# Recursive Multi-Tree Construction With Efficient Rule Sifting for Packet Classification on FPGA

Yao Xin, Wenjun Li, Chengjun Jia, Xianfeng Li, *Member, IEEE*, Yang Xu, *Member, IEEE*, Bin Liu, *Senior Member, IEEE*, Zhihong Tian, *Senior Member, IEEE*, and Weizhe Zhang, *Senior Member, IEEE, ACM*

***Abstract*— As a programmable accelerator, SmartNIC provides more opportunities for algorithmic packet classification. Our aim in this work is to achieve both line-speed rule search and efficient rule update, two highly desired metrics for SDN data plane. We leverage the parallelism offered by the FPGA in SmartNIC following an algorithm/hardware co-design paradigm. Particularly, we first design an algorithm that constructs multiple trees for the rule set with a recursive rule sifting process. Unlike traditional space-cutting-based multi-tree construction, our rule sifting mechanism breaks the space constraints of rule-to-tree mapping and enables bounded height on each tree, thus providing the potential of bounded worst-case and line-speed performance. We then design a flexible hardware architecture with multiple systolic arrays that can be implemented in parallel on FPGA. Each systolic array works as a coarse-grained pipeline, and the multiple trees constructed earlier will be mapped onto these pipeline stages. This hardware-software mapping enables bounded worst-case rule searching. Additionally, incremental rule update is achieved simply by traversing the pipeline in one pass, with little and bounded impact on rule searching. Experimental results show that our design achieves an average classification throughput of 600.8/147.5 MPPS and an update throughput of 8.2/5.9 MUPS for 10k/100k-scale 5-tuple and OpenFlow rule sets.**

***Index Terms*— SDN, SmartNIC, packet classification, FPGA.**

Manuscript received 23 March 2023; revised 15 September 2023; accepted 18 October 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Chen. This work was supported in part by the National Key Research and Development Program of China under Grant 2022ZD0115303; in part by the National Natural Science Foundation of China under Grant 62372123, Grant 62102203, Grant U20B2046, Grant 61872212, Grant 62150610497, Grant 62172108, Grant 62272258, Grant 62032013, and Grant 62061160489; in part by the Key-Area Research and Development Program of Guangdong under Grant 2020B0101130003 and Grant 2021B0101400001; in part by the Major Key Project of Peng Cheng Laboratory under Grant PCL2023AS5-1 and Grant PCL2021A15; in part by the 173 Program of China under Grant 2021-JCJQ-JJ-0483; and in part by the China Postdoctoral Science Foundation under Grant 2020TQ0158, Grant 2020M682825, and Grant PC2021037. This paper was presented in part at the ACM/IEEE ANCS, Layfette, IN, USA, December 13–16, 2021 [DOI: 10.1145/3493425.3502752]. *(Corresponding author: Wenjun Li.)*

Yao Xin and Zhihong Tian are with the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510006, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: xinyao@gzhu.edu.cn; tianzhihong@gzhu.edu.cn).

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

Chengjun Jia is with the Department of Automation, Tsinghua University, Beijing 100084, China (e-mail: jcj18@mails.tsinghua.edu.cn).

Xianfeng Li is with the International Institute of Next Generation Internet, Macau University of Science and Technology, Taipa, Macau, China (e-mail: xifli@must.edu.mo).

Yang Xu is with the School of Computer Science, Fudan University, Shanghai 200433, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: xuy@fudan.edu.cn).

Bin Liu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: liub@mail.tsinghua.edu.cn).

Weizhe Zhang is with the Peng Cheng Laboratory, Shenzhen 518055, China, and also with the School of Cyberspace Science, Harbin Institute of Technology, Harbin 150000, China (e-mail: weizhe.zhang@pcl.ac.cn).

Digital Object Identifier 10.1109/TNET.2023.3330381

## I. INTRODUCTION

WITH the development of network function virtualization (NFV) and the rise of software-defined networking (SDN) technology, the virtual functions of the network protocol stack in data center can be updated frequently. As features added and network speeds increased, these network stacks become increasingly complex, and running them on CPU cores takes away much processing power from Virtual Machines (VMs), increasing the cost of running cloud services. With the increasing demand for line-speed of 100GbE in data center, packets have to be handled at a maximum throughput of 148.8 MPPS (where the packets are 64-byte minimum size), making it looks bleak to conduct packet processing on CPU cores. In this context, SmartNICs are gaining popularity due to the high performance and programmability to offload fungible SDN data plane functions [2]. Among these offloaded functions, multi-field packet classification is an essential component, which provides a way to discriminate packets into different "flows" and enables differentiated functionalities, so that all packets belonging to the same flow would be processed in a similar manner by the SmartNIC.

As a widely studied bottleneck, packet classification has attracted extensive research attention [3], [4], [5]. Despite more than twenty years of research efforts, packet classification at line-speed remains to be a challenging problem [6]. Worse still, backed by SDN, the popular OpenFlow puts forward higher requirements for packet classification [7], such as more classification dimensions and faster rule update speed. Thus, hardware using Ternary Content Addressable Memory (TCAM) is still the dominant implementation of packet classification in high-end OpenFlow switches. However, TCAM is very area-inefficient, expensive and power-hungry, which seriously limit its scalability in SmartNIC [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. Meanwhile, with the explosion of network traffic in data center, the number of rules in

classifier increases rapidly, making this classical problem more challenging than ever.

Under this circumstance, Field Programmable Gate Array (FPGA) has been regarded as promising hardware to realize line-speed packet processing in SmartNIC, due to its flexible programmability, high performance and rich capacity [18]. However, none of previous works satisfy the demands of high speed classification and frequent rule update brought by OpenFlow for large-scale rule sets. Specifically, the bit-vector (BV) decomposition consumes a huge amount of distributed RAMs, which restricts the scale of applied vectors [18], [19], [20], [21]. So only *small-scale rule sets* can be supported by BV decomposition architecture, although rule update is well supported. On the other hand, although decision tree based approaches do not have the restriction of rule set scale, they could hardly support *dynamic rule update* (i.e., without pre-computing the memory content or rebuilding the tree/subtree) due to the notorious rule replication problem, which makes it very difficult to ensure atomicity and consistency in the update process, or the overly scattered storage in fully pipelined architectures [22], [23], [24], [25], [26], [27], [28]. Moreover, most of the existing FPGA designs are based on the off-the-shelf algorithms, and the algorithms themselves were not specifically customized according to FPGA characteristics, so the potential of FPGA cannot be brought into full play. Thus, algorithm/hardware co-design schemes are more desired.

In response to the above problems, we propose an algorithm/hardware co-design scheme for parallel and updatable packet classification on FPGA. As an algorithmic solution designed for FPGA from scratch, the proposed software algorithm (i.e., KickTree) fully takes into consideration of the FPGA hardware characteristics, and takes advantage of parallel computing. The proposed hardware architecture (i.e., KickTree_Systolic) adopts a flexible design, which eliminates the need for multi-tree result parsing, and can promptly adapt to various rule sets. The main contributions are as follows:

- Unlike traditional multi-tree building scheme based on static and empirical rule subset partition, KickTree builds multiple trees in a dynamic and recursive manner, which breaks the space constraints of rule-to-tree mapping, and makes it more scalable for OpenFlow rules with arbitrary number of fields.
- Each decision tree completely avoids rule replications by recursively kicking duplicated rules to the remaining subset during the tree construction process, which makes it inherently supportable for incremental rule update. The maximum tree depth and the number of rules contained in each leaf node are both strictly limited, which balances the search time of each tree in hardware and reduces the bottleneck effect, thus providing the potential of bounded worst-case and line-speed performance.
- A hardware architecture with multiple parallel systolic arrays is designed, without the need of parallel result resolving for multiple trees. Each systolic array works as a coarse-grained pipeline, and the constructed multiple trees will be flexibly mapped onto these pipeline stages. Furthermore, incremental rule update is achieved by traversing the pipeline in one pass, with little and bounded impact on rule searching.
- An auxiliary linear search Processing Element (PE) with limited rule capacity is added in hardware to improve the update success rate, so that the hardware can avoid recompiling after rule update failures in all tree-based PEs for an extended period of time, and the overall architecture remains worst-case bounded.

TABLE I
EXAMPLE RULE SET WITH FOUR IPV4 HEADER FIELDS

| *rule id* | *priority* | src_addr | dst_addr | src_port | dst_port | *action* |
|---|---|---|---|---|---|---|
| $R_1$ | 13 | 228.128.0.0/9 | 124.0.0.0/7 | 119:119 | 0:65535 | *action1* |
| $R_2$ | 12 | 223.0.0.0/9 | 38.0.0.0/7 | 20:20 | 1024:65535 | *action2* |
| $R_3$ | 11 | 175.0.0.0/8 | 0.0.0.0/1 | 53:53 | 0:65535 | *action3* |
| $R_4$ | 10 | 128.0.0.0/1 | 37.0.0.0/8 | 53:53 | 1024:65535 | *action4* |
| $R_5$ | 9 | 0.0.0.0/2 | 225.0.0.0/8 | 123:123 | 0:65535 | *action5* |
| $R_6$ | 8 | 107.0.0.0/8 | 128.0.0.0/1 | 59:59 | 0:65535 | *action6* |
| $R_7$ | 7 | 0.0.0.0/1 | 255.0.0.0/8 | 25:25 | 0:65535 | *action7* |
| $R_8$ | 6 | 106.0.0.0/7 | 0.0.0.0/0 | 0:65535 | 53:53 | *action8* |
| $R_9$ | 5 | 160.0.0.0/3 | 252.0.0.0/6 | 0:65535 | 0:65535 | *action9* |
| $R_{10}$ | 4 | 0.0.0.0/0 | 254.0.0.0/7 | 0:65535 | 124:124 | *action10* |
| $R_{11}$ | 3 | 128.0.0.0/2 | 236.0.0.0/7 | 0:65535 | 0:65535 | *action11* |
| $R_{12}$ | 2 | 0.0.0.0/1 | 224.0.0.0/3 | 0:65535 | 23:23 | *action12* |
| $R_{13}$ | 1 | 128.0.0.0/1 | 128.0.0.0/1 | 0:65535 | 0:65535 | *action13* |

With extensive experiments, we show that, even for rule sets up to 100k entries, KickTree can still construct shallow decision trees with a limited number of subsets. The FPGA implementation result shows that it can achieve high performance in both packet classification and real-time rule update for 5-tuple and OpenFlow rule sets. Specifically, the average classification throughput and update throughput for 10k-scale rule sets has reached 600.8 MPPS (Million Packets Per Second) and 8.2 MUPS (Million Updates Per Second), respectively. Even for large-scale 100k rule sets, it can reach an average classification throughput of 147.5 MPPS and an average update throughput of 5.9 MUPS.

The rest of the paper is organized as follows. Section II summarizes background and related work briefly. Section III presents the algorithmic details of our proposed KickTree. Section IV illustrates the hardware architecture. Section V shows experimental results. Finally, Section VI draws the conclusion.

## II. BACKGROUND AND RELATED WORK

### A. The Packet Classification Problem

Packet classification is classifying network traffic 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 $i$th field of the packet header, which could be a prefix, range or 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 rule set with four IPv4 header fields. 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 a high priority rule. Packet classification has been extensively researched in last two decades with numerous

algorithmic approaches proposed, such as decision tree [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], decomposition [42], [43], [44], [45], [46], [47], [48], [49], and Tuple Space Search (TSS) [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61]. Since the current FPGA-based packet classification solution is mainly based on decision trees and decomposition algorithms, in the following two subsections, we will focus on these two approaches to review the related work on FPGA.

