A Heterogeneous and Adaptive Architecture for Decision-Tree-Based ACL Engine on FPGA

Yao Xin, Chengjun Jia, Wenjun Li, Ori Rottenstreich, Yang Xu, *Member, IEEE*, Gaogang Xie, Zhihong Tian, *Senior Member, IEEE*, and Jun Li

***Abstract*—Access Control Lists (ACLs) are crucial for ensuring the security and integrity of modern cloud and carrier networks by regulating access to sensitive information and resources. However, previous software and hardware implementations no longer meet the requirements of modern datacenters. The emergence of FPGA-based SmartNICs presents an opportunity to offload ACL functions from the host CPU, leading to improved network performance in datacenter applications. However, previous FPGA-based ACL designs lacked the necessary flexibility to support different rulesets without hardware reconfiguration while maintaining high performance. In this paper, we propose HACL, a heterogeneous and adaptive architecture for decision-tree-based ACL engine on FPGA. By employing techniques such as tree decomposition and recirculated pipeline scheduling, HACL can accommodate various rulesets without reconfiguring the underlying architecture. To facilitate the efficient mapping of different decision trees to memory and optimize the throughput of a ruleset, we also introduce a heterogeneous framework with a compiler in CPU platform for HACL. We implement HACL on a typical SmartNIC and evaluate its performance. The results demonstrate that HACL achieves a throughput exceeding 260 Mpps when processing 100K-scale ACL rulesets, with low hardware resource utilization. By integrating more engines, HACL can achieve even higher throughput and support larger rulesets.**

***Index Terms*—ACL, SmartNIC, FPGA, pipeline, parallel processing.**

Received 22 December 2023; revised 18 August 2024; accepted 29 September 2024. Date of publication 10 October 2024; date of current version 12 December 2024. This work was supported in part by the Major Key Project of Peng Cheng Laboratory under Grant PCL2023A06; in part by the National Natural Science Foundation of China under Grant 61872212, Grant 62372123, Grant 62102203, Grant 62072430, Grant 62372129, Grant 62172108, and Grant U20B2046; in part by the Science and Technology Innovation Project of Guangdong under Grant 2023TQ07X362 and Grant 2023TQ07X004; in part by the Basic Research Program under Grant 2021-JCJQ-JJ-0483; in part by the National Key Research and Development Program of China under Grant 2021YFB2012402; and in part by the Key-Area Research and Development Program of Guangdong under Grant 2021B0101400001 and Grant 2020B0101130003. Recommended for acceptance by J. Shu. *(Yao Xin and Chengjun Jia contributed equally to this work.) (Corresponding authors: Wenjun Li; Jun Li.)*

Yao Xin and Zhihong Tian are with the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510530, China, and also with Huangpu Research School of Guangzhou University, Guangzhou 510530, China (e-mail: xinyaobuaa@126.com; tianzhihong@gzhu.edu.cn).

Chengjun Jia is with Huawei Technologies Company, Ltd., Beijing 100095, China (e-mail: jiachengjun2@huawei.com).

Wenjun Li is with Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: wenjunli@pku.org.cn).

Ori Rottenstreich is with the Technion, Haifa 3200003, Israel (e-mail: or@technion.ac.il).

Yang Xu is with the School of Computer Science, Fudan University, Shanghai 200433, China (e-mail: xuy@fudan.edu.cn).

Gaogang Xie is with the Computer Network Information Center, Chinese Academy of Sciences, Beijing 100083, China (e-mail: xie@cnic.cn).

Jun Li is with the Department of Automation, Tsinghua University, Beijing 100084, China (e-mail: junl@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TC.2024.3477955

I. Introduction

ACCESS Control List (ACL) is a security mechanism used to manage access to resources and prevent unauthorized access based on predetermined rules, which can lead to data breaches, security threats, and loss of confidential information [1]. In modern networks, ACL is an essential tool for enforcing security policies and regulations, which is crucial for ensuring the confidentiality, integrity, and availability of information assets. A typical ACL ruleset is shown in Table I. For each packet, the classifier engine searches the ruleset and outputs the action of the highest-priority rule matched for subsequent processing.

With the trends of software-defined networking (SDN) and programmable networking, SmartNIC is emerging as a promising unit for offloading network functions from server CPUs [2]. In modern datacenter networks (DCNs), there are several key requirements for a high-performance SmartNIC ACL engine compared to traditional ones: 1) *Ample rule capacity* which can accommodate over 100K rules, to account for the multitude of tenants in a cloud datacenter or the extensive user equipment in a carrier network. 2) *Ability to update rulesets*, due to the fact that the virtual functions of the network protocol stack in datacenters can be updated frequently. 3) *High traffic throughput*, to keep pace with the rising adoption of 100GbE NICs and the recent introduction of 400GbE NICs in modern datacenters. 4) *Low classification latency*, to avoid directly extending the completion time for services, as the ACL engine is one step of end-to-end network processing.

There are three primary hardware compositions utilized in SmartNICs: multi-core processors, specific ASICs, and FPGAs. Multi-core processors [3], known for their flexibility, can update the ruleset and expand capacity (supporting 100K+ rules) using software libraries like DPDK or Hyperscan, without relying on specific rulesets. However, handling 100Gbps traffic with multi-core processors requires a significant number of CPU cores. Running a recently-proposed popular algorithm [4] consumes more than 30 cores, and the throughput is limited by DRAM bandwidth under large rulesets [5].

TABLE I
EXAMPLE ACL RULES

Id	$Addr_{src}$	$Addr_{dst}$	$Port_{src}$	$Port_{dst}$	Protocol	*Action*
R_1	175.77.88.155	119.106.158.230	*	80	0x06 (TCP)	a_1
R_2	95.105.143.33	144.209.187.155	*	27400	0x06 (TCP)	a_2
R_3	95.105.142.0/23	193.4.164.231	*	*	0x06 (TCP)	a_3
R_4	95.105.143.51	204.13.220.0/22	*	*	0x01 (ICMP)	a_4
R_5	95.105.143.6	192.206.76.132	*	*	*	a_3
R_6	0.0.0.0/0	0.0.0.0/0	*	*	0x01 (ICMP)	a_4
R_7	0.0.0.0/0	0.0.0.0/0	*	*	*	a_5

The ASIC-based SmartNIC, such as the NVIDIA ConnectX series [6], can have a programmable data path that is relatively simple to configure. However, this functionality is constrained by predefined functions within the ASIC [7], [8]. TCAM (Ternary Content Addressable Memory), a traditional and dominant ASIC solution used in traditional switches for ACL in the industry [9], [10], is capable of achieving high throughput and ultra-low latency. However, this comes at the expense of limited capacity, high chip area, and high power consumption. Meanwhile, the atomicity and high cost of rule updating are two other big challenges for TCAM [11]. For traditional routers/switches widely used in regular backbone networks and enterprise campuses, TCAM is sufficient to cope with their small ACL scales. However, the above reasons make TCAM unsuitable for large-scale ACL scenarios in virtualized cloud data centers in recent years.

FPGA-based SmartNIC opens up a myriad of possibilities owing to its flexibility and parallelism [12]. Notably, in the realm of ACL, there has been a surge of FPGA designs in recent years. However, it is imperative to acknowledge the limitations of previous solutions such as the BitVector (BV)-based [13] [14] or hash-based [15] approaches, which unfortunately only support approximately 1K-scale rules and require the hardware architecture to be reset for rule updates. In contrast, decision-tree-based algorithms are proven to be highly suitable in scenarios characterized by large rulesets and high-performance requirements.

Previous FPGA decision-tree designs can be categorized into two main groups: fixed-pipeline [16], [17] and non-pipeline [18], [19]. However, these designs have certain limitations that hinder their practicality. Fixed-pipeline designs face a challenge when the number of nodes at a certain level exceeds the preallocated capacity. This limitation prevents the system from accommodating rule updates, restricting its flexibility. On the other hand, non-pipeline designs mainly employ Run-to-Completion (RTC) technique, whose throughput drops sharply when the depth of the tree is high. The design paradigm of previous decision tree dedicated architectures is shown in Fig. 1(a): the ruleset first determines the data structure, which in turn determines the hardware architecture. This paradigm highlights the limitation that none of the current FPGA designs offer the necessary flexibility to accommodate various rulesets without requiring hardware reconfiguration. This lack of adaptability hampers their usability in real-world applications with dynamic rule requirements.

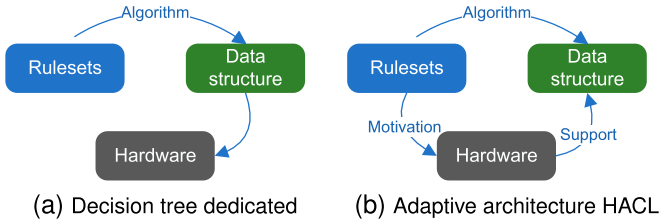


Fig. 1. The methodology of designing for two different architectures.

In this paper, we propose **HACL**, a **H**eterogeneous and **A**daptive A**CL** engine that utilizes the decision tree approach on FPGA-based SmartNICs. Unlike previous FPGA designs, we have adopted a completely innovative paradigm, as depicted in Fig. 1(b). The architecture is built based on the potential characteristics of rulesets that may need to be accommodated, and it supports arbitrary data structures generated by various decision tree algorithms, as long as there is sufficient memory space. Besides its high flexibility, HACL can provide a higher rule capacity than TCAM and a throughput far exceeding that of the CPU. It also reduces the processing pressure on the back-end engine through the cache design of hit rules. The major contributions are summarized as follows:

- We propose a heterogeneous architecture that takes advantage of both pipeline and RTC technique. HACL leverages the recirculation of a pipeline to adapt to the variation of decision tree depth. Each pipeline node supports both cutting and splitting method for the construction of decision trees.
- We also develop a heterogeneous framework in which, in addition to HACL on the FPGA platform, there is a software compiler based on the CPU platform that can effectively map decision trees to different types of FPGA pipeline modules.
- We implement HACL on a typical SmartNIC, Xilinx U50, and the evaluation demonstrates that the resource requirement of HACL is pretty low, and HACL achieves over 260Mpps (Million packets per second) throughput for various 100K rulesets.

The paper is organized as follows. We briefly introduce the decision-tree algorithm in Section II and related work in Section III. Then we elaborate the design of HACL including FPGA architecture and compiler in Section IV and Section V. We illustrate some evaluation results to verify the feasibility and high performance of HACL in Section VI.

