Updatable Packet Classification on FPGA with Bounded Worst-Case Performance

Yao Xin[§] Wenjun Li[⋆§✉] Gaogang Xie[◇] Yang Xu[†§] Yi Wang[‡§△]
[§]Peng Cheng Laboratory [⋆]Harvard University [◇]CNIC of the Chinese Academy of Sciences [†]Fudan University
[‡]Institute of Future Networks in SUSTech [△]Heyuan Bay Area Digital Economy Technology Innovation Center
Email: xiny@pcl.ac.cn, wenjunli@seas.harvard.edu, xie@cnic.cn, xuy@fudan.edu.cn, wangy37@sustech.edu.cn

Abstract—**FPGA has been recognized as an attractive accelerator for line-speed packet classification in SmartNIC due to its ability to reconfigure and provide massive parallelism. As a promising algorithmic approach that can fully exploit the FPGA characteristics, decision tree based packet classification on FPGA has been actively investigated in the past decade. However, most of them suffer from unbalanced tree structures with unpredictable depths under certain rule sets, so the potential of FPGA may not be brought into full play. Worse still, few of them can support efficient rule updates on-the-fly, which is highly required in virtualized data centers. To address these issues, we design and implement an efficient hardware architecture based on the recently proposed KickTree algorithm, which consists of multiple balanced trees with bounded depth. A strategy of multi-PE (processing element), parallel search, and serial update is adopted to decouple the search and update process. The parsing of multiple tree search results adopts a modular and hierarchical design, supporting architecture with various tree numbers. Additionally, incremental rule updates can be achieved simply by traversing all PEs in one pass, with little and bounded impact on rule searching. Experimental results on FPGA show that our design can achieve an average classification throughput of 182.6 MPPS and an average update throughput of 3.1 MUPS for various 100k-scale rule sets.**

Index Terms—**FPGA, Packet Classification, Decision Tree**

1. Introduction

With the diversification of Internet applications, data centers have become efficient and promising infrastructures for deploying diverse network services and applications (e.g., video streaming, cloud computing). However, these applications and services typically place multiple resource demands on the underlying infrastructure, which traditional data centers cannot meet. In such circumstances, services based on the softwarization and virtualization paradigms, such as Network Functions Virtualization (NFV) and Software Defined Networking (SDN), have emerged as a solution for data centers that can provide flexibility by implementing a software-based network over physical infrastructure [1], [2].

Virtualization has enabled tens to hundreds of virtual machines (VMs) per server in data centers using multi-core CPU technology. However, the network functions of the virtual network protocol stack in data centers are updated frequently and increased all along. As a result, packet processing functions, such as packet classification, routing decisions, encryption/decryption, etc., have increased exponentially. Worse still, x86 servers not optimized for packet processing are inefficient to implement in software. To address this issue, SmartNIC based on specific hardware platforms such as Network Processor (NP) and Field Programmable Gate Array (FPGA) has been increasingly adopted to offload these functions. Among these, packet classification is a fundamental and essential task, which is to discriminate packets into separate "flows" and enables differentiated functionalities [3], so that all packets belonging to the same flow will be processed similarly.

Among various hardware platforms, FPGA has been regarded as promising hardware to realize packet processing in data center scenarios, thanks to its flexible programmability, high performance, and rich capacity [4], [5], [6]. The majority of FPGA-related work is based on bit-vector (BV) decomposition and decision trees. However, almost none of these satisfy the demands of the increasing scale of rules and frequent rule updates brought by NFV and SDN. Specifically, the BV decomposition method requests vast distributed RAMs for applied vectors [7], [8], [9], [10]. Thus, only small-scale rule sets can be supported by architectures of this type due to the limited hardware resources, although rule update is well supported. On the other hand, although decision tree based approaches are capable of performing large-scale rule lookups, dynamic rule update remains a challenge due to: i) the rule replication problem, as it is difficult to ensure atomicity and consistency in the update process; and ii) the overly scattered storage in pipelined architectures [11], [12], [13], [14], [15], [16], [17], [18].

To address the above issues, we propose an updatable FPGA-based packet classifier to meet the requirements of network virtualization. The hardware architecture is developed based on our recently proposed algorithmic packet classification scheme called KickTree [19], a multi-tree algorithm without any rule replications. For high lookup throughput, we adopt multiple processing elements (PEs) running in parallel to perform rule search, and multiple computing cores of the classifier enter into force at the top layer. In order to support incremental rule updates, we adopt a strategy of centralized memory and serial access in each PE rather than an entire pipeline. With respect to the organization of multiple PEs in each classifier, a method of parallel search and serial update is proposed to decouple the search and update process. The experimental results after implementation on a Xilinx Ultrascale+ FPGA show that, for various 100k-scale rule sets generated by ClassBench [20], it can achieve an average classification throughput of 182.6 MPPS and an update throughput of 3.1 MUPS. The major contributions of this paper are as follows:

- The search result resolution for multiple PEs with unpredictable numbers is designed in a modularized manner through the hierarchical composition of the two-input result resolvers as the fundamental unit. In this method, rule sets with any number of trees can be supported.

TABLE 1: Example rule set with four IPv4 header fields

rule id	priority	src_addr (SA)	dst_addr (DA)	src_port (SP)	dst_port (DP)	*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*

- To avoid repeating updates in multiple PEs, the update process is decoupled from the search one and is performed in a serial fashion between PEs.
- In order to prevent all tree PEs from failing to update, an update guarantee scheme is designed, which is supplementing with a linear search based PE as the final layer without compromising the overall performance.