### B. FPGA Solutions Based on Decision Trees

The decision tree based architectures can achieve high classification performance by taking advantage of the full pipeline. Qi et al. [22] presented an FPGA-based architecture targeting 100 Gbps packet classification based on HyperSplit [33], which is an efficient pipeline architecture. A node merging algorithm to reduce the number of pipeline stages is also proposed, together with a leaf-pushing algorithm to balance memory allocation. This design can achieve a throughput of 118 Gbps, and more than 50k rules of 5-tuple can be supported with a Virtex-6 FPGA chip.

Chang and Wang [62] proposed a cutting tree scheme called CubeCuts to build a binary decision tree by selecting a subcube and dividing the search space into one inside the subcube and the other outside the subcube. By using the hybrid scheme that combines the CubeCuts and a constrained version of classical HiCuts [29] to allow a small amount of replicated rules, a balance between the required memory usage and the height of the decision tree can be achieved.

The well-known HyperCuts [31] is optimized in several ways by Jiang and Prasanna [26] to reduce the memory requirement and minimize rule duplication. The rule set is partitioned into multiple subsets that are built into multiple optimized decision trees. The trees are mapped into a 2-D multi-pipeline architecture with linear pipelines. This design can store 10k 5-tuple rules in a single Xilinx Virtex-5 FPGA, and sustain 80 Gbps throughput. Another hardware accelerator based on a modified version of HyperCuts is presented by Kennedy and Wang [63], in which a new pre-cutting process is used to reduce the amount of memory needed for large rule sets.

Chang et al. [64] implemented a pipelined architecture using a new recursive endpoint-cutting (REC) decision tree. Considering that the bucket memory requirement becomes a serious problem for the pipeline architecture, a bucket compression scheme to reduce rule duplication in memory bucket pipeline is proposed. The experimental results based on Xilinx Virtex-5/6 FPGA show that much less BRAM is needed by REC than other FPGA-based approaches.

In [65], Tan et al. proposed a hardware solution with multi-pipeline architecture based on the MBitTree [66] algorithm, which can achieve high throughput and support large-scale rule sets. Special logics are designed to traverse the decision tree quickly. Furthermore, they proposed several pipeline optimization techniques to improve the architecture performance. Experimental results show that the architecture can accommodate 100k rule sets on a single NetFPGA_SUME chip, and achieve beyond 250 Gbps throughput on 10k rule sets.

Although the above work could have efficient memory usage and achieve high throughput, the rule duplication problem remains. Concerns of dynamic rule update in hardware, such as how to overcome the effects of rule duplication, how to update leaf or ancestor nodes backwards, or even how to dynamically create new nodes in a fully pipelined architecture, are not addressed. Especially for some optimized and balanced memory algorithms, the real-time update becomes more complicated.

TcbTree [67] introduces TSS [50] to assist tree construction to avoid rule duplication and supports dynamic rule updates. However, it still encounters several problems: i) The partitioning of rule subsets heavily relies on empirical *small fields* characteristics of rules, which seriously limits its scalability for OpenFlow rules with an arbitrary number of fields; ii) The distribution of each rule subset is uneven, which makes the depth of each decision tree very different, and there are a large number of rules in the leaf nodes that are difficult to be further split, resulting in a bulky decision tree; iii) The TSS structure based on hash tables is the bottleneck of the overall performance as it is not efficient in rule search. Therefore, TcbTree has low performance on large-scale rule sets.

Xin et al. [68] proposed a parallel and updatable architecture for decision trees with large-scale rule sets. A multi-level result resolver is constructed for the convergence of multiple trees. However, the complexity of parsing multi-input results is high and takes up more resources, which affects the performance of small-scale rule sets. Furthermore, implementing this architecture is inflexible as it requires changing the result parsing architecture to accommodate rule sets with different numbers of trees. More seriously, all ongoing searches need to be terminated when updating rules, which is not conducive to dynamic update scenarios.

### C. FPGA Solutions Based on BV Decomposition

The BV decomposition is another type of method widely explored for hardware acceleration [43].

Qu and Prasanna [20] presented a 2-D pipelined architecture for packet classification on FPGA based on the Field-Split BV (FSBV) approach, which can achieve high throughput while supporting dynamic rule update. In this architecture, modular self-reconfigurable Processing Elements (PEs) are arranged and each PE accesses its designated memory locally. A couple of optimization techniques are exploited to make tradeoffs between various design parameters and performance metrics. A Virtex-6 FPGA can sustain a throughput of 650 MPPS for classification with 1 million updates/second for a 1k 15-tuple rule set. However, this approach only supports prefix and exact match, and does not support range match.

Aiming to process range fields, Chang and Hsueh [28] proposed two schemes. The first one is similar to StrideBV [19], which is named the range bit vector encoding (RBVE) scheme using specially designed codes to store the pre-computed results in memory. The second scheme uses a simple subrange match in a sequential fashion called sequential subrange

compare (SSC) scheme. By performing experiments on a Virtex-6 FPGA, the proposed designs can handle more than 5k OpenFlow rules and achieve the throughput of 566 MPPS.

In the BV-based algorithms, stringent memory resources in FPGA are wasted to store relatively useless wildcards since there are a lot of wildcards in the rules. To address this issue, Shi et al. [69] presented a memory compression scheme MsBV and constructed a memory-shared homogeneous pipeline architecture MsTP. They utilized a bit matrix and proposed a rearrange technology for MsTP to determine the potential of minimizing memory consumption. The experimental results show that, compared to StrideBV [19] for ACL and OpenFlow rule sets, MsBV has a significant resource reduction in terms of memory, ALUTs and Registers.

According to the issue that the update latency of BV-based approaches is proportional to the number of rules, which can hardly support the SDN switch effectively, Li et al. [21] presented an FPGA solution called SplitBV for efficient rule update by using several distinguishable exact-bits to split the rule set into rule subsets that can be searched in parallel. Experimental results show that SplitBV can reduce the rule update latency by an average of 37% and 41% compared with [20] on two typical rule sets separately.

### D. Summary of Prior Art

Clearly, FPGA-based packet classification has been actively investigated for more than ten years. But as far as we know, none of them can achieve high performance on both lookups and updates for large-scale rule sets. Furthermore, almost all the existing FPGA designs are based on previously proposed algorithms, and the algorithms themselves are not fully customized according to FPGA hardware characteristics. The characteristics of hardware are different from those of software, so the migration and mapping process from software to hardware will sacrifice some intrinsic advantages, and the advantages of FPGA cannot be brought fully into play.

To address these issues, we propose an algorithm that can take advantage of hardware, and design a flexible hardware architecture for this algorithm, which is suitable for fast adaptation to various rule sets and supports rule updates with little impact on the search process, achieving a good algorithm and hardware co-design. Next, we will introduce the details from aspects of algorithmic design and hardware design respectively.

## III. Hardware-Friendly Algorithm Design

Based on the above description, we can conclude that the desired demands of FPGA design for decision trees can be refined as follows: i) multiple independent trees taking advantage of the multi-concurrency feature of hardware; ii) shallow trees with bounded depth and limited number of rules in each leaf node; iii) maximum balance among trees to reduce performance bottlenecks. Besides, in order to support dynamic rule update, building trees without rule replications is also desired. Based on this conclusion, we give decision tree algorithm design (i.e., KickTree) in this section.

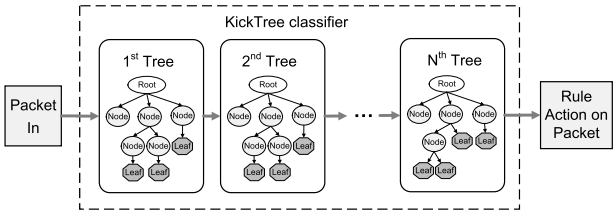



Fig. 1. The algorithmic framework of KickTree.

### A. Ideas & Framework

Previous tree algorithms generally have uncontrollable tree depth, or the number of rules in leaf nodes is not fixed. The depth determines the time latency of node search, while the latter determines the latency of linear search in nodes. These two variables interact and influence each other. Limiting the maximum tree depth will increase the number of terminal node rules, and conversely, limiting the number of terminal node rules will expand the levels of intermediate nodes. However, from the perspective of hardware design, many decision trees with a fixed depth and a small number of rules in leaf nodes are more preferred than a small number of deep and bulky trees. Because the hardware can support concurrent operations of multiple trees, the tree with the worst performance will become the bottleneck of the overall algorithm.

Based on this observation, KickTree adopts a balanced concept to build decision trees instead of empirical and static partitioning of rules subsets. In this approach, we break the restriction of pre-partitioning rule subsets, gather all possible header fields together as a bit-selection pool, and dynamically extract valid bits (not wildcards) each time to build a decision tree in a recursive manner. Before building the tree, the maximum depth and the threshold for the number of rules in each leaf node (i.e., *binth*) are specified to make worst-case bounded. In the process of tree building, the local optimal principle is used to select bits sequentially, and the rules that do not meet the bit-selecting conditions (i.e., the value of the rule in the selection position is a wildcard) or exceed the leaf node rule threshold are removed from the current tree. After the tree is built, if there are remaining rules, we continue to build the decision tree in the same way and retain the rules of being kicked out for constructing the next level of tree. This process continues recursively until there are no rules left. The algorithmic framework of KickTree is shown in Fig. 1. Based on the framework, we next give more algorithm details on each tree construction and multi-tree construction in subsection III-B and subsection III-C respectively.

### B. Bit-Selecting Tree With Efficient Rule Sifting

As each non-wildcard bit can map rules into at most two subsets without any rule replications, each tree is built by selecting non-wildcard bits with the best effort. We focus on two targets, the first is a shallow tree with small tree depth and *binth*, the second is efficient rule sifting to produce a small number of trees with fast convergence.

In order to achieve the first goal, we strictly limit the maximum tree depth and the value of *binth* for each leaf node.

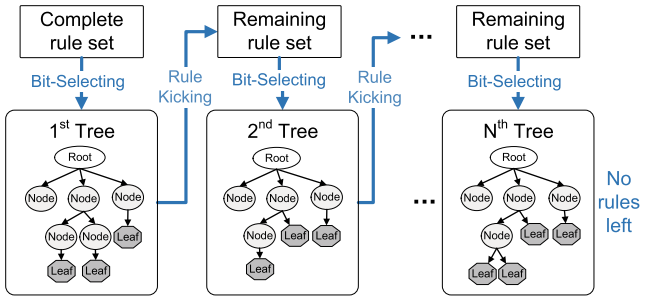



Fig. 2. The construction process of KickTree.

Besides, choosing more bits for each node to be divided can increase the number of forks and reduce the depth of the tree. However, too many bits will increase the logic and storage resources of the hardware and cause performance degradation. Therefore, the choice of the number of bits is a matter of trade-off. To control the width of the tree, we assume that at most $C$ bits are allowed to be selected in each tree node.

In order to build a shallow and balanced decision tree and make the recursive building process of the decision trees converge quickly, in the construction process of each tree, the heuristic local optimal strategy bit-selecting algorithm is utilized to select th e most distinguishing non-wildcard bits.