II. BACKGROUND

The ACL matching problem belongs to the area of multi-field packet classification [20], [21]. The goal is to classify network traffic by comparing d fields of packet headers to a predefined ruleset. A ruleset R consists of the ordered rules: $r_1 < r_2 < \ldots < r_n$. Each rule is described by a cartesian product of d fields, i.e.,

$$r_i = F_1^i \times F_2^i \times \ldots \times F_d^i \quad (1)$$

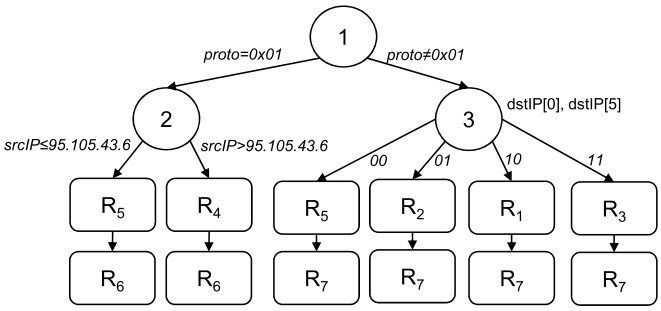


Fig. 2. An example decision tree for Table I.

where F_j^i represents a finite set on the packet j-th header field. F_j^i can be expressed by prefix, range, or exact value, which are all continuous on the integer domain. A rule r_i can be viewed as a specific hypercube in the space with d fields. When a packet with the header $(v_1, v_2, \ldots, v_d)$ is classified, the engine would find the matched rule with the highest order. Note that ACL configuration constraints do not allow rules with the same order (or priority) to appear. All matched rules can be expressed as a set

$$R' = \{r_k | r_k \in R; v_j \in F_j^k, \forall j = 1, 2, \ldots, d\} \tag{2}$$

and the final matching result is the rule with the highest priority:

$$\hat{r} : \forall r_k \in R' \text{ and } r_k \neq \hat{r}, \hat{r} < r_k \tag{3}$$

For the decision tree classification algorithm, a corresponding typical decision tree is depicted in Fig. 2 for the ACL ruleset in Table I. When the classifier lookups the matched result of a packet, it reads the required field of the packet and decides where to go for the next node until it reaches a leaf node. Then the classifier would search linearly against the leaf rules. For example, if a packet includes the 5-tuple header (95.105.143.6, 192.206.76.132, 20, 40, 0x06), the classifier would first judge if the protocol field (the 5th field) is equal to 0x01 according to the requirement of the root node. As the protocol field value is 0x06, the classifier would go to the far right and then read the 1th and 6th bit of $Addr_{dst}$ (the 2nd field) to determine the next direction. The value of $Addr_{dst}$ is '00', so it compares the packet against both R_5 and R_7 to get the matched result with the highest priority, R_5.

Different algorithms potentially build different trees based on their heuristic observations and the operations at nodes could be various. For example, HiCuts [22] chooses several continuous bits of one selected dimension to *cut* the packet header; HyperCuts [23] and EffiCuts [24] use the combination of a few continuous bits from several dimensions; while BitCuts [25] picks several discrete bits. HyperSplit [16] and SmartSplit [26] compare the field value with a specific value to *split* to reduce the size of the decision tree. CutSplit [20] and ByteCuts [27] allow *cut* or *split* operations within a node to further expand the decision space and build better trees at lower depths.

This diversity in decision tree structures emphasizes the complexity and variability of ACL rulesets and highlights the need for adaptable hardware that can effectively handle diverse rule configurations. Fig. 3 displays the number of tree nodes across different layers within a tree constructed using CutSplit for various ACL rulesets. Note that the term "nodes" encompasses both intermediate nodes and leaf nodes in this context. One notable observation is that even when dealing with rulesets containing the same number of rules, the decision trees produced by the same algorithm exhibit significant variations. This observation is not limited to CutSplit alone; it extends to other algorithms such as HiCuts, EffiCuts and other methods as well.

III. Related Work

When running in a CPU-DRAM environment, the uncertainty of the tree depth and the diversity of node counts are not an issue. Memory can be dynamically allocated for different levels of the tree. However, if we want to fully utilize the parallel and pipelined processing capabilities in an FPGA, diversity becomes a major issue. According to the characteristics of hardware, FPGA designs based on decision trees can be mainly divided into two kinds: fixed pipeline and non-pipeline. Fig. 4 compares fixed-pipeline and non-pipeline mechanisms through a node search example with a depth of 3.

Fixed-pipeline architectures account for the majority of decision-tree-based FPGA designs, and representative works include ParaSplit [16], CubeCuts [28], REC [29], and MBitTree [30]. As shown in Fig. 4, the tree node information is distributed in multiple SRAMs corresponding to different pipeline stages, and results are output every cycle so that classification throughput is directly linear to operating frequency. However, the number of pipeline stages will be determined according to the data structure generated by the specific ruleset. If the compiled tree depth is larger than preallocated, the FPGA must be reconfigured, which makes the ruleset update unavailable in this scenario.

For non-pipelined designs based on decision trees, such as UTPC [18], TcbTree [19], and KickTree [31], they adopted a flexible approach of storing all nodes at different levels of a tree in a single SRAM. This method can accommodate nodes of various rulesets without reconfiguration and breaks the limit of tree depth, which is much more versatile than the fixed-pipeline counterparts. However, different nodes would be accessed in a serial manner, so that the throughput is limited to $\frac{B}{D}$pps where B is the SRAM frequency and D is the tree depth, abandoning the FPGA advantages. For example, assuming that the FPGA operates at 300MHz, the SRAM can be read $3 * 10^8$ times per second. As in Fig. 4, if the depth of the tree is 3, it means that the SRAM needs to be accessed sequentially up to 3 times to get a result, then the number of results can be computed per second is $3 * 10^8/3 = 1 * 10^8$ times, corresponding to a minimum throughput of 100Mpps, which is 1/3 of the frequency. This performance gap can be bridged by duplicating multiple processing cores, which in turn imposes significant demands on storage resources. For smaller-capacity FPGAs, it becomes impossible to accommodate multicores, leading to scalability issues with this method.

Although FPGA-based ACL has been actively investigated for many years, as far as we know, none of them can achieve

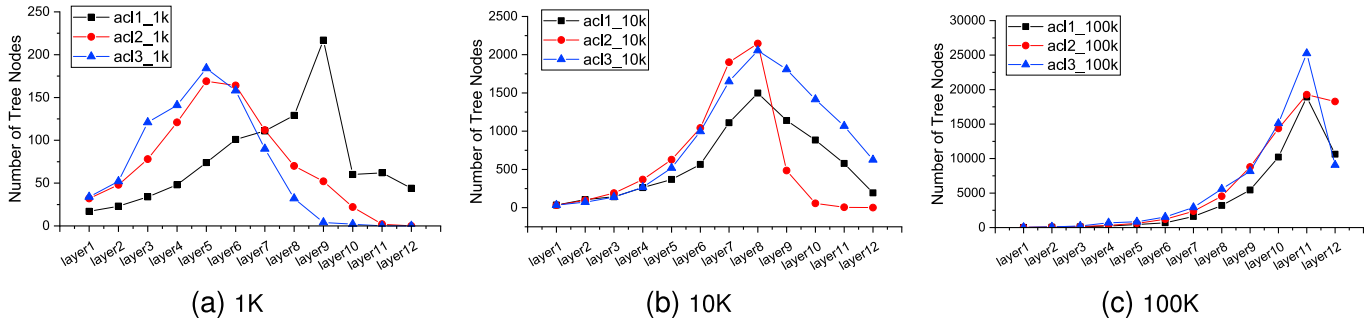


Fig. 3. Number of nodes (intermediate and leaf nodes) at different layers in the tree built using CutSplit for various ACL rulesets.

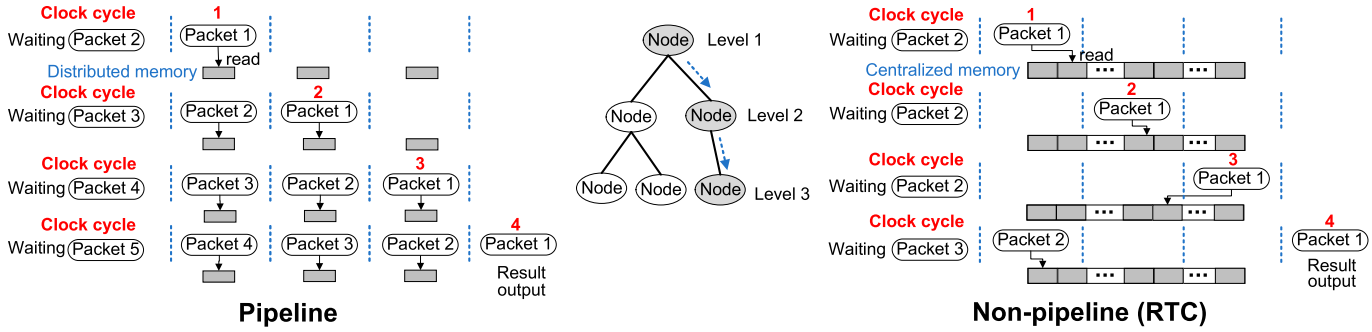


Fig. 4. Fixed-pipeline vs. Non-pipeline.

TABLE II
ACL ENGINE ON FPGA

Architecture	*Rule*		*Classification*	
	Capacity	*Update*	*Throughput*	*Latency*
Fixed pipeline [16], [17], [32]	Low	Hard	**High**	**Low**
Non pipeline [18], [19], [31]	**High**	**Easy**	Low	Uncertain
HACL (Heterogeneous)	**High**	**Easy**	**High**	**Low**

both flexibility and high performance at the same time. Our design HACL is aimed at mapping the decision trees to one fixed pre-configured FPGA circuit. For a specific decision-tree algorithm, the structure of multiple trees is different for various rulesets. They have diverse depths and various nodes at each level, but they can all be mapped in HACL. In the preliminary version of this paper [33], pipelines and non-pipelined linear search units are mixed and scheduled by a Network-on-Chip (NOC), resulting in a complex structure and large resource consumption. While this work fully leverages the advantage of pipelines to achieve high throughput of classification with a simpler structure and low hardware resource consumption. By comparing HACL and existing alternatives in Table II, it can be seen that HACL could achieve high rule capacity and high throughput simultaneously.