The rest of the paper is organized as follows. Section 2 summarizes background and related work briefly. Section 3 provides a summation of KickTree algorithm. Section 4 illustrates the hardware architecture. Section 5 shows experimental results. Finally, Section 6 draws the conclusion.

2. Background and Related Work

2.1. 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, together with an action to be taken in case of a match. r_i is a regular expression on the i field of the packet header, which could be a prefix, a range, or an exact value. A packet $p = (p_1, p_2, ..., p_d)$ is said to match rule r if $\forall i, p_i \in r_i$. Each rule is associated with a priority, indicating the degree of importance. If a packet conforms to more than one rule, the low priority rules will give way to the highest priority rule. Table 1 shows an example rule set with four IPv4 header fields. As a widely studied bottleneck, packet classification has attracted extensive research attention, and many algorithmic approaches have been proposed in the past two decades [21], such as decision tree [15], [17], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], decomposition [36], [37], [38], and Tuple Space Search (TSS) [39], [40], [41], [42].

Despite more than twenty years of research, the software packet classification algorithm still has a severe performance bottleneck, which makes it challenging to meet the ever-increasing line-speed forwarding requirements. As a result, hardware using Ternary Content Addressable Memory (TCAM) has been the dominant implementation of packet classification in the industry. However, TCAM has low area efficiency, high energy consumption, and high cost [43], [44], [45], [46], [47], [48], severely limiting its scalability in SmartNIC. Under such circumstances, FPGA has been recognized as an attractive accelerator for algorithmic packet classification in SmartNIC due to its flexible programmability, high performance, and productive capacity. Next, we will review some related FPGA designs for packet classification.

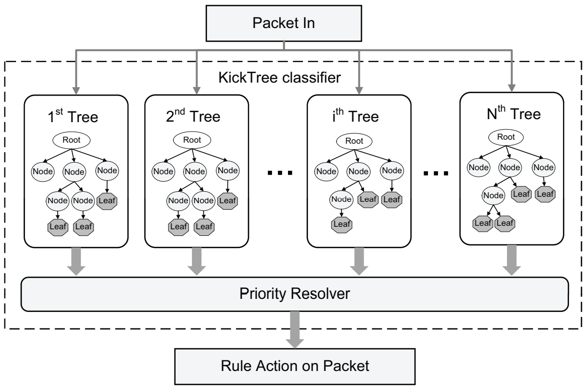

Figure 1: The algorithmic framework of KickTree [19].

2.2. FPGA-based Packet Classification Solutions

According to the different algorithm techniques, the current FPGA-based algorithmic packet classification can generally be divided into two categories: decision tree based and decomposition-based designs. Therefore, we will focus on these two approaches to review the related work on FPGA.

Decision Tree based Designs on FPGA: The working scheme of decision tree based methods involves partitioning the search space recursively into several smaller sub-regions based on information from one or more fields in the rules until each sub-region contains the number of rules below a certain threshold (i.e., *binth*). The vast majority of decision tree based FPGA architectures have adopted a fully pipelined design, which can benefit from the high frequency and high throughput. The primary concerns of such a method are memory reduction and performance enhancement [11], [15], [49], [50], [51], [52]. However, despite various techniques to minimize rule replication, this problem persists, leading to inefficient FPGA memory consumption, and making dynamic rule updates (i.e., without precomputing the memory content or rebuilding trees) difficult. Rule updates can only be achieved by calculating what storage content to change ahead of time and then sequentially issuing write bubbles to the pipeline stages. Furthermore, most decision tree algorithms are intended for software without considering hardware characteristics. Thus, the migration and mapping process from software to hardware would sacrifice some intrinsic advantages.

Decomposition-based Designs on FPGA: The principle of the decomposition-based method is to decompose a complex multi-domain search problem into multiple simple single-domain concurrent searches, which can make full use of the parallel characteristics of FPGA. Most of the current decomposition implementations on FPGA are based on the well-known BV algorithm [36], which splits each field into multiple subfields and can be searched in a pipelined manner in all subfields [8], [9], [10], [17], [53]. The BV-based design can sustain high throughput for packet classification and fully support dynamic rule updates. However, this method essentially lists all possible matching combinations of bits in rules exhaustively. Therefore, rules with wildcards consume more hardware, especially logical resources. This feature always constrains the scale of rule sets accommodated by FPGAs.

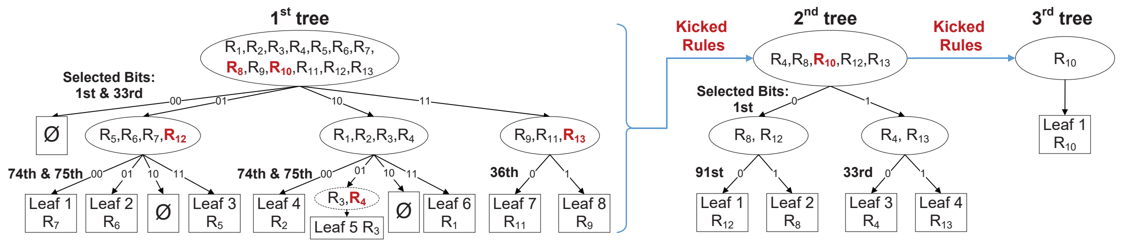

Figure 2: A working example of KickTree [19], with *Max_Depth* = 2, *Max_Selecting_Bits* = 2, and *binth* = 1.