***Local Optimal Strategy:*** When splitting the rules of a node into child nodes, the method first enumerates all possible combinations (strategies) of fields where bit selection is available. Under each strategy, the first unselected bit in the selected filed is chosen for splitting rules, then it traverses the generated child nodes, counts the sum of the number of rules within the node and the number of rules kicked under the current strategy, and selects the maximum value denoted as $MaxNum[j]$. Equation 1 represents the above steps, where *#Child[j][i]* and *#RuleKick[j]* are the number of rules in $i$th child node and the number of "kicked" (described in subsection III-C) rules for strategy $j$ respectively, where $i = 1, 2, \ldots, 2^C$ ($C$ is the bit count for rule set splitting at each tree node), and $j = 1, 2, \ldots, d^C$ ($d$ is the number of fields where bit selection is available). Next, all strategies are traversed, and the strategy with the minimum $MaxNum$ is selected as the local optimal one as shown in Equation 2.

$$MaxNum[j] = \arg\max_i(\#Child[j][i] + \#RuleKick[j]) \tag{1}$$

$$Strategy = \arg\min_j(MaxNum[j]) \tag{2}$$

### C. Multiple Bounded Trees Without Rule Replications

The construction process of KickTree is shown in Fig. 2 and Algorithm 1. The function *SelectBits*$(R, UB)$ selects bits for each node splitting with the aforementioned local optimal strategy, and $UB$ records the selected bits for current tree building. The classifier construction starts from building the first tree with the complete rule set as the root node, by using the above described single tree algorithm.

In one of the following situations, the method stops the bit selection process: 1) The tree depth reaches the predefined maximum value; 2) The number of rules in the tree node is less than the predefined threshold *binth*; 3) The remaining unselected rule bits share the same value and cannot further separate the rules from each other.

A rule would be "kicked" out of current tree in one of the following two cases: 1) At least one of the values of the selected bits in this rule is a wildcard, which means that this rule will be duplicated in each child if it is not kicked out; 2) This rule cannot be accommodated by a leaf node because of the *binth* space limit.

This recursive manner might result in multiple decision trees with evenly distributed depth and number of leaf node rules. These trees could be implemented on FPGA and run independently and simultaneously to perform packet classification. By minimizing the search delay among different decision trees, this balanced feature can improve the overall classification result generation speed, thereby preventing the so-called bottleneck effect.

### D. A Working Example

This subsection illustrates a KickTree classifier construction example for the 13 rules given in Table I. Assume that the maximum tree depth is two, each internal tree node is allowed to select a maximum of two bits for rule mapping and the *binth* of the leaf node is one. Each port range $R_{ab}$ can be simply transformed to its Longest Common Prefix $LCP_{ab}$, which is the lowest common ancestor of integer $a$ and integer $b$ in a binary prefix tree [70]. For example, for the 8-bit width range $R_{ab} = [97, 127] = [01100001, 01111111]$, its longest common prefix $LCP_{ab} = 011*****$. Thus, each rule in Table I can be converted into a 96-bit width ternary (i.e., 0, 1, *) string as shown in Table II.

The process starts from building the first tree with the complete rule set. Based on the bit-selecting strategy described in subsection III-B, the selected bits for dividing root node are in 1st and 33rd, which would remove $R_8$ and $R_{10}$ as their 33rd bit or 1st bit is a wildcard. This bit-selecting generates three valid mapping nodes. The first valid node $\{R_5, R_6, R_7, R_{12}\}$ then chooses 74th and 75th bits in the same way, and generates three leaf nodes where $R_{12}$ is removed since its corresponding bits are wildcards. The selecting bits for the second valid node $\{R_1, R_2, R_3, R_4\}$ are the same which generate three valid nodes including two leaf nodes. The intermediate node $\{R_3, R_4\}$ reaches the maximum tree depth and the number of rules exceeds *binth*, so the higher priority rule $R_3$ remains as a leaf node. With the rules removed from the first tree as the root node, the second tree is built and the rule $R_{10}$ is removed to build the third tree. Then no rules are left and the process of classifier construction is done. The constructed KickTree classifier is illustrated in Fig. 3.

### E. Packet Classification & Rule Update

*1) Classification:* The classification mechanism for KickTree could be similar to that of other multiple decision tree approaches, that is, incoming packets are searched in all trees and the results are collected to choose the rule with the highest priority. However, in our FPGA implementation, the packet

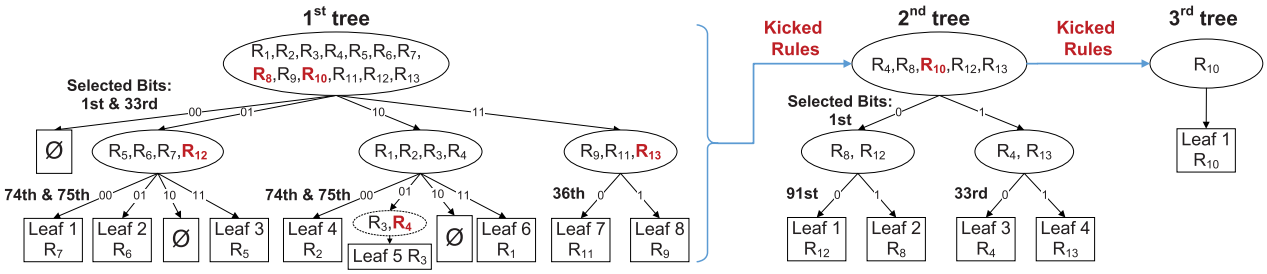



Fig. 3. A working example of KickTree.

searching procedure is in a cascaded manner. Each packet passes through the trees in turn and the matching rule with the highest priority is eventually selected.

*2) Update:* For rule deletion, the process is relatively simple. Trees are traversed by starting the search from the first one. Thanks to the fact that rule replication does not exist in KickTree, once the tree where the rule is located is found, the rule would be deleted from this tree and the remaining trees do not need to be searched further. The above is the same implementation method of software and hardware for deletion, while the insertion process is more complicated.

For rule insertion, search also starts from the first tree. To make the framework extensible to the amount of inserted rules, the algorithm supports dynamically reconstructing trees which is similar to the process of KickTree construction, and this is how our software is implemented. However, dynamic tree reconstruction is inefficient for hardware implementation, so a recursive method is adopted for our systolic array architecture. If the number of rules of the leaf node to be inserted has already reached *binth*, or the current selecting bit is a wildcard, then the next tree is entered to search until a suitable tree is found to insert. To handle the case of failure to insert rules in all trees, we design an update guarantee mechanism which is introduced in Section IV-F.

## IV. Hardware Architecture Design

Based on the proposed KickTree algorithm, we give a flexible hardware architecture design (i.e., KickTree_Systolic) in this section, which can be agilely adapted to various rule sets, get rid of the need for multi-tree result parsing, and support rule updates that have little impact on search.

### A. Top-Level Architecture

The basic hardware architecture for a complete KickTree structure is in the manner of 1-D systolic array in which Processing Elements (PEs) are cascaded in one dimension. Each systolic PE is corresponding to a tree and receives inputs of a command of search/update and a result from the previous PE. An input command consists of a packet/rule along with an operation code of SEARCH (for packet), or DELETE/INSERT (for rule), while a result consists of a matched rule ID and a result code of RULE_FOUND, RULE_NOT_FOUND, UPDATE_SUCCESS, UPDATE_FAILURE, etc. The PE continues to output the result and the corresponding command

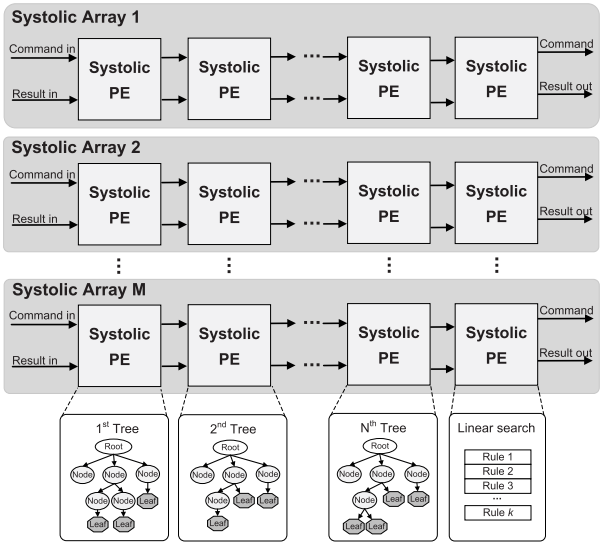



Fig. 4. Top-level architecture design of the system.

after completing the search/update. When the last PE processed the command, the final and optimal result would be produced. All PEs work simultaneously in the form of cascade, and only the adjacent PEs communicate by transferring data, which constitute a coarse-grained pipeline. The internal architecture of each PE is not purely pipelined, in order to facilitate the update of rules.

Compared with the fully concurrent architecture with all PEs processing the same set of packets in parallel, the most significant advantages of the systolic array design lie in: i) Since the resolution of rule priorities is done sequentially during PE pipeline processing, there is no requirement of complex out-of-order result aggregation and synchronization for multiple trees in hardware; ii) It facilitates the gradual rule updates of multiple trees, rather than parallel update, as updating simultaneously within all trees may result in duplicated rule insertions, and it is hard to locate the most appropriate tree to insert a rule. Moreover, an FPGA chip can accommodate multiple systolic arrays as long as the resource is sufficient, each of which can process packets individually. Fig. 4 shows the top-level block diagram of proposed FPGA architecture. $M$ is the number of systolic arrays implemented on an FPGA. This value is almost proportional to the classification throughput because each array works independently, and more arrays mean that more packets can be processed simultaneously.

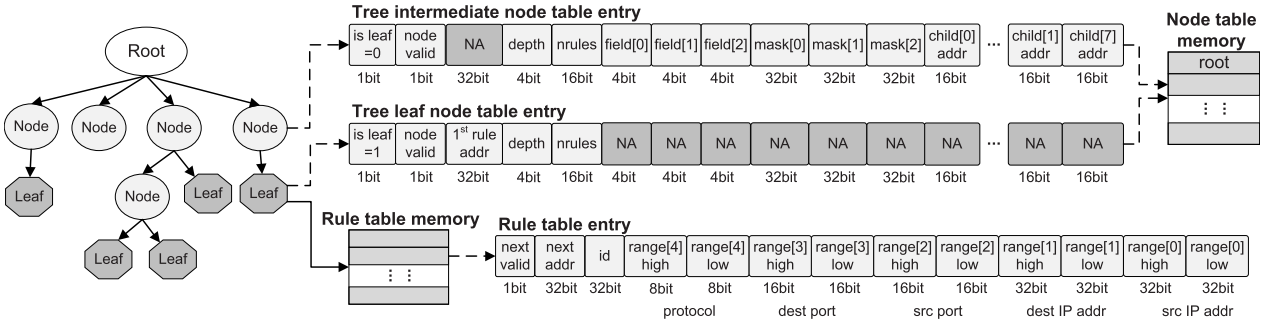



Fig. 5. The data structure of search tree.

### B. Search Tree Data Structure

As shown in Fig. 5, each search tree structure is represented by a set of chain-table-like data structures. The intermediate node (including root node) and leaf node are separately denoted by two types of node table, and the collection of all node tables of a tree is stored in a single node table memory. Each leaf node is associated with the rule table memory caching rule tables describing all rules that belong to this leaf node.

*1) Node Table Entry:* The node tables in Fig. 5 show the case of the selectable bit number being 3 as each node has a maximum of 8 child nodes. The first bit *is_leaf* indicates the current node is an intermediate node or leaf node. The second bit *node_valid* indicates if the current node is valid, which could be modified during an update. This bit being 0 means there is no rule associated with this node. The following bits *depth* and *nrules* represent the node level depth and the number of related rules for the current node. The above bit fields are common to both types of node tables, as the remaining fields are distinct for these node tables. In the intermediate node table, the bits $field[i]$ $(i = 1, 2, 3)$ are utilized to select the cutting rule fields (or dimensions). Bit selection is accomplished by $mask[i]$ $(i = 1, 2, 3)$ in which only 1 bit is 1, which can avoid multi-level multiplexers to select bits in rule prefix. When the 3 bits are determined and spliced together, the next-level child node address indicated by *child_addr* would be determined consequently. In the leaf node table, only $1^{st}\_rule\_addr$ is referred to locate the rule table memory address of the first rule within the leaf rule subset.