IV. ARCHITECTURE DESIGN

The fixed-pipeline architecture heavily relies on the ruleset characteristics, meaning the number of pipeline stages is determined by the data structure generated from a specific ruleset.

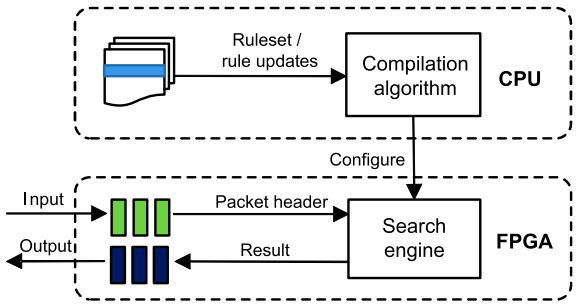


Fig. 5. The framework of HACL.

On the other hand, the non-pipeline architecture faces performance bottlenecks due to centralized storage. HACL takes the advantages of both approaches, to provide a certain degree of flexibility while ensuring performance. HACL adopts typical grouping and multi-domain decision-making decision tree algorithms to facilitate the design of hardware search engines. While achieving high throughput and high capacity, it is adaptable to various rulesets to cope with frequent updates of DCN rules, preventing the hardware solution from becoming obsolete after changes to the ruleset.

As shown in Fig. 5, HACL consists of two parts: a search engine located on the FPGA and a compiler located on the CPU. The original ruleset or rule updates are input into the compilation algorithm, which outputs configurations for the search engine. These configurations update the data structure stored on the search engine. The input to the search engine is

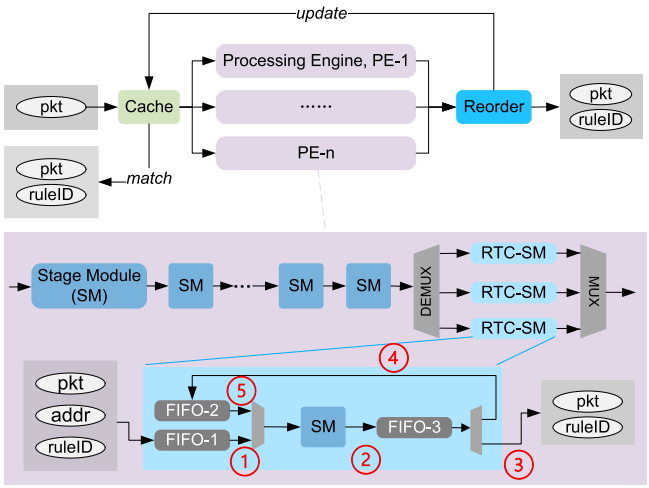


Fig. 6. HACL's search engine architecture diagram.

the header of each packet to be searched, and the output is the ID of the highest priority matching rule.

A. The Search Engine Overall Architecture

As shown in Fig. 6, the HACL search engine architecture is mainly composed of three parts: Processing Engines (PEs), Reorder Module, and Cache Module. We need to consider the versatility of the architecture for decision tree algorithms. Most of mainstream decision tree algorithms are based on multiple trees. Rulesets are grouped according to the prefix length of each field and each decision tree is established. Thus multiple PEs in the architecture correspond to individual decision trees. Each PE contains multiple Stage Modules (SMs) and Run-to-Completion Stage Modules (RTC-SMs), which are the core modules to search rules. It employs a hybrid architecture, where the majority of decision tree layers are implemented using pure pipelining, while the unevenly distributed tail nodes of multiple decision trees are allocated to RTC-SMs.

Although the RTC architecture is generally not considered suitable for high-performance designs, in this context, the RTC structure is not directly responsible for the entire decision tree search; rather, it handles only the final processing of tree nodes. This approach significantly conserves storage space. If the entire decision tree were implemented using pure pipelining, the sparse distribution of nodes in the tail layers would lead to substantial storage inefficiencies, as large amounts of memory would need to be pre-allocated, often resulting in underutilization.

To prevent the RTC-SM from becoming a performance bottleneck, multiple RTC-SMs operate in parallel, thereby matching the throughput of the preceding pipelined stages. Unlike previous designs, where each RTC involved redundant node replication, our approach assigns unique nodes to each RTC-SM module. This method avoids wasting storage resources and achieves higher resource utilization compared to a purely pipelined approach.

Moreover, Reorder module is used to merge the output results of multiple PEs, and the Cache module is used to improve overall throughput and reduce processing pressure on the SM by utilizing the locality features of the packet flow. In the following sections, modules would be introduced according to the processing flow of packets in the engine.

B. Stage Module

In each PE, multiple SMs are connected in series to form a pipeline, and each SM corresponds to one of the stages. When a packet enters the first SM module, the node data is read from the 0th address of the local storage unit by default. The node data (shown in Fig. 7) indicates whether it is an internal node (storing classification operations) or a rule node (storing a rule). The packet information and the data stored in the node are combined to calculate the node address, *addrNxt*, required for the next step of the packet lookup process. The rule node determines whether the rule matches the packet, and updates the matching result accordingly. Next, the data packet continues forward and the next SM to be queried is determined based on *addrNxt*. The node address, packet information, and matching result together form the search instruction passed between SMs. Each SM performs operations based on the input search instruction and outputs the updated instruction to the next SM. Multiple levels of SMs form an unobstructed pipeline structure until the search instruction is finally transmitted to the RTC-SM. Note that the node address consists of two parts: module ID and memory address in the module. This encoding method allows the lookup engine to access the third SM directly without reading the node on the second SM, after accessing the first SM. In other words, an empty operation is performed on the second SM. This design makes memory allocation on the search engine more flexible.

C. Run-to-Completion Stage Module

Multiple RTC-SM modules are connected after the SM pipeline. Each RTC-SM has a distinct module ID, and an RTC-SM is selected to process each search instruction output by the last level SM. RTC-SM has three First-In-First-Out (FIFO) queues for caching search instructions. FIFO-1 caches the input instruction, FIFO-3 caches the output instruction from RTC-SM. If the node address in the instruction output by FIFO-3 is 0, then the search process has ended, and the packet information and matching result will be output to the backend. If the module ID in the node address output by FIFO-3 points to the current RTC-SM, then the instruction will be output to FIFO-2. The SM in RTC-SM selects and processes the lookup instruction from FIFO-1 and FIFO-2, and outputs the processing result to FIFO-3. In this way, packets can be cycled through within an RTC-SM. The following is an example to illustrate the working principle of RTC-SM: Assume that RTC-SM contains only one SM, with the ID *SMRamID*.

1) An instruction containing the packet *pkt* reaches FIFO-1 from the previous pure SM pipeline section, entering the SM via path ① at the bottom of Fig. 6. Upon lookup, the SM outputs the address for the next lookup, *addrNxt*, which is then input to FIFO-3 via path ②.

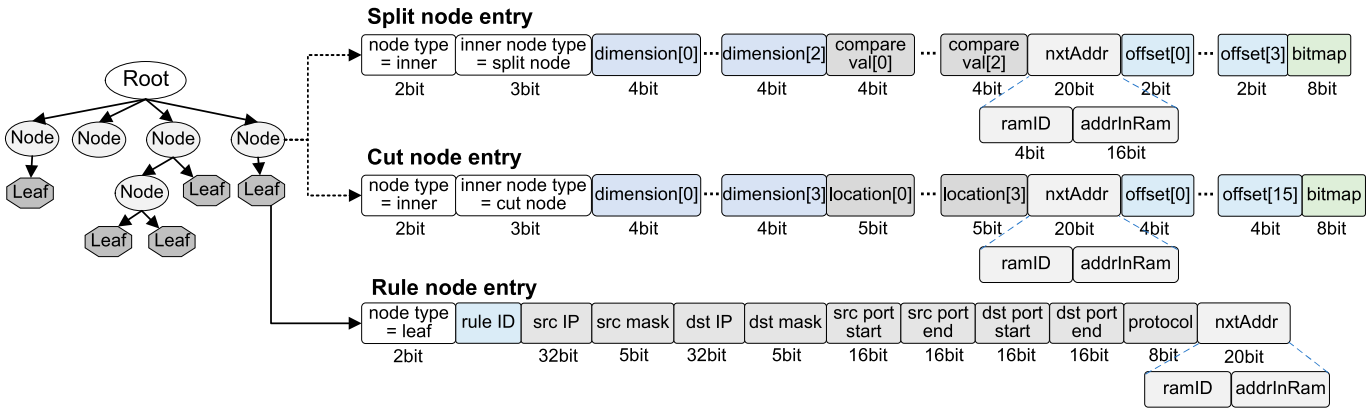


Fig. 7. Data structure of different types of nodes stored in the hardware.

2) Subsequently, the value of *addrNxt* is checked. If it is not 0, and *addrNxt.ramID* = *SMramID*, *pkt* returns to FIFO-2 through path ④.
3) *pkt* reenters the SM via path ⑤, and after lookup, the SM outputs the address for the next lookup, *addrNxt*, which is then input to FIFO-3 via path ②.
4) The value of *addrNxt* is then checked. It is found to be 0, indicating the end of the lookup process, and the result is output via path ③.

The recirculation design allows packets to read node information for any number of times, thereby handling the situation where "different rulesets require different numbers of nodes to be read". The number of RTC-SMs in each PE is set to 3. Experiments show that this quantity is sufficient to ensure that the pipeline composed of SMs will not be suspended in general.

D. Search Process

The processing of the search module is illustrated by the pseudocode in Algorithm 1, with three inputs/outputs.

- The packet information *pkt* is used to pass the value of the packet header. The values of multiple fields are treated as an array *pkt.fields*, and the total length of the array is 5 for 5-tuple. The SM can select a specific field from it. In addition, the values of each field are concatenated in order and can also be viewed as a bit string *pkt.bits*. The SM can select a certain bit from it.
- The node address *addr* is used to pass the address information of the next node to be accessed, which consists of two parts: the module number *ramID* and the memory address in that module *addrInRam*. Each SM in the pipeline has a unique identifier *SMRamID*, which is used to distinguish between different search modules.
- The *ruleID* is used to pass the ID of the matched rule during the search process. 0 indicates that no rules match, and the higher the priority of the rule, the larger the *ruleID*.

The processing of the SM mainly consists of three steps:

1) Compare whether the input *addr* belongs to the current SM (see line 3 in Algorithm 1). If not, all inputs are forwarded to the next SM. If it belongs, move on to step 2.

