INFOCOM*, Mar. 2000, pp. 1213–1222.
[43] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, 2003, pp. 213–224.
[44] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM*, 1998, pp. 203–214.
[45] F. Geraci, M. Pellegrini, P. Pisati, and L. Rizzo, "Packet classification via improved space decomposition techniques," in *Proc. IEEE INFOCOM*, Mar. 2005, pp. 304–312.
[46] Y. Xu, Z. Liu, Z. Zhang, and H. J. Chao, "High-throughput and memory-efficient multimatch packet classification based on distributed and pipelined hash tables," *IEEE/ACM Trans. Netw.*, vol. 22, no. 3, pp. 982–995, Jun. 2014.
[47] W. Li, D. Li, Y. Bai, W. Le, and H. Li, "Memory-efficient recursive scheme for multi-field packet classification," *IET Commun.*, vol. 13, no. 9, pp. 1319–1325, 2019.
[48] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM*, 1999, pp. 135–146.
[49] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. HOTI*, 1999, pp. 34–41.
[50] W. Li, X. Li, H. Li, and G. Xie, "CutSplit: A decision-tree combining cutting and splitting for scalable packet classification," in *Proc. IEEE INFOCOM*, Apr. 2018, pp. 2645–2653.
[51] Y. Chang and Y. Wang, "CubeCuts: A novel cutting scheme for packet classification," in *Proc. AINA Workshops*, 2012, pp. 274–279.
[52] Y.-K. Chang and H.-C. Chen, "Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA," *Comput. J.*, vol. 62, no. 2, pp. 198–214, Feb. 2019.
[53] A. Kennedy and X. Wang, "Ultra-high throughput low-power packet classification," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 2, pp. 286–299, Feb. 2014.
[54] Y. R. Qu, S. Zhou, and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA," in *Proc. ACM/IEEE ANCS*, Oct. 2013, pp. 125–136.
[55] Z. Shi, H. Yang, J. Li, C. Li, T. Li, and B. Wang, "MsBV: A memory compression scheme for bit-vector-based classification lookup tables," *IEEE Access*, vol. 8, pp. 38673–38681, 2020.
[56] W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs," in *Proc. Archit. Netw. Commun. Syst.*, 2013, pp. 71–82.
[57] W. Yu, S. Sivakumar, and D. Pao, "Pseudo-TCAM: SRAM-based architecture for packet classification in one memory access," *IEEE Netw. Lett.*, vol. 1, no. 2, pp. 89–92, Jun. 2019.
[58] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.

Yao Xin received the M.S. degree from the Department of Electronic Science and Technology, Beihang University, China, in 2011, and the Ph.D. degree from the Department of Electronic Engineering, City University of Hong Kong, Hong Kong, in 2015. He was a Visiting Research Scholar at the University of Southern California, Los Angeles, CA, USA, in 2014. He is currently working as an Assistant Researcher with the Peng Cheng Laboratory, China. His research interests include high-performance VLSI design for networking and deep learning.

Wenjun Li received the B.Sc. degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2011, and the M.Sc. and Ph.D. degrees from Peking University, Beijing, China, in 2014 and 2020, respectively. From 2014 to 2015, he worked as a Research Engineer with Huawei Technologies Company, Ltd. He is currently a Post-Doctoral Research Fellow at Harvard University. Before Harvard University, he was a Post-Doctoral Research Fellow at the Peng Cheng Laboratory. His research interests include SDN, NFV, network measurements, and network algorithms.

Guoming Tang (Member, IEEE) received the bachelor's and master's degrees from the National University of Defense Technology, China, in 2010 and 2012, respectively, and the Ph.D. degree in computer science from the University of Victoria, Canada, in 2017. He was a Visiting Research Scholar at the University of Waterloo, Canada, in 2016. He is currently a Research Fellow at the Peng Cheng Laboratory, China. His research interests include green computing, cloud/edge computing, and networking systems.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences. He is currently an Associate Professor with the Department of Computer Science and Technology, Peking University. He has published more than ten papers in SIGCOMM, SIGKDD, SIGMOD, and NSDI. His research interests include network measurements, sketches, IP lookups, Bloom filters, and KV stores.

Xiaohe Hu received the B.Eng. and Ph.D. degrees from the Department of Automation, Tsinghua University, China, in 2014 and 2020, respectively. He visited the Berkeley NetSys Laboratory, University of California, Berkeley, during his Ph.D. study time. He is currently a Post-Doctoral Researcher with the Department of Computer Science and Technology, Tsinghua University, China. His research interests include network architecture, high performance network processing, and network security.

Yi Wang received the Ph.D. degree in computer science and technology from Tsinghua University in 2013. He is currently a Research Professor with the Institute of Future Networks, Southern University of Science and Technology. His research interests include future network architectures, information centric networking, software-defined networks, and the design and implementation of high-performance network devices.