*2) Rule Table Entry:* Each rule in tree leaf nodes is represented by a rule table entry. This paper only shows the case of a 5-tuple format, and this table is scalable to other formats of rules. The rule table entries in the rule subset associated with each leaf node are stored consecutively, and the bit *next_valid* indicates whether the next rule is valid. Rule mask is transferred to ranges with two endpoints in advance and recorded in this table. The rule table can be implemented with Ultra RAM, Block RAM, or Distributed RAM, depending on the scale of the rules to be stored in PE.

### C. Systolic PE Architecture

The right part of Fig. 6 shows the detailed design of systolic PE. Since the search result in the current PE needs to be compared with the previous PE result for a specific command, the results and commands need to be aligned by the packet/rule ID. However, the employment of multiple search units in the search tree module brings about a challenge, as they are free to accept packets or rules whenever they are idle and process them at different speeds. In order to address this issue, we set the alignment RAM of command and result. When a result is produced by search tree module, the packet/rule ID is extracted to act as the common access address of alignment RAMs in which commands and previous results are sequentially buffered. In this way, the newly produced results after comparison are synchronized with the commands in interface FIFOs.

### D. Search Tree Module Architecture

The architecture for each tree structure is composed of Node Searcher module and Rule Processor module, which is illustrated in the left part of Fig. 6. The Node Searcher traverses the tree nodes level by level from the root, finding the leaf possibly containing matched rules and locating the start address of the subset associated with this leaf. More specifically, it starts from reading the root node table entry from the first address of node table RAM. The $field[i]$ $(i = 1, 2, 3)$ and $mask[i]$ $(i = 1, 2, 3)$ in the entry would select 3 bits from the packet/rule to get $h$, and the next-level child node address indicated by *child_addr[h]* would be determined by the value of 3-bit $h$. Next, the next-level node entry is obtained from the RAM, and the above process will continue until a leaf node is found, or the next-level child node address is invalid. The rule address and the cached node information of last two levels are transferred to Rule Processor. The Rule Processor searches rules linearly by looking up the rule table RAM and makes actions of search, deletion, or insertion according to the operation code (OP code).

Node Searcher processes rules and packets in the same way, and the only difference is that the input of the rule is the lower endpoint of a range, while Rule Processor handles packet and rule to delete/insert by rule search and rule delete/insert modules respectively. To prevent the memory write/read collision of multiple search units in Node Searcher, only the first search unit processes rule update request with the other ones closed until the current update is completed.

These two modules operate independently, forming a two-stage coarse-grained pipeline. In the Node Searcher module and Rule Search module of Rule Processor, the memory is read sequentially. Since the tree-traversal process is sequential rather than fully pipelined, the tree depth is proportional to the

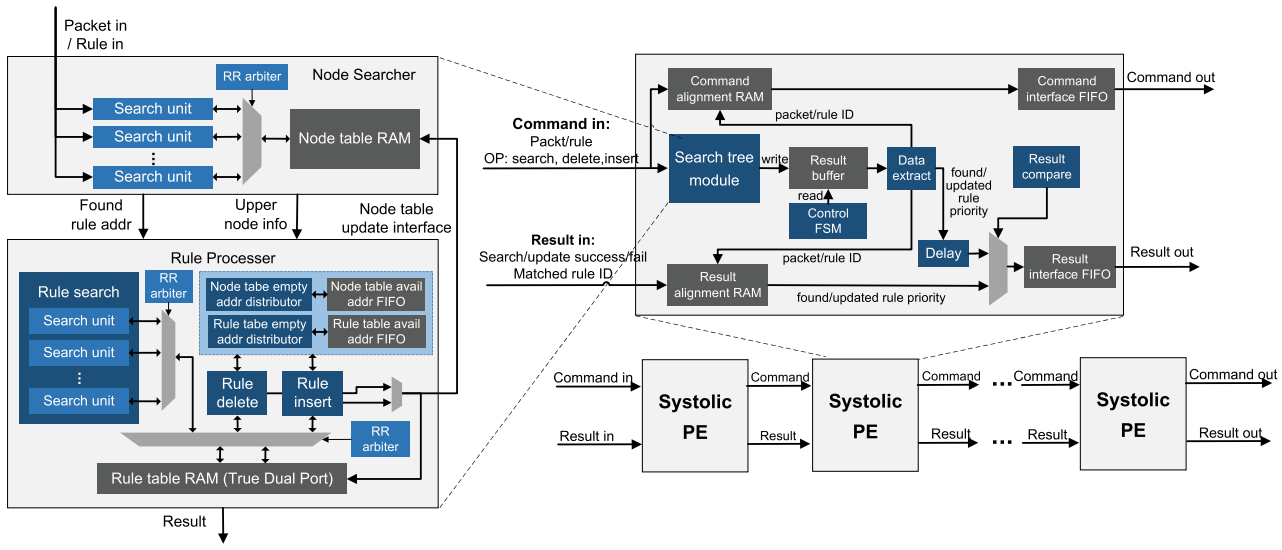



Fig. 6. The architecture of systolic array.

search latency. A large tree depth will result in high search latency. To address this issue, multiple units are implemented to work simultaneously to speed up the search process, and each unit can independently process a packet. Although these units share a common bus and access the memory in different time slots, increasing the number of units could significantly improve memory utilization, because the memory access has a latency and each unit only takes a limited time to query the storage. However, excessive units 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 units. Thus we configure the number that can maximize memory access efficiency with minimum resource usage in the actual implementation. The arbitration between multiple search units in both modules is ensured by Round Robin, as each unit has the same priority.

### *E. Dynamic Rule Update Mechanism*

Our hardware architecture supports dynamic rule update (deletion or insertion) without the need for pre-computation of memory content. Inside the search tree architecture, the Node Searcher would cache complete information of up to two levels of traversed nodes. Therefore, upper-level nodes can be traced back to modify related node table content.

Moreover, in the update operation of the rule table, the rule subset associated with each leaf node will always maintain the order of priority from high to low. Initially, the rules are sorted in order of priority from highest to lowest. When inserting a rule, the priority of each rule will be checked sequentially until a rule with a priority lower than or equal to the rule to be inserted is found, and the rule to be inserted is placed before this rule, by updating the *next_addr* field of the rule table before the rule.

Because dynamic rule update involves the management of storage space, two empty address distributors are designed to interact with rule delete and insert units to recycle and reallocate empty entries in real time for node table memory and rule table memory, respectively. The associated FIFOs record the addresses of the emptied content after deletion and provide available addresses for insertion. In the situation when a rule is being added and the FIFO is empty, an unused address in the memory will be automatically assigned. In order to effectively cope with this situation, the memory corresponding to the node table and the rule table needs to be allocated some idle space in advance.

Next, some specific situations that may occur during the rule update process are enumerated, followed by the corresponding handling methods in our mechanism.

***Insert in the middle of leaf rule subset:*** If the number of rules is less than *binth*, the rule to insert is written in the entry of the address assigned by rule table empty address distributor, and the *next_addr* of the previous rule table is updated to this newly allocated address. The *nrules* of the leaf node table is incremented by 1. Otherwise, a result code of INSET_FALTURE is released.

***Delete in the middle of leaf rule subset:*** The address of the deleted rule is recycled by the rule table empty address distributor, while the content is invalid by default, and the *next_addr* of the previous rule table is updated to *next_addr* of the deleted rule table. The *nrules* of the leaf node's table entry is subtracted by 1.

***The position to insert/delete the rule is the first in the leaf node rule subset:*** After the rule table memory is updated, the $1^{st}\_rule\_addr$ field of the upper-level node table needs to be updated to indicate the address of the first rule after updating.

***The last rule associated with a leaf node is deleted:*** Besides recycling the rule table address, the *node_valid* bit in the node table is set to 0, which indicates this leaf node is invalid without rules. This bit could be certainly reset to 1 if rules are inserted within this leaf.

***Creation of a new leaf node:*** When a rule could be inserted into a lower-level leaf node that the intermediate node does not originally have, a new leaf node table and the associated rule table are created accordingly at the newly assigned addresses by node table empty address distributor and rule table empty address distributor.

**Algorithm 1** Construction of KickTree

**Parameters**: Max tree depth $L$, leaf bin size $B$, bit count $C$ for rule set splitting at each tree node
**Input:** Rule set $R$
**Output:** Decision Trees $T[n]$
**begin**
  Tree count $i = 0$
  /* $UB[k] = 1$ if bit $k$ was used for rule set split */
  Reset $UB$
  **while** $R \neq \varnothing$ **do**
    $R \leftarrow$ KickTree($T[i]$, $R$, $UB$, 1)
    $i \leftarrow i + 1$

**Function** KickTree($T$, $R$, $UB$, $depth$)
**begin**
  **if** $|R| < B$ **then** /* $|R|$ is the size of rule set $R$ */
    $T.Rules \leftarrow R$
    **return** $\varnothing$
  **if** $depth = L$ **then**
    $T.Rules \leftarrow \{r \mid r \in R$ *with top* $B$ *priorities*$\}$
    **return** $R - T.Rules$
  /* $T.Split$: bit positions for rule set splitting */
  $T.Split \leftarrow$ SelectBits($R$, $UB$)
  $UB[b] \leftarrow 1\ \forall b \in T.Split$
  $R_* \leftarrow \{r | r \in R$ *with any wildcard split bit*$\}$
  $R \leftarrow R - R_*$
  /* $T.Children$: the child nodes of the node currently being split in the tree. */
  **foreach** $Child \in T.Children$ **do**
    $R' \leftarrow \{r | r \in$ $R$ *matching* $Child's$ *split bit values*$\}$
    $R_* \leftarrow R_* \cup$ KickTree($Child$, $R'$, $UB$, $depth + 1$)
  $UB[b] \leftarrow 0\ \forall b \in T.Split$ /* Recover UB for parent */
  **return** $R_*$

### F. Update Guarantee Mechanism

As the rule update on the hardware is difficult to achieve the dynamic tree reconstruction mechanism on the software, the update failure case may happen: a rule cannot be inserted into any trees. For the purpose of improving the overall rule insertion success rate, the last PE of each systolic array is set to linear search with relatively relaxed *binth* restrictions, which can accommodate rules of insertion failure. This feature is shown in Fig. 4. In practice, the situation that all previous PE update failures rarely occur, especially when the number of PEs is large. For example, 9 PEs guarantee a better update success rate than 3 PEs.

Nevertheless, note that when the rules for linear search in the last PE accumulate to a certain extent over a long period, it is necessary and recommended to reconstruct the classifier as a whole and rewrite the memory in each systolic array, to avoid this auxiliary update module from turning into the throughput bottleneck of the system. Therefore, the overall systolic architecture is still worst-case bounded.

### G. Result Consistency Preservation

If this architecture were to perform rule updates at the same time as classifying the packet, there could be a risk of introducing inconsistencies in classification results. However, if all PEs in the systolic array run only one type of task in order to avoid inconsistency, the performance will be significantly reduced. To cope with this situation, an isolation mechanism for two tasks is designed in the search tree architecture inside the PE. Update commands will not be accepted if there is an outstanding classification process within a PE, and vice versa. In this way, tasks of a different category in the array can never catch up with the previous task to avoid mutual interference between them, and the latency sacrificed for isolation is reduced to the span of one PE, which has less impact on overall performance.

## V. FPGA Implementation Result

### A. Experiment Setup

In hardware implementation, we only focus on the performance of relatively large-scale rule sets. Three types of 5-tuple rule sets are generated by ClassBench [71] using default parameters, which are ACL, FW and IPC, and the size of each type of rule set includes 10k and 100k. For each size, 12 rule sets based on 12 seed parameter files (i.e, 5 ACL, 5 FW, and 2 IPC) are generated. The software source code of KickTree can be downloaded from the website (https://www.wenjunli.com/KickTree).