Algorithm 1 The search process of Stage Module (SM)

```
Input: pkt, addr, ruleID, SMRamID
Output: pktNxt, addrNxt, ruleIDNxt
 1: procedure SM
 2:     pktNxt ← pkt
 3:     if addr.ramID != SMRamID then
 4:         addrNxt ← addr; ruleIDNxt ← ruleID
 5:         return
 6:     // Read node information from the addr.addrInRam to node
 7:     switch node.type do
 8:         case NODETYPE_RULE                                  // Rule node
 9:             addrNxt ← node.nextAddr
10:             // Compare each field of pkt headers with the corresponding range given by node.rule in parallel
11:             if rule matched then
12:                 ruleIDNxt ← max(ruleID, node.ruleID)
13:             else
14:                 ruleIDNxt ← ruleID
15:         case NODETYPE_CUT                                   // Cut node
16:             ruleIDNxt ← ruleID
17:             for i = 0 to 3 do
18:                 cutBits[i] ← pkt.bits[node.cutLoc[i]]
19:             if node.isNext[cutBits] == 1 then
20:                 addrNxt ← node.nextAddr + node.offset[cutBits]
21:             else
22:                 addrNxt ← 0
23:         case NODETYPE_SPLIT                                 // Split node
24:             ruleIDNxt ← ruleID
25:             for i = 0 to 2 do
26:                 cmpFieldVal ← pkt.fields[node.splitLoc[i]]
27:                 if node.cmpVal[i] >= cmpFieldVal then
28:                     splitBits[i] ← 1
29:                 else
30:                     splitBits[i] ← 0
31:             if node.isNext[splitBits] == 1 then
32:                 addrNxt ← node.nextAddr + node.offset[splitBits]
33:             else
34:                 addrNxt ← 0
```

2) Read the corresponding node information from memory into the register *node* based on the input *addr* (see line 6 in Algorithm 1).
3) Perform different processing based on the node type, update the output *addr* and matched rule (see lines 7-34 in Algorithm 1), and output *pkt* without modification.

Fig. 7 depicts the data structure of different decision tree nodes stored in the hardware. The current version of HACL defines three types of nodes: rule nodes that stores rules, and two types of nodes that store intermediate node information: cut nodes and split nodes, which respectively correspond to two common methods of building decision trees. More types can be further defined as needed.

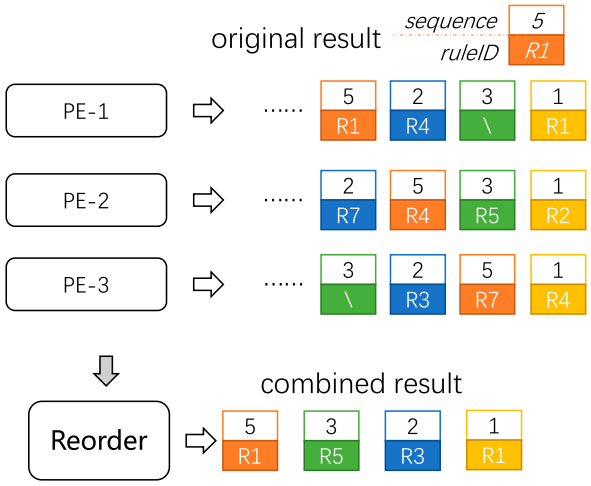


Fig. 8. The Function diagram of the out-of-order Reorder module.

The **rule node** stores detailed information about the rule and its ID. The SM compares each field of the packet in parallel to see if it falls within the range defined by the rule. If all fields match, the packet is considered a match and the output rule ID is updated. Otherwise, the previous input rule ID is retained in the output.

The **cut node** selects up to four bits from *pkt.bits*, combines them into *cutBits*, and uses *cutBits* as an index to access the boolean array *node.isNext*. If *isNext[cutBits]* is 1, there will be a next node to visit, and *addrNxt* is calculated based on *node.nextAddr* and node.offset[cutBits]. If *isNext[cutBits]* is 0, *addrNxt* is set to 0 (which means the packet does not match any rule). When *addrNxt.ramID* is 0, the packet will not be processed by any subsequent SMs.

The **split node** selects three fields from the packet header (which may include duplicates) and compares them to the three values stored in the node. The results of the comparisons are concatenated into a 3-bit *splitBits*, which is similar to *cutBits* used in cut nodes.

For the search process of a PE, the best-case scenario is that the packet completes the search in the pure pipeline section without entering the RTC-SM module. The worst-case scenario is that after completing the search in the pure pipeline section, a packet enters an RTC-SM module and cycles through it m times to complete the search, where m is the number of RTC-SM modules. However, HACL sets up multiple RTC-SM modules to ensure stable overall performance in various cases.

E. Out-of-Order Reorder Module

RTC-SM completes the final search and outputs the matching results to a FIFO, which is then sent to the Reorder module. The Reorder module is responsible for aggregating the results from multiple PEs and outputting the final matching result. In HACL, a ruleset is partioned into groups and assigned to multiple PEs. Each input packet is sent to all PEs for search, and each PE will provide a matching result: whether there is a match and the ID of the matching rule. To obtain the ID of the highest-priority matching rule, multiple results (rule IDs) generated in different PEs for the same packet must be aggregated and resolved. Although a packet (corresponding to a unique sequence number) enters each PE simultaneously, the time it takes to output the result from each PE is different due to the unpredictable number of iterations within the RTC-SM. Therefore, the Reorder module aligns the out-of-order results from different PEs based on the packet's sequence number. Note that multiple results generated by one packet correspond to the same sequence number. The final matching result output from the Reorder module will also be fed back to the Cache module at the front of the search engine to update the corresponding cache entry.

In Fig. 8, three PEs output a series of matching results respectively. Taking PE-1 as an example, it outputs in turn: the packet with sequence number 1 matches R1, packet 3 does not match any rules, packet 2 matches R4, and packet 5 matches R1. Since the RTC-SM in Fig. 6 cannot guarantee the processing delay of packets, although packet 2 is input to PE-1 first, its output result is after packet 3. The Reorder module aggregates the results of the three PEs and output the highest priority matching result. For instance, packet 2 corresponds to outputs of R4 by PE-1, R7 by PE-2, and R3 by PE-3. Then the final result would be R3 which has the highest priority.

The out-of-order rearrangement function of HACL has similarities with the classic Push-In-First-Out (PIFO) [34] queue scheduling. PIFO rearranges results that do not arrive in order internally, and the arranged results are output in ascending order by sequence number. The Reorder module of HACL can use multiple PIFO queues to perform reordering on each PE separately, and then merge the results. However, PIFO maintains elements as a complex ordered linked list structure or heap structure, which will occupy a lot of hardware resources and is not suitable for use in HACL. On the other hand, there is an upper limit to the number of packets that HACL can accommodate, which is the sum of three FIFOs in RTC-SM and all SM modules. If the depth of each FIFO is 64 and there are 3 RTC-SMs and 10 SMs in each PE, then the maximum number of unmatched packets that the PE can accommodate is 64*3*3 + 10*3 = 606. Therefore, the sequence number only needs to be encoded using 10 bits, which is much smaller than the default 16b/32b of PIFO. If PIFO is directly used in the Reorder module, there will be a lot of redundancy. In summary, the Reorder module of HACL needs to be redesigned instead of directly reusing PIFO. In this work, we propose two schemes to address this issue. The first scheme is to use a large number of on-chip registers, which is called the ReorderReg scheme (Fig. 9); the second solution is to reduce the on-chip register footprint by doubling the BRAM footprint, called the ReorderRAM scheme (Fig. 10).

1) ReorderReg Scheme: As shown in Fig. 9, ReorderReg uses three blocks of RAM, each with independent read and write functionalities, responsible for recording the results of the three PEs in Fig. 8. The packet sequence number is indexed as the address in RAM, and the matching rule ID is written to the corresponding position in RAM. For each address, ReorderReg uses 3 bits (i.e., flag bit) for each address to record the readiness of the rule ID corresponding to the current packet. The first three

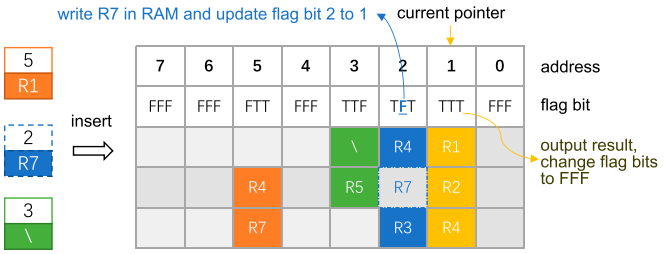


Fig. 9. The ReorderReg scheme.

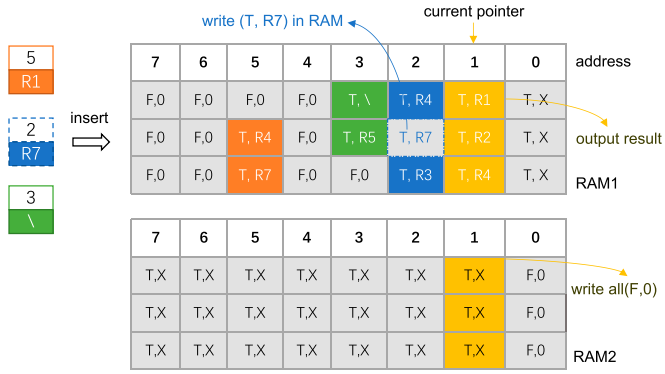


Fig. 10. The ReorderRAM scheme.

results from Fig. 8 have been stored in RAM, and the fourth set of results is being inserted. For example, for the (2, R7) result that PE-2 wants to insert, its rule ID result R7 is written to the position at address 2 in the second block of RAM, RAM[1][2] = R7, and the corresponding flag at position flags[2][1] is set to True. When the bit values of flags[i] are all True, the results for the three PEs of the packet with sequence number i have all been recorded, as shown in flags[1] in Fig. 9.

ReorderReg also has a pointer pointing to the next packet sequence number to be output, which in Fig. 9 points to sequence number 1. Since flags[1] is all True, if the backend requests output results at this point, ReorderReg will read the three results from RAM[0][1], RAM[1][1], and RAM[2][1] in parallel, then calculate the maximum rule ID among them, which is R1, and clear flags[1], and move the pointer to the next position with sequence number 2. If the current flags are not all True, then the pointer will not move forward, and ReorderReg will notify the backend that the data is not yet ready.