As seen from the above description, although FPGA has been increasingly recognized as a promising platform for algorithmic packet classification in the last decade, existing FPGA-based packet classifiers cannot satisfy the burgeoning needs proposed by virtualized data centers, such as large-scale rule sets and frequent rule updates.

3. Algorithm Overview

Since our aim in this work is to present a novel FPGA hardware design based on our recently proposed algorithmic KickTree [19], which is a multi-tree algorithm dedicated to FPGA, we will first give some technical review of the KickTree algorithm in this section from the following two aspects:

Why is FPGA Hungry for Decision Trees from Scratch? First, the previously designed FPGA solutions based on decision trees could hardly support dynamic rule updates due to: i) the rule replication problem that makes it very hard to ensure atomicity and consistency during the update; and ii) the overly scattered storage in fully pipelined architectures. Second, the vast majority of existing FPGA solutions are based on off-the-shelf algorithms that are not tailored to hardware features. Furthermore, the migration and mapping process from software to hardware will sacrifice some inherent advantages due to the distinctions in characteristics between software and hardware, resulting in the potential of FPGA not being fully realized. In response to the above problems, we redesigned a hardware-targeted algorithm KickTree, which builds a few worst-case bounded trees without any rule replications and takes appropriate hardware characteristics into account. The algorithmic framework of KickTree is shown in Figure 1.

How to Build Hardware-friendly Trees in KickTree? KickTree first converts each range field into a *Longest Common Prefix* [54], [55], [56], allowing each rule to be represented by a sequence of ternary strings (i.e., 0, 1, wildcard). After that, several balanced trees of bounded depth are constructed in a recursive manner, consisting of two key steps: 1) Tree building by bit-selecting. Non-wildcard bits are dynamically extracted from all possible header fields using a locally greedy strategy for shallow and balanced tree purposes; 2) Rule sifting. Rules that do not meet the bit selection conditions (e.g., the rule value for the selected bit position is a wildcard) or rules that exceed *binth* in leaf nodes are "kicked out" from the current tree. If there are still rules left after building the current tree, the same method would be utilized to construct the decision tree continuously, and the kicked-out rules would be retained for building the next tree. This process continues until there are no more rules. Figure 2 illustrates a KickTree classifier construction example for the rules given in Table 1.

4. Hardware Design

4.1. Design Overview

While the KickTree algorithm appears to be a good fit for FPGA, there are still many challenges in concrete FPGA implementation because the most widely studied pure-pipeline designs can hardly support dynamic rule updates without precomputing memory content. In addition, as a multi-tree scheme, how to efficiently collect and parse the results of multiple trees and how to cope with the relationship between classification and update remains challenging. Therefore, to fully support dynamic rule updates without sacrificing lookup performance for large-scale rule sets (e.g., 100k), we prefer to design a new hardware architecture from scratch rather than use a classic pure pipeline design, so that the potential of the KickTree algorithm can be fully unleashed. Next, we will introduce the top-level architecture and storage organization dedicated to KickTree. After that, the detailed architectural design of each search tree PE is elaborated, followed by a hierarchical concurrent result collection scheme. Finally, we present the rule update mechanism.

4.2. Top-level Architecture

The top-level architecture of each classifier adopts a parallel-search, serial-update strategy, as illustrated on the right side of Figure 3. Each processing element (PE) corresponds to a tree in Figure 1. Each PE processes rule search and update separately. As a result, there are two interfaces for receiving commands for rule search and update and two interfaces for exporting search and update results, respectively. The input command is made up of a packet/rule along with an operation code of SEARCH (for packet), or DELETE/INSERT (for rule), while the result consists of the matched rule ID and result code of RULE_FOUND, RULE_NOT_FOUND, UPDATE_SUCCESS, UPDATE_FAILURE, etc.

The top-level classifier handles the two input commands differently: the search command is delivered to all PEs for parallel execution, whereas the update command is only distributed to the first PE and executed serially in subsequent PEs. Up until the last resolver provides the final result, the results of all PE searches are parsed and merged in pairs by the result resolver level by level.

The update results, however, go through each PE in turn. The rule update operation is continued in the current PE if the update in the preceding PEs fails, and the result is passed until the update is successful in a particular PE. From a high-level perspective, each classifier can operate independently. Thus, as long as hardware resources allow, multiple computing cores can enter into force on the FPGA to improve the overall performance.