Furthermore, to verify the scalability of our architecture for OpenFlow rules which contain more fields, ClasshBench-ng [72] are utilized to generate OpenFlow-like rule sets of 10k and 100k sizes, each with 2 seed parameter files. The source code of the modified ClassBench-ng can also be downloaded from the website mentioned above. The rule has 9 fields: source IP address, destination IP address, source port, destination port, protocol, in port, source MAC address, destination MAC address, and Ethernet type. We have supplemented the trace-generation capability for ClasshBench-ng to generate corresponding trace files. Accordingly, we extend the data structure shown in Fig. 5 to support OpenFlow rules, and make minor changes to the hardware logic.

We first evaluate the number of subsets of 5-tuple rules for KickTree under two parameters with different values: the maximum depth of the tree and *binth*, with the number of selection bits fixed at 3. In subsequent evaluations for FPGA implementation, the maximum tree depth, selection bit number and *binth* are set to 8, 3, and 10, respectively. We further reduce the number of trees by increasing the *binth* and maximum depth settings of the trees generated later. This

TABLE II
TERNARY STRING FOR EXAMPLE RULE SET

| *rule* *id* | src_addr (SA) 1-32th bits | dst_addr (DA) 33-64th bits | src_port (SP) LCP 65-80th bits | dest_port (DP) LCP 81-96th bits |
|---|---|---|---|---|
| $R_1$ | 111001001********************** | 0111110********************** | 0000000001110111 | ********** |
| $R_2$ | 110111110********************** | 0010011********************** | 0000000000010100 | ********** |
| $R_3$ | 10101111********************** | 0****************************** | 0000000000110101 | ********** |
| $R_4$ | 1****************************** | 00100101********************** | 0000000000110101 | ********** |
| $R_5$ | 00****************************** | 11100001********************** | 0000000001111011 | ********** |
| $R_6$ | 01101011********************** | 1****************************** | 0000000000111011 | ********** |
| $R_7$ | 0****************************** | 11111111********************** | 0000000000011001 | ********** |
| $R_8$ | 0110101********************** | ****************************** | ********** | 0000000000110101 |
| $R_9$ | 101********************** | 111111********************** | ********** | ********** |
| $R_{10}$ | ****************************** | 1111111********************** | ********** | 0000000001111100 |
| $R_{11}$ | 10****************************** | 1110110********************** | ********** | ********** |
| $R_{12}$ | 0****************************** | 111****************************** | ********** | 0000000000010111 |
| $R_{13}$ | 1****************************** | 1****************************** | ********** | ********** |

TABLE III
HARDWARE CONFIGURATIONS FOR DIFFERENT RULE SETS

| Rule set | Tree number | Number of nodes/rules | | | | | | RAM depth (bit) | | | | | | | RAM type | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1st tree | | 2nd tree | | 3rd tree | | 1st tree | | | 2nd tree | | 3rd tree | | 1st tree | | | 2nd tree | | 3rd tree | |
| | | node | rule | node | rule | node | rule | u_node | l_node | rule | node | rule | node | rule | u_node | l_node | rule | node | rule | node | rule |
| acl1_10k | 6 | 2791 | 7506 | 642 | 1738 | 110 | 454 | 10 | 11 | 13 | 10 | 11 | 7 | 10 | Block | Ultra | Ultra | Block | Ultra | Block | Ultra |
| acl2_10k | 9 | 2406 | 7270 | 530 | 1355 | 106 | 535 | 0 | 12 | 13 | 10 | 11 | 7 | 10 | NA | Ultra | Ultra | Ultra | Ultra | Ultra | Ultra |
| acl3_10k | 9 | 2085 | 6057 | 379 | 882 | 187 | 1027 | 0 | 12 | 13 | 9 | 10 | 9 | 11 | NA | Ultra | Ultra | Ultra | Ultra | Ultra | Ultra |
| acl4_10k | 9 | 1990 | 6145 | 412 | 1128 | 128 | 724 | 0 | 11 | 13 | 9 | 11 | 8 | 10 | NA | Ultra | Ultra | Ultra | Ultra | Ultra | Ultra |
| acl5_10k | 5 | 1415 | 4070 | 211 | 615 | 39 | 126 | 0 | 11 | 13 | 9 | 10 | 6 | 7 | NA | Ultra | Ultra | Ultra | Ultra | Ultra | Ultra |
| ipc1_10k | 9 | 1570 | 3869 | 765 | 1979 | 328 | 1461 | 0 | 11 | 13 | 10 | 11 | 9 | 11 | NA | Ultra | Ultra | Ultra | Ultra | Ultra | Ultra |
| ipc2_10k | 3 | 1171 | 8886 | 529 | 1114 | 1 | 1 | 0 | 11 | 14 | 10 | 11 | 1 | 6 | NA | Ultra | Ultra | Ultra | Ultra | Dist | Block |
| fw1_10k | 8 | 4681 | 6631 | 665 | 2454 | 21 | 78 | 0 | 13 | 13 | 10 | 12 | 5 | 7 | NA | Ultra | Ultra | Ultra | Ultra | Block | Ultra |
| fw2_10k | 4 | 2341 | 8352 | 892 | 1270 | 3 | 13 | 0 | 12 | 14 | 10 | 11 | 2 | 5 | NA | Ultra | Ultra | Ultra | Ultra | Dist | Block |
| fw3_10k | 8 | 1912 | 6372 | 681 | 2191 | 43 | 248 | 0 | 11 | 13 | 10 | 12 | 6 | 8 | NA | Ultra | Ultra | Ultra | Ultra | Block | Ultra |
| fw4_10k | 9 | 3457 | 6635 | 546 | 1464 | 28 | 269 | 0 | 12 | 13 | 10 | 11 | 5 | 9 | NA | Ultra | Ultra | Ultra | Ultra | Block | Ultra |
| fw5_10k | 9 | 3275 | 5463 | 689 | 2930 | 27 | 85 | 0 | 12 | 13 | 10 | 12 | 5 | 7 | NA | Ultra | Ultra | Ultra | Ultra | Block | Ultra |
| of1_10k | 7 | 4023 | 9628 | 520 | 1116 | 104 | 259 | 0 | 12 | 14 | 10 | 11 | 7 | 9 | NA | Ultra | Ultra | Ultra | Ultra | Block | Block |
| of2_10k | 6 | 4221 | 9709 | 48 | 144 | 9 | 32 | 8 | 12 | 14 | 6 | 8 | 4 | 6 | Block | Ultra | Ultra | Block | Ultra | Block | Block |
| acl1_100k | 8 | 38934 | 86055 | 3185 | 9684 | 638 | 3264 | 13 | 15 | 17 | 12 | 14 | 10 | 12 | Block | Block | Ultra | Ultra | Ultra | Distr | Dist |
| acl2_100k | 21 | 5259 | 15781 | 2943 | 9331 | 2169 | 12407 | 0 | 13 | 14 | 12 | 14 | 12 | 14 | NA | Ultra | Ultra | Ultra | Ultra | Ultra | Ultra |
| acl3_100k | 8 | 35228 | 82351 | 6930 | 15561 | 198 | 1262 | 12 | 15 | 17 | 13 | 14 | 8 | 12 | Ultra | Ultra | Ultra | Block | Block | Dist | Block |
| acl4_100k | 9 | 35556 | 82970 | 3811 | 10776 | 581 | 3391 | 12 | 15 | 17 | 12 | 14 | 10 | 12 | Block | Block | Ultra | Block | Ultra | Dist | Ultra |
| acl5_100k | 3 | 30654 | 92449 | 654 | 2306 | 1 | 1 | 0 | 15 | 17 | 10 | 12 | 1 | 7 | NA | Block | Ultra | Block | Block | Dist | Dist |
| ipc1_100k | 6 | 37483 | 90553 | 4030 | 8480 | 43 | 160 | 13 | 15 | 17 | 12 | 14 | 6 | 8 | Ultra | Block | Ultra | Block | Ultra | Dist | Dist |
| ipc2_100k | 4 | 9363 | 81920 | 2482 | 11009 | 4680 | 7071 | 0 | 14 | 17 | 12 | 14 | 13 | 13 | NA | Ultra | Ultra | Dist | Block | Block | Ultra |
| fw1_100k | 11 | 37449 | 66448 | 6239 | 18121 | 824 | 3109 | 13 | 15 | 17 | 13 | 15 | 10 | 12 | Ultra | Block | Ultra | Block | Ultra | Dist | Ultra |
| fw2_100k | 5 | 18473 | 80808 | 7207 | 13059 | 492 | 2069 | 11 | 14 | 17 | 13 | 14 | 9 | 12 | Block | Block | Ultra | Ultra | Block | Dist | Block |
| fw3_100k | 8 | 18725 | 63115 | 5195 | 15224 | 467 | 2317 | 12 | 14 | 16 | 13 | 15 | 9 | 12 | Block | Block | Ultra | Ultra | Ultra | Block | Block |
| fw4_100k | 16 | 18925 | 57688 | 3411 | 7953 | 1179 | 3896 | 12 | 14 | 16 | 12 | 14 | 11 | 12 | Block | Ultra | Ultra | Block | Ultra | Block | Block |
| fw5_100k | 11 | 37448 | 54157 | 9104 | 25692 | 471 | 2578 | 13 | 15 | 16 | 14 | 15 | 9 | 12 | Block | Ultra | Ultra | Block | Ultra | Block | Block |
| Generic | 21 | 38934 | 92449 | 9104 | 25692 | 4680 | 12407 | 13 | 15 | 17 | 14 | 15 | 13 | 14 | Block | Ultra | Ultra | Block | Ultra | Block | Block |
| of1_100k | 8 | 34146 | 77698 | 3706 | 7435 | 535 | 1384 | 11 | 15 | 17 | 12 | 13 | 10 | 11 | Dist | Block | Ultra | Ultra | Ultra | Dist | Ultra |
| of2_100k | 7 | 35843 | 94183 | 422 | 1338 | 222 | 659 | 12 | 15 | 17 | 10 | 11 | 9 | 10 | Block | Block | Ultra | Block | Ultra | Dist | Block |

adjustment is based on an observation: most of the rules are concentrated in the first three trees.

In most cases, the first PE contains the majority of the rules and therefore has the highest storage requirement. On the other hand, the storage depth for FPGA is a power of 2. For example, if $2^n + 1$ entries are needed, then a storage depth of $2^{(n+1)}$ is required to be reserved, which is wasteful for memory resources, especially for node tables that are added infrequently. To deal with this situation, we divide the node table RAM in the first PE into two parts: upper_half and lower_half. Among them, upper_half stores entries whose address is higher than $2^n$, while the depth of lower_half is $2^n$. This partitioning approach for the first PE can achieve the maximum benefit. In contrast, for the other PEs, the benefit is not significant at the cost of higher complexity.

The evaluation platform is Xilinx Virtex UltraScale+ VU9P FPGA, which is equipped with a large amount of Ultra RAMs. By taking advantage of this property, multiple systolic arrays can be instantiated to explore a high performance for packet classification. The number of search units in Node Search and Rule Processor in each systolic PE is set to 5 and 6 separately. The optimal number of search units is determined through trial and error with extensive experiments.