In HACL, the packet sequence number increases directly by 1, and when the number reaches the maximum value, it returns to 0. Therefore, ReorderReg's use of RAM is similar to a circular queue, allowing for the reuse of RAM. For ReorderReg, each block of RAM is only written when the corresponding PE outputs a matching result, and is only read when the result is required. Therefore, at most one read operation and one write operation can be performed in each clock cycle, and it can be implemented using standard dual-port RAM. The entire workflow can be fully pipelined, meaning that input and output processing can be performed on every cycle without blocking. Compared to PIFO, ReorderReg uses bucket sorting instead of insertion sorting, reducing the computational complexity of insertion and increasing throughput. In addition, by enforcing sequential output (the matching result with sequence number 3 must be output after the matching result with sequence number 2 is output), ReorderReg eliminates the linked list relationship in PIFO and reduces the complexity of hardware implementation.

2) ReorderRAM Scheme: The ReorderRAM scheme is similar to the ReorderReg design idea, but it utilizes two groups, a total of 6 RAMs to achieve the same function. In Fig. 10, ReorderRAM no longer maintains flags, but writes the flag bit information and rule ID into the corresponding position of RAM. The insertion operation of (2, R7) is to directly set RAM1[1][2]=(True, R7). When outputting the result, ReorderRAM will read RAM1[0][1], RAM1[1][1], RAM1[2][1] at the position pointed by the current pointer in parallel. If all three values are recorded as True, it means the location is ready to output the final merged result, and the values of RAM2[0][1], RAM2[1][1] and RAM2[2][1] are simultaneously set to (F, 0) to indicate that the location is cleared. The serial number of the packet in HACL is 10b by default, and there will be 2^{10} addresses of RAM1. When the serial number reaches the maximum value and overflows to 0, ReorderRAM will not directly return to the original 0 of RAM1 like ReorderReg, but switches to RAM2. Similarly, when the pointer overflows from the maximum address of RAM2 to 0, it switches to RAM1. The reason for such design is that, when input and output exist at the same time, the input operation will perform a RAM write (the rule ID result is written to the corresponding position), the output operation will perform a RAM read (the data pointed to by the pointer is read) and a RAM Write (clear the location of successful output). ReorderRAM performs two RAM writes and one RAM read within one clock, and operates data of three addresses at the same time. This operation cannot be realized on a dual-port RAM, thus ReorderRAM realizes the above functions by operating two RAMs.

When the backend continues to request data and the Reorder module needs to do continuous output, ReorderRAM has certain disadvantages compared to ReorderReg. ReorderReg directly judges whether the data of the current pointer is ready through the register flags, without reading delay. ReorderRAM needs to wait for the data on the RAM to return, and there must be a delay from the initiation of the read command to the return of the data (the read delay of FPGA's BRAM/URAM is 1-3 clock cycles). Table III illustrates the scenario where the RAM read latency is 2 clock cycles.

The upper table in Table III shows a case when ReorderRAM does not have any optimization and adopts the "simple read" strategy. ReorderRAM initiates a request to read address 10 data in the first cycle, gets the data in the third cycle, and then judges that the data is ready. Then the pointer moves to the next position, and ReorderRAM in the fourth cycle initiate a request to read data at address 11. It can be seen that under this strategy, ReorderRAM outputs every 3 clock cycles, and the throughput is very low. To address this issue, ReorderRAM adopts the "adventurous read" strategy. After requesting the data of address 10 in the first cycle, although the current pointer still points to address 10 in the second cycle, a request for the

TABLE III
ADVENTUROUS READING OF REORDERRAM (THE RAM ACCESS LATENCY IS 2 CLOCK CYCLES)

Simple reading

Clock	1	2	3	4	5	6	7	8	9
Request address	10	-	-	11	-	-	12	-	-
Return data	-	-	10	-	-	11	-	-	12
Data success	-	-	T	-	-	T	-	-	T
Current pointer	10	10	10	11	11	11	12	12	12
Output result	-	-	10	-	-	11	-	-	12

Adventurous reading

Clock	1	2	3	4	5	6	7	8	9
Request address	10	11	12	13	14	15	16	14	15
Return data	-	-	10	11	12	13	14	15	16
Data success	-	-	T	T	T	T	F	T	F
Current pointer	10	10	10	11	12	13	14	14	14
Output result	-	-	10	11	12	13	14	-	-

data of address 11 is initiated, so that the data at address 10 can be obtained in the third cycle, and the data at address 11 can be obtained in the fourth cycle. If the No. 10 data of the third cycle and the No. 11 data of the fourth cycle are ready, the pointer can smoothly move to the next position until a certain position returns "not fully ready". As shown in Table III, since the No. 14 data returned in the 7th clock cycle is not ready yet, an adventure failure occurs, and the pointer is no longer moved. Although the No. 15 data returned in the 8th cycle is ready, the pointer of ReorderRAM still stays at the No. 14 position, and the current returned No. 15 data is discarded. Then the request to read No. 14 data is re-initiated, followed by a re-initiated read of No. 15 data in the 9th cycle. By comparison, "simple read" successfully output 3 data in 9 clock cycles, while "adventurous read" successfully output 5 data, with a 67% increase in throughput. Ideally, "adventurous read" can output data every clock without wasting cycles.

F. Cache Module

The Cache module uses the packet header's 5-tuple value as the key and the matching rule ID as the value to construct a fixed-length hash table for lookup. The cache organization method used is "direct mapping". The packet first enters the cache module, which checks whether the packet hits the cache. If there is a cache hit, the system directly outputs the matching result, that is, the matching rule ID; if there is no cache hit, HACL adds a sequence number to the packet information to indicate the order of the packet. Then, multiple copies of *pkt* are sent to n PEs for parallel processing. The HACL cache replacement strategy is direct replacement, which means that when HACL provides a new lookup result, it would replace the original entry. Subsequent evaluations have shown that this approach achieves good results.

V. OVERVIEW OF THE COMPILER

The compiler acts as a conduit that seamlessly integrates decision tree algorithms with HACL hardware. It converts an input ruleset into a hardware-recognizable data structure file based on design requirement parameters. The block diagram of the compiler is illustrated in Fig. 11.

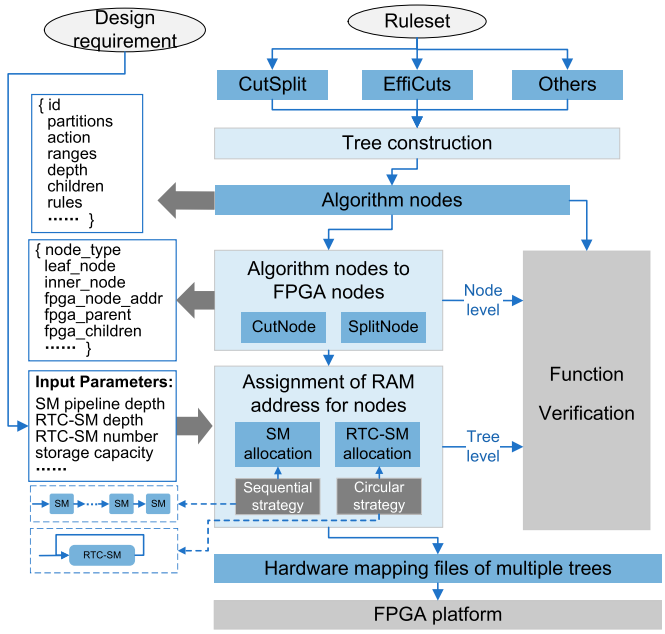


Fig. 11. Block diagram of the compiler.

A. Workflow of the Compiler

The following describes the working principle of the compiler in detail according to its operational sequence.

(1) Generation of Decision Tree Nodes: Based on a given set of rules, the compiler first builds decision trees based on a specific algorithm, and then generates the nodes of the algorithm. Various decision tree algorithms, such as HyperSplit [35], HyperCuts [23], CutSplit [20], ParaSplit [16], BitCuts [25], and ByteCuts [27] that use cut and split operations can be easily implemented on HACL's hardware engine.

(2) Conversion of Tree Nodes to Hardware Nodes: In this step, the nodes of the algorithmic decision trees are converted into an encapsulated form suitable for hardware implementation, where the parameter of *node_type* is used to distinguish between nodes adopting cut operation or ones using split operation. The hardware operation types are then configured according to the algorithm's settings. As can be seen from Fig. 11, there is a big difference in the representation of hardware nodes and algorithm nodes.

This conversion also includes assigning values to each hardware node structure such as *cmpDim* (comparison or cutting dimension), *cmpVal* (comparison value), *location* (cutting location), *bitmap* (indicating valid nodes), and *offset* (offset address relative to base address). The design of the *offset* and *isNext* parameters in the Stage Module (SM) allows binary trees, ternary trees, quad trees, and even octrees to be directly deployed in HACL without generating extra empty nodes.

(3) Assignment of Hardware Address for Each Node: This key step involves allocating addresses for all hardware nodes. Initially, the input constraints are determined based on the design requirements, including the depth of the SM pipeline, the address width of the RTC-SM, the number of RTC-SM, etc., and more importantly, the capacity limitation of the SM

corresponding to each layer. Additionally, considering the inherent topological relationship between different SMs, the compiler carefully calculates the storage address of each node in the hardware.

Specifically, due to the limited number of nodes that each SM level can accommodate in the search engine (pre-determined by FPGA resource configuration), the original data structure converted in the previous step cannot be directly mapped to the RAM of the hardware engine. In other words, multiple nodes from the same level of the decision tree may need to be mapped into different SMs due to capacity constraints. Furthermore, this mapping process must consider the parent-child relationship of the nodes: child nodes must be assigned to the SM or RTC-SM that comes after their parent nodes' SM or RTC-SM. Taking into account the capacity limitations and the aforementioned topological constraints, once the position of the parent node is determined, the set of available SMs for the child nodes is also determined.

The node mapping of RTC-SM can be regarded as ring-shaped SMs. When reaching the end of the RTC-SM pipeline and the mapping of the deepest node is not yet completed, the *ramID* of the address for the next mapped node will be the ID of the starting SM in the RTC-SM. It is important to note that HACL tends to allocate fewer decision tree layers to the RTC-SM, as an excessive number of layers in the RTC-SM will require multiple accesses during the packet lookup process, ultimately reducing overall performance. This process continues until the addresses of all hardware nodes have been calculated.