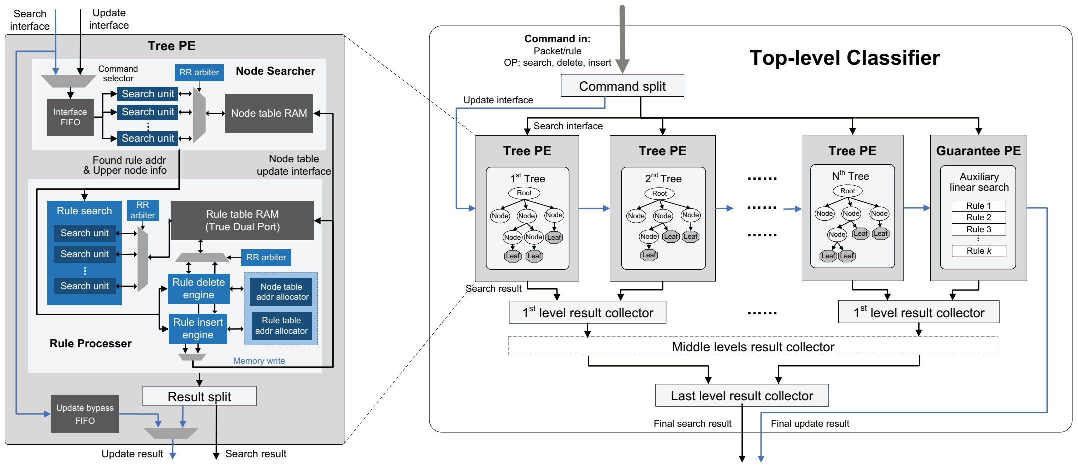

Figure 3: The architecture of the classifier.

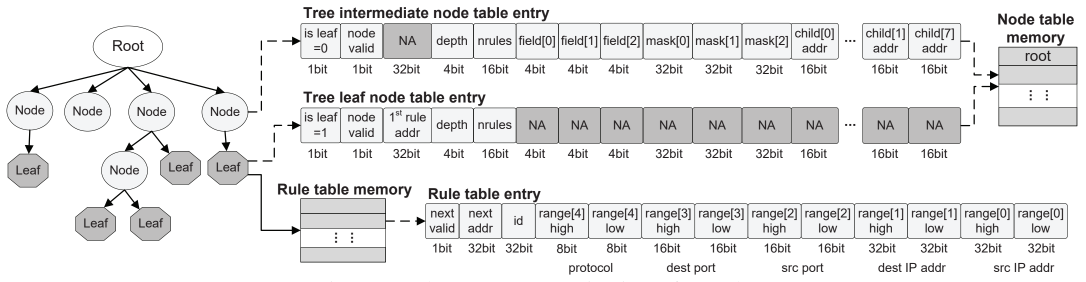

Figure 4: The storage organization of search tree.

4.3. Search Tree Storage Organization

As shown in Figure 4, the storage organization of each search tree is cached in two centralized memories: node table RAM and rule table RAM. The entries of these memories constitute three types of unidirectional chained lists. The intermediate node table and leaf node entries represent intermediate nodes (including the root node) and leaf nodes, respectively. And each rule table entry represents a specific rule. Each leaf node is associated with a rule subset, and the collection of subsets of all leaf nodes is cached in the rule table RAM.

Node Table Entry: The node tables in Figure 4 show the case of the selectable bit number being 3, then each node has a maximum of 8 child nodes. The first bit, *is_leaf*, indicates if the current node type is intermediate or leaf. The *node_valid* bit indicates if the current node is valid (i.e., 0 means there is no rule associated with this node). Unlike the leaf node entry, the intermediate node table has bits $field[i]$ $(i = 1, 2, 3)$ for selecting rule dimensions, and One-Hot Encoding $mask[i]$ $(i = 1, 2, 3)$ for bit selection. The *child_addr* records the next-level child node address determined by the selected 3 bits. When a leaf node table is reached, $1^{st}\_rule\_addr$ is referenced to locate the rule table RAM address of the first rule in the relevant subset.

Rule Table Entry: Each rule table entry corresponds to a rule in the leaf node. This table can be scaled to different format rules, however it only illustrates the situation of the 5-tuple format here. The bit *next_valid* is used to link rules within a leaf node by indicating whether the next rule is valid or not. Each mask is transferred to ranges with two endpoints in advance and documented in this entry.

4.4. Search Tree PE

Instead of a fully-pipelined design, the architecture of the search tree PE adopts a centralized memory and serial access method, as illustrated on the left side of Figure 3. The Node Searcher traverses the tree nodes level by level from the root by reading linked node table entries from memory. When a valid leaf node is found, the rule subset address will be delivered to the Rule Processor, which will linearly search the rule table RAM and take the appropriate actions of search, delete, or insert, according to the operation code. To enhance memory utilization, we implement multiple search units to work in parallel, since they can access the memory in different time slots with limited query time. Furthermore, Round Robin ensures the arbitration between multiple units in both modules as each unit has the same priority.

PEs are designed to process rules and packets similarly by sharing hardware resources. In particular, the Node Searcher processes rules using the lower endpoint of the range identical to the packet format as input. At the same time, the Rule Processor handles the specific operations of packets (search) and rules (delete/insert) with different modules. In addition, each PE implements an update bypass FIFO which caches successful update results and forwards them, bypassing the tree. The Node Searcher, in contrast, processes only unsuccessful update results in preceding PEs. This mechanism avoids repeated updates of multiple PEs and reduces the delay of serial updates.