In order to explore the characteristics of various rule sets and achieve the optimal performance for a specific rule set, hardware configurations for different sizes and types of rule sets have been customized and finely tuned, which are listed in Table III. It mainly shows the configurations for the first three trees for two reasons. First, the first three trees contain the vast majority of rules, so their configuration is representative. Second, listing the configurations of all the trees would be

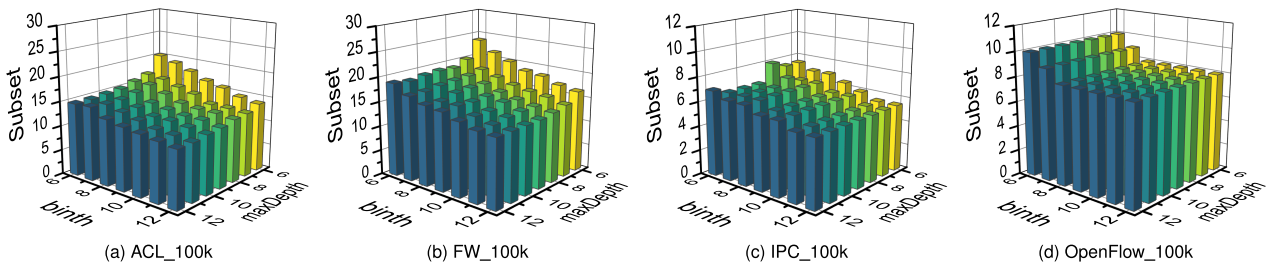



Fig. 7. Number of subsets for different rule sets.

TABLE IV
RESOURCE UTILIZATION FOR DIFFERENT RULE SETS

| Rule set | Array num | CLB LUTs (1182240) | CLB Registers (2364480) | BRAM (2160) | URAM (960) | LUTRAM (591840) | Max frequency (MHz) |
|---|---|---|---|---|---|---|---|
| acl1_10k | 20 | 71.87% | 51.67% | 72.64% | 41.67% | 4.49% | 201.25 |
| acl2_10k | 15 | 79.29% | 54.84% | 96.74% | 50.00% | 1.19% | 225.28 |
| acl3_10k | 14 | 77.08% | 53.08% | 63.75% | 64.17% | 3.49% | 203.21 |
| acl4_10k | 15 | 80.99% | 53.94% | 77.29% | 81.25% | 0.00% | 222.32 |
| acl5_10k | 25 | 73.87% | 52.11% | 53.73% | 72.92% | 3.86% | 250.50 |
| ipc1_10k | 14 | 75.31% | 50.55% | 69.54% | 81.67% | 0.00% | 250.12 |
| ipc2_10k | 30 | 51.92% | 37.64% | 47.13% | 62.50% | 2.02% | 251.32 |
| fw1_10k | 15 | 72.46% | 51.61% | 64.14% | 50.00% | 4.07% | 250.06 |
| fw2_10k | 30 | 69.81% | 50.50% | 56.16% | 87.50% | 3.68% | 250.44 |
| fw3_10k | 15 | 71.92% | 51.11% | 53.03% | 50.00% | 4.86% | 250.31 |
| fw4_10k | 15 | 75.29% | 50.62% | 74.72% | 70.00% | 0.00% | 250.56 |
| fw5_10k | 15 | 82.63% | 57.76% | 43.31% | 75.00% | 4.07% | 200.00 |
| of1_10k | 11 | 72.43% | 41.31% | 81.09% | 40.10% | 0.26% | 200.64 |
| of2_10k | 11 | 61.90% | 35.65% | 68.10% | 34.38% | 0.74% | 200.08 |
| acl1_100k | 6 | 42.86% | 23.30% | 85.00% | 97.50% | 25.70% | 179.23 |
| acl2_100k | 6 | 79.19% | 56.32% | 87.78% | 62.50% | 4.54% | 201.13 |
| acl3_100k | 5 | 29.40% | 19.11% | 83.22% | 85.42% | 8.25% | 200.00 |
| acl4_100k | 6 | 28.63% | 23.11% | 88.89% | 97.50% | 16.13% | 189.07 |
| acl5_100k | 6 | 9.45% | 7.56% | 77.92% | 80.00% | 2.09% | 190.73 |
| ipc1_100k | 6 | 38.38% | 21.52% | 86.39% | 90.00% | 27.57% | 176.09 |
| ipc2_100k | 6 | 29.11% | 15.73% | 92.64% | 90.00% | 22.74% | 185.39 |
| fw1_100k | 5 | 50.56% | 27.18% | 92.01% | 89.58% | 30.43% | 184.06 |
| fw2_100k | 6 | 21.20% | 14.41% | 67.08% | 90.00% | 5.66% | 200.00 |
| fw3_100k | 5 | 37.43% | 25.00% | 74.77% | 87.50% | 7.18% | 194.29 |
| fw4_100k | 7 | 74.46% | 50.93% | 78.10% | 70.00% | 12.78% | 201.25 |
| fw5_100k | 7 | 53.18% | 35.28% | 91.23% | 93.33% | 11.97% | 185.56 |
| of1_100k | 5 | 52.98% | 27.31% | 75.69% | 93.75% | 24.67% | 157.26 |
| of2_100k | 5 | 45.89% | 23.30% | 86.81% | 88.54% | 20.09% | 155.3 |

tedious and take up too much space. The RAM types include Distributed (Dist for short), Block and Ultra. The u_node and l_node denote upper_half node table RAM and lower_half node table RAM respectively.

On the other hand, a generic configuration that can accommodate all 5-tuple benchmark rule sets is also implemented. In the generic architecture, the maximum number of PEs is required, with each PE having a storage capacity equal to the maximum value of the corresponding PEs among all rule sets. In this case, a systolic array consumes more resources, and the number of arrays that can be accommodated by an FPGA is lower than other configurations.

The use of three kinds of RAM in the architecture follows two principles: 1) The Ultra RAM, Block RAM, and Distributed RAM are selected mainly depending on the scale of the nodes or rules to be stored in each PE. Specifically, the minimum depth of Ultra RAM, Block RAM, and Distributed RAM on FPGA is 4096, 512, and 64, respectively. The node table RAM or rule table RAM could adopt Ultra RAM when the number of entries is large. It could adopt Block RAM when the entry number is moderate, or use Distributed RAM when the number is small. 2) The resource usage of the three kinds of RAM on the chip should be balanced. If one type of RAM is consumed too much, some of it will be replaced with other RAM, regardless of the actual depth requirements. In summary, we first roughly allocate three kinds of memory according to principle 1 and then fine-tune them according to principle 2.

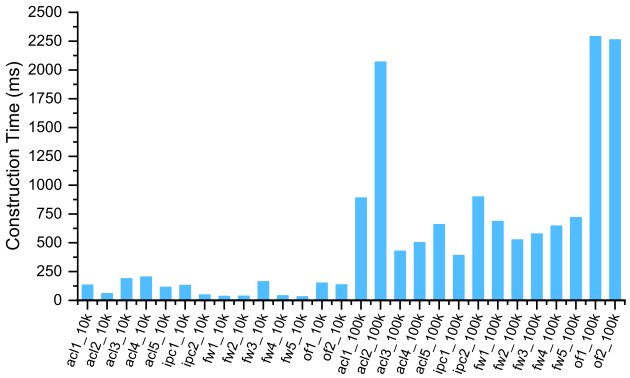



Fig. 8. The construction time for 10k and 100k rule sets.

### B. Number of Subsets

The number of subsets is the number of trees in KickTree, which is determined by a set of parameters, such as maximum tree depth and the threshold of the number of leaf node rules *binth*. The number of bits selected per node is fixed at 3. Fig. 7 shows the number of generated trees in KickTree under different parameter combinations for 100k rules. Specifically, the smaller the depth of the tree, and the smaller the *binth*, the greater the number of trees. By controlling the above two parameters, the actual number of trees can be maintained at an acceptable level for FPGA implementation. Obviously, KickTree can produce a relatively stable number of subsets by adjusting the construction parameters for large size rule sets, with the help of our heuristic optimal local strategy for bit selection. More efficient strategies would be explored in future work to further reduce the number of subsets. Moreover, through the observation of the experimental results, it can be found that most of the rules are concentrated in the first two or three trees, and the number of subsequent trees can be reduced by stepwise setting parameters, which is also the strategy we adopt in the actual hardware implementation.

### C. Construction Time

Fig. 8 shows the construction time of data structures for various rule sets. All experiments are conducted on a PC

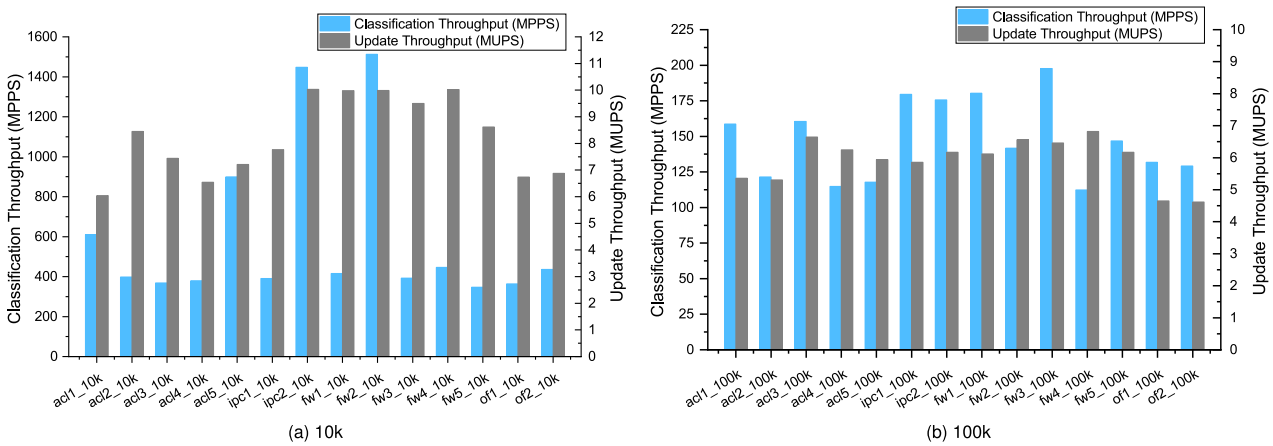



Fig. 9. Classification and update throughput for 10k and 100k rule sets.

equipped with Intel Core i7 CPU@3.2GHz and 16G memory. Even for 100k-scale rule sets, most of the reconstruction processes can be completed within 1 second. Having a shorter construction time holds a significant advantage when the long-term incremental update causes the last PE to be too large and the data structure needs to be restructured.

### *D. Resource Utilization*

Table IV summarizes the number of generated systolic arrays, resource usage and maximum frequency of hardware implementations for various rule sets after synthesis, placement and routing using Vivado 2021.2 design tool. The OpenFlow rule sets begin with "of". The "Array num" is the maximum number of systolic arrays the FPGA can accommodate.In fact, the consumption of hardware resources by a single systolic array corresponding to the rule set of 10k and 100k is very different, which leads to the difference in the corresponding maximum number of arrays under the limitation of hardware resources because we want to make full use of FPGA resources. Therefore, the overall gap in resource consumption between different rule sets is not large.

Since our hardware architecture supports real-time rule update, sufficient spare storage space should be allocated for the search tree in each PE at the beginning, in case there are more insertions than deletions. Naturally, it can be noted that the memory including Ultra RAM and Block RAM is the most consumed FPGA resource. Thus an FPGA platform equipped with Ultra RAM is more suitable for our dynamically updatable architecture.

### *E. Performance Evaluation*

In this subsection, we evaluate our FPGA implementation in performance of throughput, which consists of packet classification throughput and rule update throughput in units of MPPS (Million Packets Per Second) and MUPS (Million Updates Per Second) respectively. These two kinds of throughput are calculated by simulation. We first generate the data structure files of a specific rule set. Then we simulate our architecture with these data structure files at the maximum frequency obtained in Section V-D to perform classification/update with the packets/rules in the trace/rule file. 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 endurance (i.e., 100ms) and calculating the average value.