(4) Different Levels of Verification: In the previous conversion steps from algorithm nodes to FPGA nodes, upon completing the conversion of each node, a verification of the node's function is conducted, which involves comparing its search results with those of the algorithmic node. Similarly, in the step of hardware address allocation, upon completing the address allocation of all nodes of a complete pipeline, a decision tree function validation is also performed. Immediate verification at the node and tree levels effectively ensures the correctness of the compiler's functionality.

(5) Mapping to FPGA: Eventually, the node mappings of multiple trees are printed into multiple files, and then the file contents are transferred to the FPGA. HACL sets up a dedicated interface for node updates. The contents of the tree node mapping files are sequentially written into the SRAM of each SM and RTC-SM in HACL through the update interface. Therefore, updating or replacing the ruleset does not require regenerating the bitstream.

B. Address Coding and Assignment Strategy

Address Coding Mode: It is essential to note that the SM modules within an individual PE are structured to form a unidirectional cyclic graph. Supposing the parent node is allocated to the SM module labeled *id1*, and within HACL, the address for the child node linked to the same parent is denoted as *baseAddr+offset*. This ensures that all child nodes related to the same parent are located within a single SM module, and the corresponding SM module id is denoted as *id2*. Then starting from *id1*, *id2* must be reachable. Given the cyclic nature of the graph produced by the SM, trees of diverse depths can be mapped onto this graph.

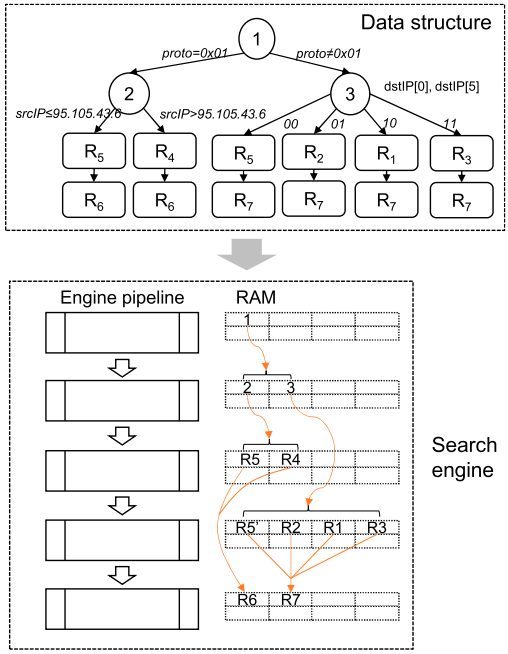


Fig. 12. The compilation illustration.

Greedy Algorithm for SM Selection: The compiler capitalizes on a greedy algorithm, opting selectively for the SM that is in closest proximity to the parent node. Utilizing a DFS (Depth First Search) traversal, the addresses for all nodes are determined sequentially by the compiler. To wrap it up, the compiler ingeniously amalgamates the decision tree algorithms with the HACL hardware, ensuring optimized performance and seamless integration.

C. An Compilation Example

According to the later evaluation, hardware resources are sufficient for a 100K ACL ruleset scenario. Here is an example of compiling the ruleset in Table I to help understand the HACL compilation process. The top half of Fig. 12 demonstrates the decision tree data structure generated by the algorithm for the ruleset. In the bottom half, a possible mapping result of the decision tree on the hardware search engine is displayed, with yellow arrows indicating the *node.nextAddr*. In the data structure, the children of nodes 2 and 3 are on the same level, but they belong to different levels in the search engine, which is an adjustment made by the compilation algorithm (nodes in the same level of the tree can be mapped to different levels of the SM pipeline). It is worth noting that the leftmost child nodes of nodes 2 and 3 both contain rule R5, but because they need to compare different subsequent rules: R6 and R7 respectively, R5 rule is duplicated and appears on the third and fourth levels of SM in the search engine with different *node.nextAddr*. For R6 and R7 rules, there are no other rules to match subsequently,

TABLE IV
RESOURCE CONSUMPTION OF SEARCH ENGINE

Module	Configuration		LUT (872K)	FF (1,743K)	BRAM (1,344)	URAM (640)	WNS (ns)	Max Frequency (MHz)
Stage module (SM)	Node address width	4	1905	3810	0	0	2.337	601.32
		5	1905	3810	0	0	2.183	550.36
		6	2096	3810	0	0	2.3	588.24
		7	1716	3050	5.5	0	2.017	504.29
		8	1716	3050	5.5	0	2.003	500.75
		9	1717	3050	5.5	0	1.972	493.10
		10	1717	3050	5.5	0	2.11	529.10
		11	1718	3050	0	3	1.324	373.69
		12	1718	3050	0	3	1.532	405.19
SM pipeline (address width: 1, 3, 5, 7, 9, 11, 12, 12, 12, 12, ...)	Pipeline depth	4	7222	14471	5.5	0	1.283	368.05
		6	10522	20565	11	3	1.059	340.02
		8	13825	26659	11	9	1.007	334.11
		10	17126	32753	11	15	0.711	304.04
		12	20425	38847	11	21	1.148	350.63
		14	23725	44941	11	27	0.523	287.60
		16	27025	51035	11	33	0.736	306.37
		18	30325	57129	11	39	0.551	289.94
		20	33627	63223	11	45	0.84	316.46
PE (SM pipeline + 3 RTC-SM, address width: 5, 5, 5, 10, 10, 12, 12,12, 12, ...)	SM pipeline depth	12	38074	67151	50.5	48	0.813	313.77
		14	41383	73265	50.5	54	0.516	287.03
		16	44588	79353	50.5	60	0.459	282.41
		18	47883	85455	50.5	66	0.402	277.93
		20	51187	91549	50.5	72	0.59	293.26
HACL (4 PEs + ReorderRAM)	SM pipeline depth (RTC-SM address width: 10)	10	138025	237692	416	60	0.253	266.88
		12	151315	262090	416	84	0.474	283.61
		14	164615	286475	416	108	0.389	276.93
		16	177682	310750	416	132	0.222	264.69
		18	190811	335203	416	156	0.438	280.74
		20	203989	359781	416	180	0.176	261.51
	SM pipeline depth (RTC-SM address width: 12)	10	138090	237715	218	168	0.314	271.30
		12	151369	262083	218	192	0.386	276.70
		14	164671	286400	218	216	0.454	282.01
		16	177767	310848	218	240	0.291	269.61
		18	190878	335184	218	264	0.3	270.27
		20	204060	359529	218	288	0.328	272.33

so they only appear on the last level of SM without being duplicated. This design of multiple parent nodes pointing to the same child node effectively reduces the storage resource consumption in the search engine.

VI. EVALUATION

We use System Verilog language to implement the search engine with approximately 2,700 lines of code. The compilation algorithm for mapping adjustments of decision trees is implemented by approximately 1,000 lines of C++ code. The hardware performance evaluation is conducted by using Xilinx's Alveo U50 data center acceleration card equipped with a 16nm UltraScale+ XCVU35P FPGA chip with 872K LUTs, 1,743K registers, as well as 1,344 36Kb BRAMs and 640 288Kb URAMs. Hardware resource evaluation is performed with Xilinx Vivado v2020.1. The ACL rulesets used in the evaluation are generated by ClassBench [36], and the network traffic used in the evaluation of the Cache module is the Internet flow collected by CAIDA from Equinix NYC in 2019 [37]. This traffic can simulate the flow situation of datacenter gateways.

A. Hardware Resource Envaluation

This section evaluates the detailed resource consumption of HACL after implementation. The layout and routing strategy used is "Performance ExtraTimingOpt". In order to evaluate HACL as a hardware IP, IO port binding is not performed. The clock cycle configuration is 4ns, and the duty cycle is 50%, where WNS (Worst Negative Slack) represents the worst negative timing margin, LUT is a logic 6-input lookup table, and FF is a register.

1) Implementation Results: HACL's architecture is highly parameterized. The parameters such as the SM pipeline depth, the RTC-SM address width, and the number of PEs are adjustable. They are defined in the configuration macros of the source code files, and users can modify the architecture just by changing the macro values. After modification, the bitstream of the corresponding architecture needs to be regenerated. We have separately evaluated the resource utilization and maximum performance achievable by SM, SM pipeline, PE, and the overall HACL search engine under different configurations. The results are shown in Table IV.

The complete search engine of HACL consists of 4 PE modules and a ReorderRAM module (4-input). Different levels of SM and a fixed set of 3 RTC-SM modules with two sets of address width have been implemented, allowing for the accommodation of rulesets of various scales. As the RAM of the SM is configured in dual-port mode, each SM is equipped with two packet processing channels. One channel only reads node information from the RAM, while another channel is responsible for writing information to RAM when updating, and reading node information from RAM when not updating to improve the total throughput.

In the pipeline composed of SMs, different SMs utilize a stepped distribution of address widths. This means that the address width of each SM increases progressively to adapt to

nodes in different levels. This stepped distribution effectively utilizes resources, enhancing the flexibility and performance of the system. The maximum operating frequency represents throughput, because the pipeline can run without pauses under the current configuration.

We calculate the capacity using the example of the HACL configuration with the minimum SM pipeline depth of 10. The address widths of all stages in the SM pipeline are: 5, 5, 5, 10, 10, 12, 12, 12, 12, 12, corresponding to storage capacities of: 32 (2^5), 32, 32, 1024 (2^{10}), 1024, 4096 (2^{12}), 4096, 4096, 4096, 4096. Adding them together gives us the corresponding capacity of the SM pipeline: 32*3+1024*2+4096*5=22.624K. In addition, each RTC-SM contains 3 SMs, with each stage having an address width of 10. Thus, the capacity of three RTC-SMs is: 1024*3*3=9.216K, and the total capacity of 4 PEs is: (22.624K+9.216K)*4 $\approx$ 127K. Similarly, in an HACL configuration with the maximum SM pipeline depth of 20 and the address width of 12 for RTC-SM, 4 PEs can contain (32*3+1024*2+4096*(15+3*3))*4 $\approx$ 401K nodes and rules. This means that, under various configurations, the search engine is capable of accommodating ACL rules with a scale of 100K while delivering throughput performance exceeding 260Mpps.

Moreover, the logical resource occupation of the maximal configuration is only $\frac{204,060}{872K} = 23.4\%$, the register occupation is 20.6%, the BRAM is 16.2%, and the URAM is 45%. The U50 board still reserves a large amount of space for other modules, and more HACL search engines can be further deployed to increase the rule capacity or classification throughput.