4.5. Hierarchical Concurrent Results Collection

For the purpose of rapid development and hardware implementation of a classifier with arbitrary tree numbers,

TABLE 2: 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_100k	10	38934	86055	3185	9684	638	3260	13	15	17	12	14	10	12	Block	Block	Ultra	Ultra	Ultra	Dist	Block
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	9	35228	82351	6930	15561	394	1230	12	15	17	13	14	9	12	Ultra	Ultra	Ultra	Block	Block	Dist	Block
acl4_100k	10	35556	82970	3811	10776	946	3206	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	12	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	13	18725	63115	5195	15224	682	1998	12	14	16	13	15	9	12	Block	Block	Ultra	Ultra	Ultra	Block	Block
fw4_100k	16	18925	57688	3411	7953	2037	3868	12	14	16	12	14	11	12	Block	Ultra	Ultra	Block	Ultra	Block	Block
fw5_100k	14	37448	54157	9104	25692	471	2578	13	15	16	14	15	9	12	Block	Ultra	Ultra	Block	Ultra	Block	Block

we introduce the idea of modularity in the top-level architecture design. Since the number of trees can be random and unpredictable for any given set of rules, search result collection and priority parsing for multiple trees would be a challenge. We employ a multi-level result resolver with a 2-input result resolver as the fundamental component to overcome this problem. Accordingly, the results of the search trees are parsed hierarchically in pairs. Empty trees are used to supplement the number of search trees to powers of 2. For example, the number of resolver levels for a classifier with N trees $(N > 1)$ would be $log_2(N-1)+1$.

The architecture of the elementary 2-input result resolver is shown in Figure 5. It aims to solve two issues: 1) The classifier may produce out-of-order search results due to the existence of multiple search units and the different processing delays of each packet; 2) The speed at which discrete trees produce results is inconsistent. According to the first issue, each input channel has an independent RAM to reorder the out-of-order results, and the low-order bits of the packet ID act as the write address. Once a predetermined number of results have been written, the results of both channel are concurrently read in order and compared by priority. Rules with higher priority are output to the FIFO controlled by the bus interface to the next level module. The second issue is tackled by a dataflow balancer, which dynamically monitors the amount of data flowing into the two channels and controls the bus interface in real-time so that the input speed of the results on both sides is roughly equal.

For the purpose of resource-saving, the second channel can be set to bypass mode, which is activated when the second channel is connected to an empty tree. In bypass mode, the reorder RAM will not be implemented. In addition, the input bus handshake signal of the second channel is taken over by the first channel, and the first RAM also replaces the results that the reorder RAM should read out.

4.6. Rule Update Mechanism

Our hardware architecture supports real-time incremental rule updates (e.g., deletion or insertion) without precomputing memory contents as in pipelined designs. When performing rule updates in each PE, the Node Searcher caches the complete information of up to two levels of traversed nodes. Therefore, we can trace back two levels of nodes to modify the contents of related tables.

From the top-level perspective of the classifier, update commands and results pass sequentially through each PE. If the result of the previous PE has been successful, no

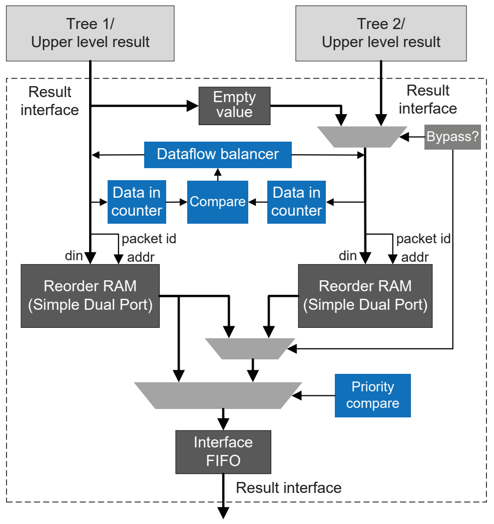

Figure 5: Modularized result resolver.

command will be passed to the tree. Instead, it will be cached directly in the bypass FIFO along with the result and continue to pass down until they are read from the last PE. The packet classification procedure must be interrupted prior to the update operation. Moreover, it is also preferable to wait until the update command has traversed all PEs before sending the subsequent one for the sake of maintaining atomic consistency.

Due to the hardware's difficulty in achieving the exact dynamic tree reconstruction as in software, there may be an update failure: a rule cannot be inserted into any tree. To prevent this from happening, the last PE of each classifier employs a linear search without the restriction of *binth* to accommodate previously inserted failing rules. Figure 3 displays this feature. Nevertheless, this is merely a guarantee mechanism. In practice, it is rare for all previous PE updates to fail, especially when the number of PEs is large.

In our experimental evaluation, we generate an average of approximately 10 PEs for 100K rule sets, and the first three PEs can handle practically all update activities. However, it should be noted 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 or follow the above algorithm to construct the tree recursively for the rules in the last PE, to avoid this auxiliary update module from turning into the throughput bottleneck of the system.

TABLE 3: Resource utilization for different rule sets

Rule set	Core num	CLB LUTs (1182240)	CLB Registers (2364480)	BRAM (2160)	URAM (960)	LUTRAM (591840)	Max frequency (MHz)
acl1_100k	6	607596	819039	1815	762	125232	200.08
acl2_100k	6	944525	1263251	2010	738	35574	200.12
acl3_100k	6	589951	802639	1494	846	108984	200.00
acl4_100k	6	573805	741532	1758	816	130818	200.36
acl5_100k	6	326950	275717	1635	576	173262	193.46
ipc1_100k	6	333575	472205	1593	762	46608	199.88
ipc2_100k	7	365928	415700	1365	684	142632	200.56
fw1_100k	6	835036	974454	1716	936	248268	192.83
fw2_100k	7	330732	465005	1519	812	47190	200.20
fw3_100k	6	896606	1049058	1170	564	271266	200.68
fw4_100k	6	84444	1243527	1536	600	82080	201.01
fw5_100k	6	799875	1101059	1524	744	13320	200.76

5. FPGA Implementation Result

5.1. Experiment Setup

In the experimental evaluation, we focus on the performance of large-scale rule sets as they are increasingly in demand in emerging data centers. ClassBench [20] is utilized to generate three types of rule sets according to the default parameters, namely ACL, FW, and IPC, all of which are 100k in size. Specifically, 12 rule sets are generated based on 12 seed parameter files (i.e., 5 ACLs, 5 FWs, and 2 IPCs). The design is developed by Vivado 2021.2 tool and evaluated on a Xilinx Virtex UltraScale+ VU9P FPGA equipped with extensive Ultra RAMs. Multiple computing cores can be instantiated by taking advantage of this property to achieve high performance.