The Fig. 9(a) and Fig. 9(b) show the classification throughput and update throughput with respect to benchmark 10k and 100k rule sets respectively. The performance varies according to different rule sets. For 5-tuple rule sets, the classification throughput is distributed between 347.5 MPPS and 1513.2 MPPS in terms of 10k scale. The storage resource requirement is relatively small, and the resource bottleneck lies in logic (i.e., LUTs). The result can be roughly summarized as: the fewer trees are generated, the more systolic arrays can be accommodated, and the better the performance. On the other side, the range of classification throughput for 100k-scale rules does not fluctuate much, which is between 112.3 MPPS and 197.7 MPPS. In the aspect of rule update performance, the throughput values for various rule sets are similar, ranging from 5.3 MUPS to 10 MUPS. For OpenFlow rule sets, the average throughput of classification/update is 400 MPPS/6.8 MUPS for 10k size, and 130 MPPS/4.6 MUPS for 100k size. Its performance does not lag far behind that of 5-tuple because it does not generate too many trees.

It is worth noting that the above classification and update performance are measured separately without mixing, because in different practical scenarios, the update frequency requirements are not fixed, depending on the actual business needs. Moreover, interspersed updates between classifications are well supported by the proposed architecture without the problem of inconsistency of classification results, thanks to the isolation design of the two tasks in the PE architecture. To illustrate this feature, we select two sizes of rule sets (i.e., acl1_10k, acl1_100k) to evaluate the classification performance at different rule update rates, as shown in Fig. 10. The update rate gradually decreased from one update per 10 packets to one update per 10000 packets. It can be seen that after one update per 500 packets, the classification performance is almost the same as that without an update.

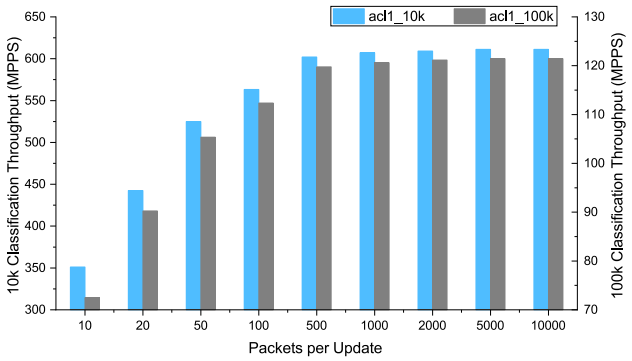



Fig. 10. Classification throughput at different update rates.

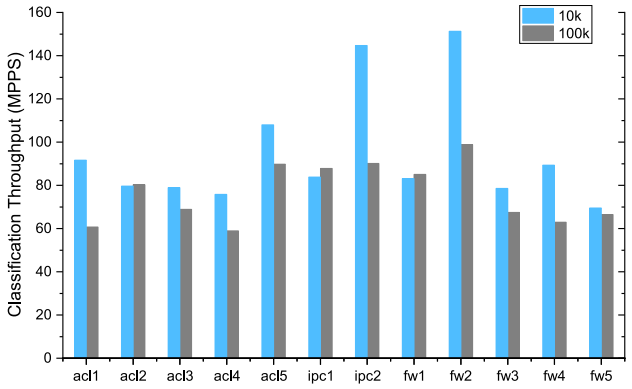



Fig. 11. The classification throughput for the generic hardware configuration.

In other words, for the 10k rule set, we can simultaneously support about 600 MPPS classifications and 1.2 million rule updates per second.

Fig. 11 visualize the classification throughput of various rule sets under the generic configuration. Note that the overall performance across all rule sets has experienced a decline, particularly for the 10k-scale rule sets, dropping to a similar magnitude as the 100k-scale rule sets. This compromise in performance was made in order to achieve a solution that offers greater flexibility and applicability.

The latency information used by the architecture to classify each packet is also calculated to provide a more comprehensive analysis, including average, minimum, and maximum latencies, which is depicted in Fig. 12. It is obvious that our architecture does not have an advantage in terms of latency compared with tens of nanosecond delays of TCAMs, due to the serial multi-PE design. However, supporting a flexible architecture entails certain trade-offs. Nonetheless, this trade-off is sufficient to meet the microsecond requirements of the NIC. We will focus on reducing latency as the next step in our research.

### *F. Comparison With Related Work*

In this subsection, the FPGA implementation of the proposed KickTree is compared with previous well-known works based on decision trees and decomposition, which is summarized in Table V. The throughput is the average of the throughput corresponding to different types of rule sets in Fig. 9(a). 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,

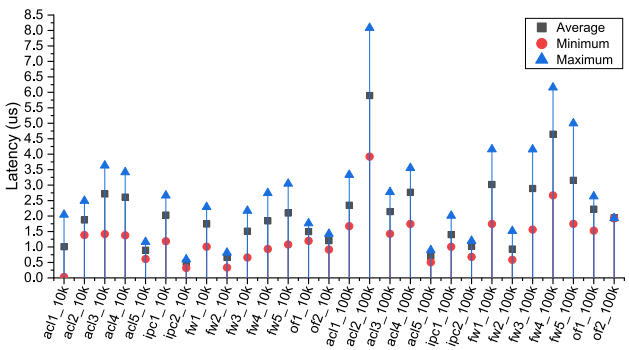



Fig. 12. Classification latency for 10k and 100k rule sets.

many previous designs do not support large-scale rules and some decision tree based implementations only adopt 10k ACL as the benchmark, while the decomposition-based ones only support small-scale rules. Therefore, in order to achieve a unified comparison, we select three different types of 10k rule sets on the VU9P FPGA for the following comparison.

In the comparison with FPGA designs based on decision trees, our architecture achieves the highest throughput on all types of rule sets except ACL. Compared with other two updatable architectures, the proposed KickTree_Systolic outperforms TcbTree in classification throughput and is much superior to KickTree_Parallel in update throughput. More importantly, it supports rule update with only little impact on the search, which is more suitable for practical application scenarios. Although [22] is claimed to be able to support on-the-fly rule update, the details about leaf node deletion/creation and intermediate node update is not discussed, and the corresponding hardware implementation is not proposed. Similarly, [64] 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 [26] 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 implementation of [65] achieves high performance in packet classification for three types of rules, but it does not support rule update.

In aspect of comparing with BV decomposition based FPGA implementations, it can be noted that the throughput for IPC rule sets outperforms other designs while the performance for ACL and FW is also comparable to other designs. The StrideBV [19] does not support range match or update. Although real-time rule update is supported in [20], this design can only perform prefix match in the source address and destination address fields, and exact match in all the other fields, and [74] have range search capabilities and support dynamic rule update. It is noticeable that none of the BV decomposition based FPGA approaches could support large-scale rule sets.

Our implementation is also compared with TCAM-based packet classification methods [76], [77], which simulate TCAM by SRAM and are implemented on FPGA, in Table V. They both support TCAM word update and range matching, but the dynamic update performance is not provided in the literature. Note that only about 10k-scale entries of words can

TABLE V
COMPARISON WITH DECISION TREE BASED, DECOMPOSITION-BASED, AND TCAM-BASED APPROACHES ON FPGA

| Approaches | | Rule type | Rule set size | Device | Range support | Dynamic update | Classification throughput (MPPS) | Update throughput (MUPS) |
|---|---|---|---|---|---|---|---|---|
| Decision tree based | Proposed KickTree_Systolic | IPC | 10k | Ultrascale+ VU9P | ✓ | ✓ | 919.19 | 7.1 |
| | | ACL | | | | | 531.3 | 8.1 |
| | | FW | | | | | 623.2 | 9.2 |
| | TcbTree [67] | IPC | 10k | Ultrascale+ VU9P | ✓ | ✓ | 394.2 | 11.6 |
| | | ACL | | | | | 310.07 | 9.4 |
| | | FW | | | | | 508.5 | 11.4 |
| | KickTree_Parallel [73] | IPC | 10k | Ultrascale+ VU9P | ✓ | ✓ | 394.9 | 3.52 |
| | | ACL | | | | | 545.8 | 3.27 |
| | | FW | | | | | 509.4 | 3.78 |
| | MitTree on FPGA [65] | IPC | 10k | Virtex-7 XC7V690T | ✓ | × | 514.2 | \ |
| | | ACL | | | | | 496 | |
| | | FW | | | | | 524.4 | |
| | REC [64] | ACL | 10k | Virtex-5 XC5VFX200T | ✓ | × | 323.5 | \ |
| | | ACL | 10k | Virtex-6 XC6VLX760 | | | 388.2 | |
| | Modified Hypercuts [63] | ACL | 10k | Stratix III EP3SE260H780 | ✓ | × | 433 | \ |
| | D$^2$BS [27] | ACL | 10k | Virtex-5 XC5VSX240T | ✓ | × | 263.7 | \ |
| | Hypercuts on FPGA [26] | ACL | 10k | Virtex-5 XC5VFX200T | ✓ | ✓ | 250.7 | \ |
| | CubeCuts [62] | ACL | 10k | Virtex-5 XC5VFX200T | ✓ | × | 368.8 | \ |
| | Hypersplit on FPGA [22] | ACL | 10k | Virtex-6 XC6VLX760 | ✓ | × | 230.5 | \ |
| Decomposition based | Updatable Classifier1 [20] | 5-tuple | 1k | Virtex-6 XC6VLX760 | × | ✓ | ˜690 | 1 |
| | Many-field classifier [63] | 15-tuple | 1k | Virtex-7 XC7VX1140t | ✓ | × | ˜500 | \ |
| | Range-enhanced [28] | 12-tuple | 3k | Virtex-6 XC6VLX760 | ✓ | × | 566 | \ |
| | Updatable Classifier2 [74] | 5-tuple | 1k | Virtex-6 XC6VLX760 | ✓ | ✓ | ˜690 | 1 |
| | StrideBV [75] | 5-tuple | 0.5k | Virtex-6 XC6VLX760 | × | × | ˜390 | \ |
| TCAM based | TCAM on FPGA [76] | / | 16k | Virtex-7 XC7V2000T | ✓ | ✓ | 153 | \ |
| | Pseudo-TCAM [77] | / | 10k | UltraScale XCVU080 | ✓ | ✓ | 426 | \ |

be accommodated while almost exhausting all on-chip memory and logic slices, which is similar to BV decomposition based designs.

## VI. CONCLUSION

We propose an algorithm/hardware co-design multi-tree scheme specially designed for parallel packet classification on FPGA in this paper. First, an algorithm that constructs multiple shallow trees for the rule set with a recursive rule sifting process is designed, which can leverage the intrinsic parallelism of FPGA. Different from traditional space-cutting based multi-tree construction, our rule sifting mechanism breaks the space constraints of rule-to-tree mapping and enables bounded height on each tree, which can thus provide the potential of bounded worst-case and line-speed performance. Then, we design a flexible hardware architecture with multiple systolic arrays that can be implemented in parallel on FPGA. This architecture gets rid of the need of parallel parsing for multiple trees and can be agilely adapted to various rule sets. Each systolic array works as a coarse-grained pipeline, and the constructed multiple trees will be mapped onto these pipeline stages. This hardware-software mapping enables bounded worst-case rule searching. Additionally, incremental rule update can be achieved simply by traversing the pipeline in one pass, with little and bounded impact on rule searching. Extensive FPGA experimental results show that, our proposed scheme can achieve high performance on both search and update for large-scale rule sets.

## REFERENCES

[1] Y. Xin, Y. Liu, W. Li, R. Yao, Y. Xu, and Y. Wang, "KickTree: A recursive algorithmic scheme for packet classification with bounded worst-case performance," in *Proc. Symp. Archit. Netw. Commun. Syst.*, Dec. 2021, pp. 23–30.