2) Storage Overhead: HACL, ParaSplit [16], TcbTree [19], and KickTree [31] can all map the data structure of multiple decision trees onto self-designed hardware (the latter three can only accommodate specific types of decision trees, and are less versatile than HACL), but their storage resource overhead for the node address encoding varies. The calculation method for obtaining the address of the next node to be accessed in ParaSplit is *node.nextAddr* + *cutBits*, which causes ParaSplit to still consume a node's storage space for an empty child node. TcbTree saves the exact address of each child node on the node, avoiding the storage occupied by empty nodes and allowing child nodes to appear in different RAMs. However, the fixed address width of TcbTree results in excessive storage overhead, as each address occupies 20bit, and the total overhead of the octree's child node addresses reaches 8*20b=160b. HACL utilizes the base address *node.nextAddr* plus the offset to determine the position of the child node. Since the length of the offset is much shorter than *nextAddr*, the storage overhead is smaller than that of TcbTree, and it also avoids the storage overhead of empty nodes.

Fig. 13(a) shows the storage overhead of hardware address encoding of HACL compared to ParaSplit and TcbTree/KickTree under multiple ACL rulesets. The storage overhead here refers to the pure storage consumption of all nodes and rules without considering hardware redundancy, in which rule nodes, cut nodes, and split nodes are calculated separately. This can intuitively reflect the advantages and disadvantages of different encoding and addressing schemes. The storage overhead of HACL on 1K, 10K, and 100K rulesets is 0.19 - 0.25Mb, 2.20 - 2.30Mb, and 14.1 - 20.5Mb, respectively. The storage overhead increases linearly with the number of rules.

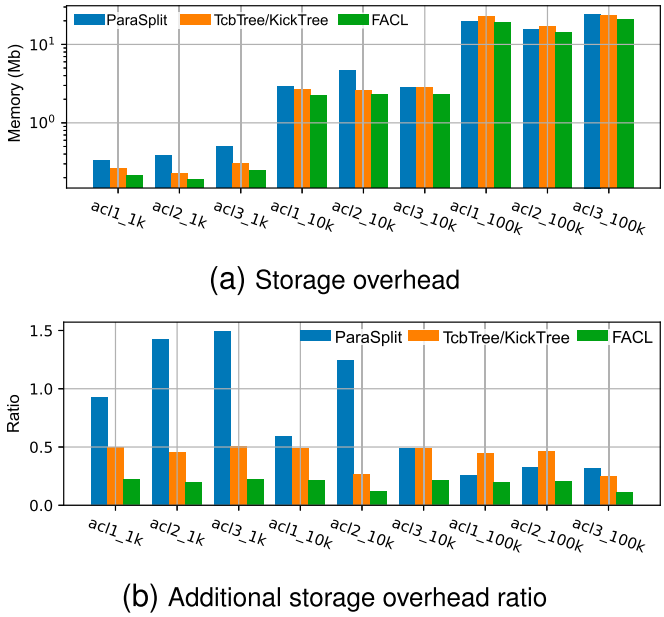


Fig. 13. The storage overhead of hardware address encoding of HACL compared to other designs.

TABLE V
THE RESOURCE CONSUMPTION OF REORDER MODULE

Module	Input num	LUT	FF	BRAM	WNS (ns)	Max Frequency (MHz)
ReorderReg (address width: 10)	2	20	20	2	2.564	696.38
	4	7042	4225	4	0.527	287.94
	8	11733	8348	8	0.625	296.30
ReorderRam (address width: 10)	2	115	63	4	1.961	490.44
	4	306	126	8	1.568	411.18
	8	572	222	16	1.549	408.00

Fig. 13(b) depicts the ratio of additional storage overhead compared to the original ruleset. The additional storage overhead refers to the address index information in the decision tree data structure (only intermediate nodes, excluding rules) for hardware. Compared to the storage size of the original ruleset, the additional storage overhead brought by HACL is around 20%, while ParaSplit has a maximum additional storage overhead ratio of 150%, and that for TcbTree and KickTree reaches 50%. HACL always has lower storage overhead than ParaSplit, TcbTree and KickTree, and on the 100K ruleset, HACL can save up to 4Mb of storage space compared to the latter three.

3) Reorder Module: It is mentioned in Section IV-E that the Reorder module can be realized by the existing PIFO design, but a single PIFO needs to consume 140K LUT resources when accommodating 1K units, and 512 units correspond to 70K LUT [38]. The hardware resource occupation of ReorderRAM and ReorderReg under 1K capacity with different input numbers is shown in Table V. The LUT resource consumption of ReorderReg with four inputs is only 5% of PIFO, and ReorderRAM's is only 0.22%. On the other hand, the LUT and Register consumption of ReorderRAM is only 4.3% and 3% of ReorderReg

TABLE VI
COMPARISON OF DECISION TREE BASED APPROACHES ON FPGA

Ruleset Scale	Approach	Device	Resource consumption				Classification Throughput (MPPS)	Adaptive
			LUT	Registers	BRAM	URAM		
100K	Proposed HACL	UltraScale+ XCVU35P	190811	335203	416	156	280.74	✓
	KickTree_Systolic [39]	Ultrascale+ VU9P	506670	550812	1836	936	121.40	×
	TcbTree [19]	Ultrascale+ VU9P	66834	111719	990	792	45.6	×
	KickTree_Parallel [31]	Ultrascale+ VU9P	607596	819039	1815	762	172.9	×
	MBitTree on FPGA [30]	Virtex-7 XC7V690T	37828	75656	818	0	175.9	×
10K	REC [29]	Virtex-5 XC5VFX200T	7044	\	173	0	323.5	×
		Virtex-6 XC6VLX760	7044	\	173	0	388.2	×
	UTPC [18]	Stratix III EP3SE260H780	40070	\	852	0	433	×
	D^{2}BS [40]	Virtex-5 XC5VSX240T	\	\	\	\	263.7	×
	Hypercuts on FPGA [41]	Virtex-5 XC5VFX200T	10307	\	407	0	250.7	×
	ParaSplit [16]	Virtex-5 XC5VSX240T	48380	\	399	0	200.4	×
	CubeCuts [28]	Virtex-5 XC5VFX200T	45656	\	195	0	368.8	×
	Hypersplit on FPGA [42]	Virtex-6 XC6VLX760	2988	5976	103	0	230.5	×

TABLE VII
STATISTICAL CHARACTERISTICS OF CAIDA DATA SET

No.	#Packets	#Flows	#Packets in the largest flow
1	2,620,909	176,545	17,053
2	2,517,685	176,266	16,499
3	2,530,158	177,734	15,974
4	2,518,856	175,956	13,092
5	2,530,680	175,806	15,594
6	2,517,752	174,663	16,463
7	2,515,396	173,449	16,765
8	2,501,781	173,972	16,448
9	2,493,844	172,686	17,329
10	2,465,020	173,883	17,491

(four inputs). Compared with ReorderReg, ReorderRAM has a higher operating frequency, but its BRAM consumption is doubled. Based on the above facts, it seems that ReorderRAM would be a more suitable choice as the out-of-order Reorder module for HACL. However, if there are limitations on the availability of BRAM resources, ReorderReg can be considered as an alternative option.

B. Performance Comparison

Table VI shows the comparison of performance and resource consumption between HACL and other decision tree based schemes. We choose the HACL configuration with SM pipeline depth of 18 for comparison, as this configuration can provide sufficient adaptability. Note that all performance data compared are from the cited works.

There are five designs capable of supporting 100K rules, in which HACL has achieved the best performance results. Compared with the up-to-date KickTree_Systolic design [39], the throughput of HACL is more than twice that of it. Since the latest approaches take advantage of FPGAs equipped with URAMs, they cannot be accommodated by the previously adopted platforms such as Virtex-5/6. Furthermore, many previous designs do not support large-scale rules and only adopt 10K ACL as the benchmark. Although their throughput can reach very high values, the low ruleset size limits their applicability. The most important thing is that among the comparison objects, only HACL is a versatile architecture that can provide adaptivity to support multiple algorithms and various decision tree sizes without the need of architecture regeneration.

C. Cost Performance Comparison

We also compare the software solutions in DCN and HACL from the perspective of cost performance. HP's cabinet server, the HPE ProLiant DL360 Gen10, with an Intel dual-socket Xeon Gold 6248/2.5 GHz (3.9 GHz), is priced at US $15,057, according to quotes from zones.com. The machine has a total of 40 cores and 80 threads. Even according to the single-thread optimal throughput of 2Mpps, the entire machine can only provide 160Mpps throughput, and the average cost of 1Mpps is US $94. The U50 network card used in the evaluation of HACL in this article is quoted at US $4,650 on this website, and the server that the network card is plugged into can be chosen as a model with mediocre performance. If the server is a single-socket Intel Xeon Gold 5218/2.3 GHz (3.9 GHz), the price is US $4,900. After comprehensive consideration, the 1Mpps of HACL costs US $38.2, which saves 60% of the cost. In addition, the server where the U50 is located and the remaining resources of the U50 can be used for other services of the gateway, thus the actual average cost will be lower.

D. Performance Improvements from Caching

This section evaluates the Cache module of HACL, which proves that the current design has a good performance improvement. Different from OvS's complex megaflow cache system [43], HACL's Cache module essentially performs the lookup and update process of the hash table.

The CAIDA traffic is divided into 10 groups in chronological order, and the duration of each group of traffic is 5-10s. Table VII shows some statistical characteristics of the 10 groups of traffic. It can be seen that the 10 groups of data are very similar in the number of flows and the number of packets of the largest flow. In other words, each test set can still well reflect the overall characteristics of CAIDA traffic, and will not affect the evaluation effect due to the shorter traffic data after segmentation.

Fig. 14 shows the box plot of the cache hit rate under different cache entry capacities. Each box reflects the maximum, minimum, and median hit rates of all traffic groups under a specific cache size. In general, the hit rate has increased from an average of 3.28% under 4 cache entries to 82.1% under 64K cache entries. The growth rate is gradually slowing down

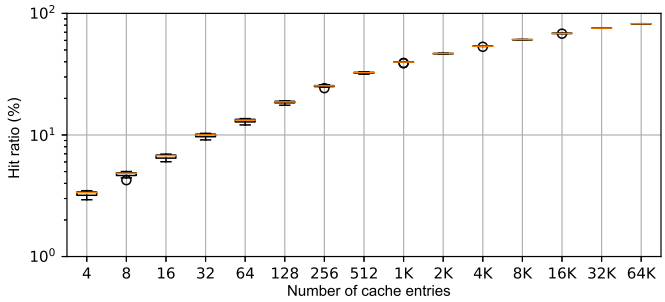


Fig. 14. HACL cache hit ratio.