In the following evaluations, the number of selectable bits, maximum tree depth, and *binth* (the threshold for the number of leaf node rules) are set to 3, 8, and 10, respectively. The numbers of search units in Node Searcher and Rule Processor in each search tree PE are set to 5 and 6 separately. Additionally, we further reduce the number of trees by increasing *binth* and the maximum depth settings for trees generated later. This adjustment is based on an observation: the first two trees concentrate most of the rules.

5.2. Storage Configuration

In most cases, the first PE has the highest storage requirements because it contains the majority of the rules. On the other hand, the memory depth of an FPGA is a power of 2. Therefore, to prevent excessive memory waste, we divide the note table RAM and rule table RAM in the first PE into two parts: upper_half and lower_half. In such a manner, instead of reserving $2^{(n+1)}$ storage depth when $2^n + 1$ entries are required, the actually implemented depth can be $2^n + 1 + Num_{spare}$, where the upper_half stores entries with addresses greater than 2^n while the lower_half has a depth of 2^n, and Num_{spare} is the entry number of the spare space allocated for rule update.

Because of the distinct characteristics of each rule set, the resulting storage organizations are different. To achieve the optimal performance for a specific rule set, hardware configurations for various rule sets have been customized and finely tuned, which are listed in Table 2. It mainly shows the storage configuration of the first three trees, which contain the vast majority of rules. RAM types include Distributed (Dist for short), Block, and Ultra. The u_node and l_node denote upper_half RAM and lower_half RAM, respectively. Because our hardware architecture supports real-time rule updates, sufficient spare storage space should be allocated for the search tree in each PE at the outset in case there are more insertions than deletions.

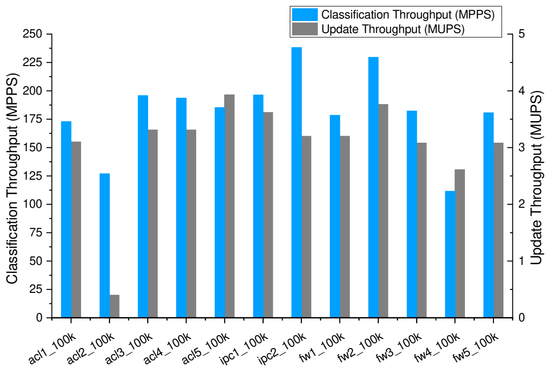

Figure 6: Classification and update throughput.

5.3. Resource Utilization

Table 3 summarizes the number of accommodated computing cores, resource usage, and maximum frequencies of hardware implementations of various rule sets after they are synthesized, placed, and routed. It is natural to note that memory, including Ultra RAM and Block RAM, is the most consumed FPGA resource. The minimum depths for Block RAM and Ultra RAM are 512 and 4096, respectively. Therefore, in the actual FPGA implementation, the Block and Ultra RAM allocated by many small data structures cannot be further split, resulting in the actual consumption of on-chip resources in Table 3 much larger than the minimum required capacity.

5.4. Throughput Evaluation

In this section, we evaluate the performance of our FPGA implementation in terms of throughput, including packet classification throughput and rule update throughput, measured in MPPS (Million Packets Per Second) and MUPS (Million Updates Per Second) as a unit. We calculate both types of throughput by simulation. Explicitly speaking, we first generate storage organization files consisting of node tables and rule tables for a specific rule set. We then simulate our architecture using these files at the maximum frequency obtained in Section 5.3 to perform classification/updates using packets/rules from trace/rules files. Classification throughput is the average of processing all synthetic packet traces, while update throughput is obtained by running randomly generated operations (including deletes, inserts, and modifications) for a long time (e.g., 100 ms) and calculating the average.

Figure 6 shows classification throughput and update throughput with respect to the benchmark rule sets. Performance varies according to different rule types. The classification throughput is distributed between 102.3 MPPS and 238.3 MPPS, with an average of 182.6 MPPS. Among them, the classification throughput corresponding to the rule sets of acl2_100k and fw4_100k lags far behind the performance of other rule sets, mainly due to the bottleneck effect caused by the excessive number of trees they generate. On the other hand, with the exception of the acl2_100k rule set, the range of update throughput does not fluctuate much, with an average of 3.1 MUPS.