[2] D. Firestone et al., "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. USENIX NSDI*, 2018, pp. 51–66.

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

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

[5] G. Varghese and J. Xu, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Mateo, CA, USA: Morgan Kaufmann, 2022.

[6] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "Scaling open vSwitch with a computational cache," in *Proc. USENIX NSDI*, 2022, pp. 1359–1374.

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

[8] K. Zheng, H. Che, Z. Wang, B. Liu, and X. Zhang, "DPPC-RE: TCAM-based distributed parallel packet classification with range encoding," *IEEE Trans. Comput.*, vol. 55, no. 8, pp. 947–961, Aug. 2006.

[9] K. Zhen, C. Hu, H. Lu, and B. Liu, "A TCAM-based distributed parallel IP lookup scheme and performance analysis," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 863–875, Aug. 2006.

[10] C. R. Meiners, A. X. Liu, and E. Torng, *Hardware Based Packet Classification for High Speed Internet Routers*. New York, NY, USA: Springer, 2010.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] Y. Sadeh, O. Rottenstreich, and H. Kaplan, "Optimal weighted load balancing in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 30, no. 3, pp. 985–998, Jun. 2022.

[18] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs," in *Proc. 21st Annu. Symp. Parallelism Algorithms Archit.*, Aug. 2009, pp. 188–196.

[19] T. Ganegedara, W. Jiang, and V. K. Prasanna, "A scalable and modular architecture for high-performance packet classification," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1135–1144, May 2014.

[20] 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.

[21] 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, p. 3068, Apr. 2020.

[22] 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.

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

[24] Y.-K. Chang, Y.-C. Lin, and C.-C. Su, "Dynamic multiway segment tree for IP lookups and the fast pipelined search engine," *IEEE Trans. Comput.*, vol. 59, no. 4, pp. 492–506, Apr. 2010.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. IEEE Hot Interconnects*, Aug. 1999, pp. 34–41.

[30] T. Y. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. IEEE INFOCOM*, Mar. 2000, pp. 1213–1222.

[31] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Aug. 2003, pp. 213–224.

[32] Y.-K. Chang, "Efficient multidimensional packet classification with fast updates," *IEEE Trans. Comput.*, vol. 58, no. 4, pp. 463–479, Apr. 2009.

[33] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 648–656.

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

[35] 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.

[36] Y.-C. Cheng and P.-C. Wang, "Packet classification using dynamically generated decision trees," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 582–586, Feb. 2015.

[37] 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.

[38] W. Li, X. Li, H. Li, and G. Xie, "CutSplit: A decision-tree combining cutting and splitting for scalable packet classification," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2645–2653.

[39] C.-L. Hsieh and N. Weng, "Many-field packet classification for software-defined networking switches," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Mar. 2016, pp. 13–24.

[40] J. Daly and E. Torng, "ByteCuts: Fast packet classification by interior bit extraction," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2654–2662.

[41] 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.

[42] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Oct. 1998, pp. 191–202.

[43] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Oct. 1998, pp. 203–214.

[44] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Aug. 1999, pp. 147–160.

[45] F. Baboescu and G. Varghese, "Scalable packet classification," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 199–210, 2001.

[46] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducing of field labels," in *Proc. IEEE INFOCOM*, Mar. 2005, pp. 269–280.

[47] 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.

[48] 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.

[49] 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, Jun. 2019.

[50] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM*, 1999, pp. 135–146.

[51] H. Song, J. Turner, and S. Dharmapurikar, "Packet classification using coarse-grained tuple spaces," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, Dec. 2006, pp. 41–50.

[52] H. Lim and S. Y. Kim, "Tuple pruning using Bloom filters for packet classification," *IEEE Micro*, vol. 30, no. 3, pp. 48–59, May 2010.

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

[54] T. Shen et al., "RVH: Range-vector hash for fast online packet classification," Techical Rep. ICT, 2018.

[55] 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.

[56] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "A computational approach to packet classification," *IEEE/ACM Trans. Netw.*, vol. 30, no. 3, pp. 1073–1087, Jun. 2022.

[57] 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.

[58] X. Zhang, G. Xie, X. Wang, P. Zhang, Y. Li, and K. Salamatian, "Fast online packet classification with convolutional neural network," *IEEE/ACM Trans. Netw.*, vol. 29, no. 6, pp. 2765–2778, Dec. 2021.

[59] C. Zhang, G. Xie, and X. Wang, "DynamicTuple: The dynamic adaptive tuple for high-performance packet classification," *Comput. Netw.*, vol. 202, Jan. 2022, Art. no. 108630.

[60] J. Zhong, Z. Wei, S. Zhao, and S. Chen, "TupleTree: A high-performance packet classification algorithm supporting fast rule-set updates," *IEEE/ACM Trans. Netw.*, vol. 31, no. 5, pp. 2027–2041, Oct. 2023.

[61] Y. Liu et al., "HybridTSS: A recursive scheme combining coarse- and fine-grained tuples for packet classification," in *Proc. ACM APNet*, 2022, pp. 1–7.

[62] Y.-K. Chang and Y.-H. Wang, "CubeCuts: A novel cutting scheme for packet classification," in *Proc. 26th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, Mar. 2012, pp. 274–279.

[63] 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.

[64] Y.-K. Chang, H.-C. Chen, and G. Parr, "Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA," *Comput. J.*, vol. 62, no. 2, pp. 198–214, Feb. 2019.

[65] J. Tan, G. Lv, Y. Ma, and G. Qiao, "High-performance pipeline architecture for packet classification accelerator in DPU," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2021, pp. 1–4.

[66] J. Tan, G. Lv, and G. Qiao, "MBitTree: A fast and scalable packet classification for software switches," in *Proc. IEEE Symp. High-Perform. Interconnects (HOTI)*, Aug. 2021, pp. 60–67.

[67] Y. Xin, W. Li, G. Tang, T. Yang, X. Hu, and Y. Wang, "FPGA-based updatable packet classification using TSS-combined bit-selecting tree," *IEEE/ACM Trans. Netw.*, vol. 30, no. 6, pp. 2760–2775, Dec. 2022.

[68] Y. Xin, W. Li, G. Xie, Y. Xu, and Y. Wang, "Updatable packet classification on FPGA with bounded worst-case performance," in *Proc. IEEE Symp. High-Perform. Interconnects (HOTI)*, Aug. 2022, pp. 21–28.

[69] 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.

[70] Y.-K. Chang, "A 2-level TCAM architecture for ranges," *IEEE Trans. Comput.*, vol. 55, no. 12, pp. 1614–1629, Dec. 2006.

[71] 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.

[72] J. Matoušek, A. Lučanskỳ, D. Janeček, J. Sabo, J. Kořenek, and G. Antichi, "ClassBench-ng: Benchmarking packet classification algorithms in the OpenFlow era," *IEEE/ACM Trans. Netw.*, vol. 30, no. 5, pp. 1912–1925, Oct. 2022.

[73] Y. Xin, W. Li, G. Xie, Y. Xu, and Y. Wang, "A parallel and updatable architecture for FPGA-based packet classification with large-scale rule sets," *IEEE Micro*, vol. 43, no. 2, pp. 110–119, Mar. 2023.

[74] Y. R. Qu, S. Zhou, and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA," in *Proc. Archit. Netw. Commun. Syst.*, Oct. 2013, pp. 125–136.

[75] 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.

[76] W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs," in *Proc. Archit. Netw. Commun. Syst.*, Oct. 2013, pp. 71–82.

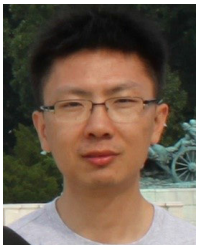
[77] W. Yu, S. Sivakumar, and D. Pao, "Pseudo-TCAM: SRAM-based architecture for packet classification in one memory access," *IEEE Netw. Lett.*, vol. 1, no. 2, pp. 89–92, Jun. 2019.

**Yao Xin** received 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 with the University of Southern California, USA, in 2014. He is currently an Associate Professor with the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangdong, China. His current research interests include network intelligent hardware acceleration, VLSI design for deep learning, and network algorithms.

**Wenjun Li** received the Ph.D. degree from Peking University in 2020. From 2020 to 2023, he was a Post-Doctoral Fellow with the Peng Cheng Laboratory and Harvard University, co-advised by Academician Yunjie Liu and Prof. Minlan Yu. From 2014 to 2015, he was a Research Engineer with Huawei Technologies Company Ltd. He is currently an Associate Researcher with the Network Research Department, Peng Cheng Laboratory. His current research interests include programmable network data planes, network telemetry, and network algorithms.

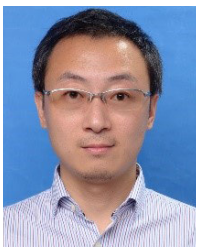
**Chengjun Jia** received the B.Eng. degree from Tsinghua University, Beijing, China, in 2018. He is currently pursuing the Ph.D. degree with the Network Security Laboratory, Department of Automation, Tsinghua University, China, under the supervision of Prof. Jun Li. He has published some articles on packet classification, congestion control, and network verification. His current research interests include parallel computer architecture, data center networks, and hardware acceleration.

**Xianfeng Li** (Member, IEEE) received the B.S. degree from the School of Computer and Control Engineering, Beijing Institute of Technology, in 1995, and the Ph.D. degree in computer science from the National University of Singapore in 2005. He is currently an Associate Professor with the Macau University of Science and Technology. Prior to that, he was an Associate Professor with the Peking University Shenzhen Graduate School. His current research interests include SDN, co-design of hardware and software, and the Internet of Things.



**Yang Xu** (Member, IEEE) received the B.Sc. degree from the Beijing University of Posts and Telecommunications in 2001 and the Ph.D. degree from Tsinghua University in 2007. He is currently the Yaoshihua Chair Professor with the School of Computer Science, Fudan University. Prior to joining Fudan University, he was a Faculty Member with the New York University Tandon School of Engineering. He has published more than 120 articles and holds more than ten U.S. and internationally granted patents on various aspects of networking and computing. His current research interests include SDN, DCN, NFV, and edge computing. He served as a TPC member for many international conferences, an Editor for *Journal of Network and Computer Applications*, and a Guest Editor for IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS Special Series on Network Softwarization and Enablers.

**Bin Liu** (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1993, respectively. He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His current research interests include high-performance switches/routers, network processors, and greening the internet. He has received numerous awards from China, including the Distinguished Young Scholar of China and won the Inaugural Applied Network Research Prize sponsored by ISOC and IRTF in 2011.



**Zhihong Tian** (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and technology from the Harbin Institute of Technology, Harbin, China, in 2001, 2003, and 2006, respectively. He is currently a Professor and the Dean with the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangdong, China. He is honored as a Pearl River Scholar in Guangdong Province. He is also a part-time Professor with Carlton University, Ottawa, Canada. Previously, he served in different academic and administrative positions with the Harbin Institute of Technology. He has authored over 200 journals and conference papers. His research has been supported in part by the National Natural Science Foundation of China, the National Key Research and Development Plan of China, and the National Hightech Research and Development Program of China (863 Program). He also served as a member, the chair, and the general chair of several international conferences. He is a Distinguished Member of the China Computer Federation.

**Weizhe Zhang** (Senior Member, IEEE) received the Ph.D. degree in computer science from the Harbin Institute of Technology, Harbin, China, in 2006. He is currently a Professor with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China, and the Director of the Department of New Networks, Peng Cheng Laboratory, Shenzhen. He has authored or coauthored more than 130 academic papers in journals, books, and conference proceedings. His current research interests include cyberspace security, cloud computing, and high-performance computing. He is a senior member of ACM.