(note that the horizontal and vertical coordinates of Fig. 14 are logarithmic scales). The mechanism behind this lies in the power-law distribution of traffic. A small number of large flows have more packets, and a certain cache can greatly reduce the pressure on the backend. We chose a cache entry size of 256 in the implementation as it serves as a reasonable trade-off point. First, it imposes relatively low resource consumption, and second, the growth rate in hit rate slows down as the number of entries continues to increase. When larger caches are used, the LUT and Register overhead does not increase much, but the RAM overhead increases exponentially.

It is worth noting that the Cache module can only reduce the pressure on the backend (some packets need to access RTC-SM multiple times), and improve the average throughput, but for instantaneous burst traffic, the cache may be completely invalid, and a large number of packets are queued for backend processing. Thus, it cannot be taken for granted that the hit rate of 80% can increase the throughput to $\frac{250Mpps}{1-80\%} = 1.22Gpps$. From the perspective of queuing theory, in order to ensure that the queue length is controlled within a certain range, the arrival rate of tasks must be less than the expected processing capacity.

Under 2K entries, the cache hit rate is about 50%, but if the CAIDA traffic is continuously injected at a rate of 375Mpps (1.5 times the processing capacity of the backend 250Mpps), the backend queue will exceed 3000 packets at most, resulting in FIFO overflow. If injected at a rate of 325Mpps (1.3 times the back-end processing capacity), the back-end queue length is at most 100 packets, resulting in an additional 400ns queue delay, which is equivalent to HACL's own 500ns processing delay. On the other hand, caching can only enhance performance in specific scenarios and is suitable for some scenarios with elephant flows.

VII. Conclusion

In this paper, we design and implement a heterogeneous and adaptive architecture for fast ACL engine on FPGA-based SmartNIC, based on decision tree algorithm. In the aspect of performance, the ACL engine can achieve at least 260Mpps throughput for 100K-scale ACL rulesets. In the aspect of adaptivity, it supports the arbitrary ruleset update because it could allow any depth of decision trees with a small sacrifice of average throughput. With extensive experimental evaluations, we demonstrate the feasibility of our FPGA architecture and the compilation algorithm.

References

[1] B. Tian et al, "Safely and automatically updating in-network ACL configurations with intent language," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 214–226.
[2] D. Firestone et al, "Azure accelerated networking: Smartnics in the public cloud," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 51–66.
[3] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren, "SmartNIC performance isolation with fairnic: Programmable networking for the cloud," in *Proc. ACM Special Interest Group Data Commun.*, 2020, pp. 681–693.
[4] W. Li, T. Yang, O. Rottenstreich, X. Li, G. Xie, and 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.
[5] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "Scaling open vSwitch with a computational cache," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 1359–1374.
[6] "ConnectX NICs," Nvidia, 2024. Accessed: 2024. [Online]. Available: https://www.nvidia.com/en-us/networking/ethernet-adapters/
[7] G. P. Katsikas, T. Barbette, M. Chiesa, D. Kostić, and G. Q. Maguire Jr, "What you need to know about (smart) network interface cards," in *Proc. Conf. Passive Act. Netw. Meas.*. Springer, 2021, pp. 319–336.
[8] "Mellanox Adapters Programmer's Reference Manual (PRM)," Mellanox. 2024. Accessed: 2024. [Online]. Available: https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf
[9] H. J. Chao and B. Liu, *High Performance Switches and Routers*. Hoboken, NJ, USA: Wiley, 2007.
[10] G. Varghese and J. Xu, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Mateo, CA, USA: Morgan Kaufmann, 2022.
[11] C. R. Meiners, A. X. Liu, and E. Torng, *Hardware Based Packet Classification for High Speed Internet Routers*. Springer, 2010.
[12] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartNICs using iPipe," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 318–333.
[13] 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.
[14] 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.
[15] C.-H. Chou, F. Pong, and N.-F. Tzeng, "Speedy FPGA-based packet classifiers with low on-chip memory requirements," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2012, pp. 11–20.
[16] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "ParaSplit: A scalable architecture on FPGA for terabit packet classification," in *Proc. IEEE Annu. Symp. High-Perform. Interconnects*, 2012, pp. 1–8.
[17] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2009, pp. 219–228.
[18] A. Kennedy and X. Wang, "Ultra-high throughput low-power packet classification," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 22, no. 2, pp. 286–299, Feb. 2014.
[19] 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.
[20] 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.*, 2018, pp. 2645–2653.
[21] O. Rottenstreich and J. Tapolcai, "Lossy compression of packet classifiers," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 2645–2653.
[22] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Jan./Feb. 2000.
[23] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM Special Interest Group Data Commun.*, 2003, pp. 213–224.
[24] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *Proc. ACM Special Interest Group Data Commun.*, 2010, pp. 207–218.
[25] Z. Liu, S. Sun, H. Zhu, J. Gao, and J. Li, "BitCuts: A fast packet classification algorithm using bit-level cutting," *Comput. Commun.*, vol. 109, pp. 38–52, 2017.

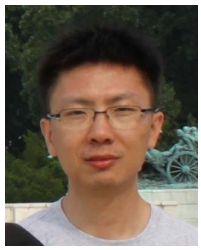
[26] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *Proc. IEEE Int. Conf. Netw. Protocols*, 2014, pp. 308–319.
[27] J. Daly and E. Torng, "Bytecuts: Fast packet classification by interior bit extraction," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 2654–2662.
[28] Y. Chang and Y. Wang, "CubeCuts: A novel cutting scheme for packet classification," in *Proc. Int. Conf. Adv. Inf. Netw. Appl. Workshops*, 2012, pp. 274–279.
[29] Y. Chang, H. 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, 2019.
[30] J. Tan, G. Lv, Y. Ma, and G. Qiao, "High-performance pipeline architecture for packet classification accelerator in DPU," in *Proc. Int. Conf. Field-Programmable Technol.*, 2021, pp. 1–4.
[31] 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./Apr. 2023.
[32] J. Tan, G. Lv, and G. Qiao, "MBitTree: A fast and scalable packet classification for software switches," in *Proc. IEEE Symp. High-Perform. Interconnects*, 2021, pp. 60–67.
[33] C. Jia, C. Li, Y. Li, X. Hu, and J. Li, "FACL: A flexible and high-performance ACL engine on FPGA-based SmartNIC," in *Proc. IFIP Netw. Conf.*, 2022, pp. 1–9.
[34] A. Sivaraman et al., "Programmable packet scheduling at line rate," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 44–57.
[35] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *Proc. IEEE Conf. Comput. Commun.*, 2009, pp. 648–656.
[36] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.
[37] T. C. for Applied Internet Data Analysis (CAIDA), "CAIDA traces," 2019. Accessed: 2024. [Online]. Available: http://www.caida.org/data/overview/
[38] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 367–379.
[39] Y. Xin et al., "Recursive multi-tree construction with efficient rule sifting for packet classification on FPGA," *IEEE/ACM Trans. Netw.*, vol. 32, no. 2, pp. 1707–1722, Apr. 2024.
[40] 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.
[41] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 20, no. 9, pp. 1668–1680, Sep. 2012.
[42] 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. IEEE Int. Conf. Field-Programmable Technol.*, 2010, pp. 241–248.
[43] W. Tu, Y. Wei, G. Antichi, and B. Pfaff, "Revisiting the open vSwitch dataplane ten years later," in *Proc. ACM SIGCOMM Conf.*, 2021, pp. 245–257.

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. Currently, he is an Associate Professor with the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangdong, China. His research interests include network intelligent hardware acceleration, VLSI design for deep learning, and network algorithms.

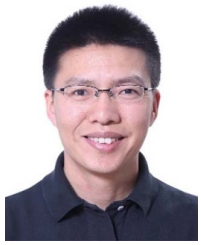
Chengjun Jia received the Ph.D. degree from the Department of Automation, Tsinghua University, Beijing, China, in 2023. Currently, he is working with Huawei Technologies Company, Ltd., Beijing, China. His research interests include parallel computer architecture, data center network, and hardware acceleration. He has published some papers on packet classification, congestion control, and network verification.

Wenjun Li received the Ph.D. degree from Peking University, in 2020. From 2020 to 2023, he was a Postdoctoral Fellow with Peng Cheng Laboratory, Harvard University. From 2014 to 2015, he was a Research Engineer with Huawei Technologies Company, Ltd. Currently, he is an Associate Researcher with Peng Cheng Laboratory. His research interests include programmable network data plane, network telemetry, and network algorithms.

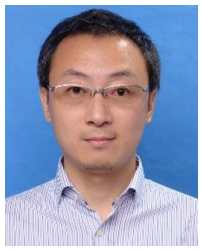
Ori Rottenstreich received the B.Sc. degree in computer engineering and the Ph.D. degree from Technion, Haifa, Israel, in 2008 and 2014, respectively, where he is an Associate Professor with the Department of Computer Science and the Department of Electrical and Computer Engineering. From 2015 to 2017, he was a Postdoctoral Research Fellow with Princeton University.



Yang Xu (Member, IEEE) received the Ph.D. degree from Tsinghua University, in 2007. He is the Yaoshihua Chair Professor with the School of Computer Science, Fudan University. He was a Faculty Member with New York University Tandon School of Engineering. His research interests include SDN, DCN, NFV, and edge computing. He has published more than 120 papers and holds more than ten U.S. and international granted patents on various aspects of networking and computing.



Gaogang Xie received the Ph.D. degree in computer science from Hunan University, in 2002. Currently, he is a Professor with the Computer Network Information Center, Chinese Academy of Sciences, and University of Chinese Academy of Sciences. His research interests include internet architecture, packet processing and forwarding, and internet measurement.



Zhihong Tian (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees from Harbin Institute of Technology, Harbin, China. He is a Professor, and the Dean, with the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangdong, China. He is also a part-time Professor with Carlton University, Canada. He has authored over 200 journal and conference papers. His research interests include computer networks and cyberspace security. He is a Distinguished Member of the China Computer Federation.

Jun Li received the B.S. and M.S. degrees from the Department of Automation, Tsinghua University, Beijing, China, in 1985 and 1988, respectively, and the Ph.D. degree from the Department of Computer Science, New Jersey Institute of Technology, in 1997. Currently, he is a Professor with the Department of Automation, Tsinghua University. His research interests include network security and network automation.