TABLE 4: Latency for different rule sets

Rule set	Avg latency per packet (ns)	Min latency (ns)	Max latency (ns)
acl1_100k	477.6	360	540
acl2_100k	648.5	450	825
acl3_100k	452.0	345	540
acl4_100k	477.0	374	554
acl5_100k	355.0	243	481
ipc1_100k	371.1	315	435
ipc2_100k	375.3	254	434
fw1_100k	517.1	358	638
fw2_100k	394.9	285	480
fw3_100k	480.6	329	658
fw4_100k	673.5	417	745
fw5_100k	483.1	359	613

5.5. Latency Evaluation

We have calculated different latency values per packet for benchmark rule sets, including average latency, worst-case and best-case latencies, and Table 4 lists the results. Latency is closely related to several factors, such as the number of trees, the depth of the tree, the number of leaf node rules, memory read latency, etc. Although our latency is not very advantageous compared to some pure pipeline designs with only a few stages, it is a trade-off to support dynamic rule updates. Nevertheless, we aim to reduce latency in our future work.

5.6. Comparison with Related Work

Among multiple FPGA-based decision tree schemes, only [57] implements the evaluation of 100k rule sets. Two pipelines are implemented by utilizing dual-port RAM, and its throughput of packet classification reaches 352 MPPS, 418 MPPS, and 390 MPPS for ACL, FW, and IPC rules, respectively. Although the classification performance is higher than our implementation, it does not support rule updates, a common drawback of most decision tree based methods. Although [11] is claimed to be able to support on-the-fly rule update, the details of leaf node deletion/creation and intermediate node update are not discussed, and the corresponding hardware implementation is not proposed. The work in [15] proposes the method of inserting *write bubbles* to pipeline memories to enable rule updates. However, the new contents of the memory are computed offline rather than changing dynamically according to the on-the-fly update orders as in our proposed method. As far as we know, only our design has realized dynamic rule updates in implementation without the need to precompute the updated contents of the memory, compared with the previous decision tree based FPGA designs.

On the other hand, most decomposition-based FPGA implementations only support rule set scales of no more than 5k [9], [17], as this kind of method requires a large amount of on-chip memory to implement bit vectors, although they can achieve high performance in classifying packets. As a result, these works are not comparable to our design regarding the rule set scale. Furthermore, some designs do not support range matching [8], [9]. The work in [58] has range searching capabilities and supports dynamic rule update updates but is still limited by the rule set size. In contrast, our implementation can not only perform range matching but also support dynamic rule updates.

6. Conclusion

The trend toward network virtualization in the data center leads to widespread concern about the FPGA-based SmartNIC due to its ability to reconfigure and provide massive parallelism to offload fungible functions. Among the offloaded functions by SmartNIC, the classification and forwarding of network data packets are fundamental and essential tasks. Although decision tree based packet classification on FPGAs has been extensively researched for a decade, most of them are not only unbalanced but also have unpredictable depth, so the potential of FPGA cannot be brought into full play. Furthermore, due to rule replication and a full pipeline, they also face the challenge of dynamic update, which is highly desired in virtualized data centers. In this paper, we design and implement an efficient hardware architecture based on the recently proposed KickTree algorithm, consisting of multiple balanced trees of bounded depth, capable of entirely using the FPGA's advantage. Generally speaking, instead of a fully pipelined design, we adopt an architecture of multi-PE, multiple computing cores of the classifier on the uppermost level. We propose a parallel-search and serial-update strategy for PEs in each classifier to decouple the search and update process. The parsing of multiple tree search results adopts a modular and hierarchical design, supporting architectures with an almost arbitrary number of trees. Furthermore, we design a guarantee mechanism to guarantee the success of the update. Experimental results show that it can achieve an average throughput of 182.6 MPPS for classification and an average throughput of 3.1 MUPS for the update for various 100k-scale rule sets.

Acknowledgment

This work is supported by National Key R&D Program of China (2020YFB1806400), Key-Area R&D Program of Guangdong Province (2020B0101130003, 2021B0101400001), 173 Program (2021-JCJQ-JJ-0483), NSFC (61725206, 62102203, 62172108, 62150610497), China Postdoctoral Science Foundation (2020TQ0158, 2020M682825, PC2021037), Guangdong Basic and Applied Basic Research Foundation (2019B1515120031), and the Major Key Project of PCL (PCL2021A02, PCL2021A08, PCL2021A15). Wenjun Li is the corresponding author.

References

[1] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[2] J. Fan, M. Jiang, O. Rottenstreich, Y. Zhao, T. Guan, R. Ramesh, S. Das, and C. Qiao, "A framework for provisioning availability of NFV in data center networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2246–2259, 2018.

[3] H. J. Chao and B. Liu, "High performance switches and routers," in *John Wiley & Sons*, 2005.

[4] D. Firestone and et al, "Azure accelerated networking: SmartNICs in the public cloud," in *USENIX NSDI*, 2018.

[5] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4-NetFPGA workflow for line-rate packet processing," in *ACM/SIGDA FPGA*, 2019.

[6] S. Pontarelli and et al, "Flowblaze: Stateful packet processing in hardware," in *USENIX NSDI*, 2019.

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

[8] T. Ganegedara, W. Jiang, and V. K. Prasanna, "A scalable and modular architecture for high-performance packet classification," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1135–1144, 2014.

[9] Y. R. Qu and V. K. Prasanna, "High-performance and dynamically updatable packet classification engine on FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 197–209, 2015.

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

[11] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna, "Multi-dimensional packet classification on FPGA: 100 Gbps and beyond," in *IEEE FPT*, 2010, pp. 241–248.

[12] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *ACM/SIGDA FPGA*, 2009.

[13] Y.-K. Chang, Y.-C. Lin, and C.-C. Su, "Dynamic multiway segment tree for IP lookups and the fast pipelined search engine," *IEEE Transactions on Computers*, vol. 59, no. 4, pp. 492–506, 2009.

[14] W. Jiang and V. K. Prasanna, "A FPGA-based parallel architecture for scalable high-speed packet classification," in *IEEE ASAP*, 2009.

[15] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 20, no. 9, pp. 1668–1680, 2012.

[16] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna, "Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU," in *ACM/IEEE ANCS*, 2015.

[17] Y.-K. Chang and C.-S. Hsueh, "Range-enhanced packet classification design on FPGA," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 2, pp. 214–224, 2015.

[18] J. Tan, G. Lv, and G. Qiao, "MBitTree: A fast and scalable packet classification for software switches," in *IEEE HOTI*, 2021.

[19] 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 *ACM/IEEE ANCS*, 2021.

[20] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499–511, 2007.

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

[22] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *IEEE HOTI*, 1999.

[23] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *ACM SIGCOMM*, 2003.

[24] T. Y. Woo, "A modular approach to packet classification: Algorithms and results," in *IEEE INFOCOM*, 2000.

[25] Y.-K. Chang, "Efficient multidimensional packet classification with fast updates," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 463–479, 2008.

[26] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *IEEE INFOCOM*, 2009.

[27] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *ACM SIGCOMM*, 2010.

[28] W. Li and X. Li, "HybridCuts: A scheme combining decomposition and cutting for packet classification," in *IEEE HOTI*, 2013.

[29] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables," in *IEEE INFOCOM*, 2013.

[30] Y.-C. Cheng and P.-C. Wang, "Packet classification using dynamically generated decision trees," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 582–586, 2013.

[31] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng, "A sorted partitioning approach to high-speed and fast-update OpenFlow classification," in *IEEE ICNP*, 2016.

[32] W. Li, X. Li, H. Li, and G. Xie, "CutSplit: A decision-tree combining cutting and splitting for scalable packet classification," in *IEEE INFOCOM*, 2018.

[33] W. Li, T. Yang, Y.-K. Chang, T. Li, and H. Li, "TabTree: A TSS-assisted bit-selecting tree scheme for packet classification with balanced rule mapping," in *ACM/IEEE ANCS*, 2019.

[34] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *ACM SIGCOMM*, 2019.

[35] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "A computational approach to packet classification," in *ACM SIGCOMM*, 2020.

[36] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *ACM SIGCOMM*, 1998.

[37] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *ACM SIGCOMM*, 1999.

[38] W. Li, D. Li, Y. Bai, W. Le, and H. Li, "Memory-efficient recursive scheme for multi-field packet classification," *IET Communications*, vol. 13, no. 9, pp. 1319–1325, 2019.

[39] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *ACM SIGCOMM*, 1999.

[40] B. Pfaff and et al, "The design and implementation of open vswitch," in *USENIX NSDI*, 2015.

[41] J. Daly, V. Bruschi, L. Linguaglossa, S. Pontarelli, D. Rossi, J. Tollet, E. Torng, and A. Yourtchenko, "TupleMerge: Fast software packet processing for online packet classification," *IEEE/ACM Transactions on Networking*, 2019.

[42] W. Li and et al, "Tuple space assisted packet classification with high performance on both search and update," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, pp. 1555–1569, 2020.

[43] H. Liu, "Efficient mapping of range classifier into Ternary-CAM," in *IEEE HOTI*, 2002.

[44] B. Vamanan and T. Vijaykumar, "TreeCAM: decoupling updates and lookups in packet classification," in *ACM CoNEXT*, 2011.

[45] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1127–1140, 2012.

[46] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *ACM SIGCOMM*, 2014.

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

[48] R. Yao, C. Luo, X. Liu, Y. Wan, B. Liu, W. Li, and Y. Xu, "MagicTCAM: A multiple-TCAM scheme for fast TCAM update," in *IEEE ICNP*, 2021.

[49] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "ParaSplit: A scalable architecture on FPGA for terabit packet classification," in *IEEE HOTI*, 2012.

[50] Y.-K. Chang and Y.-H. Wang, "CubeCuts: A novel cutting scheme for packet classification," in *IEEE AINA*, 2012.

[51] A. Kennedy and X. Wang, "Ultra-high throughput low-power packet classification," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 22, pp. 286–299, 2014.

[52] Y.-K. Chang and H.-C. Chen, "Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA," *The Computer Journal*, vol. 62, no. 2, pp. 198–214, 2019.

[53] 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. 38 673–38 681, 2020.

[54] R. Panigrahy and S. Sharma, "Sorting and searching using ternary CAMs," in *IEEE HOTI*, 2003.

[55] Y.-K. Chang, "A 2-level TCAM architecture for ranges," *IEEE Transactions on Computers*, vol. 55, no. 12, pp. 1614–1629, 2006.

[56] W. Li, D. Li, X. Liu, T. Huang, X. Li, W. Le, and H. Li, "A power-saving pre-classifier for TCAM-based IP lookup," *Computer Networks*, vol. 164, p. 106898, 2019.

[57] J. Tan, G. Lv, Y. Ma, and G. Qiao, "High-performance pipeline architecture for packet classification accelerator in DPU," in *IEEE ICFPT*, 2021.

[58] Y. R. Qu, S. Zhou, and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA," in *ACM/IEEE ANCS*, 2013.