FACL: A Flexible and High-Performance ACL engine on FPGA-based SmartNIC

Chengjun Jia, Chenglong Li*, Yifan Li, Xiaohe Hu, Jun Li
Tsinghua University, Beijing, China
{jcj18, liyifan18}@mails.tsinghua.edu.cn, {lichenglong, hxhe, junl}@tsinghua.edu.cn

***Abstract*—Access Control List (ACL) is an important network function in modern cloud and carrier networks. Nowadays, SmartNIC is becoming a promising location to perform network functions in the end-to-end transmission. However, previous ACL designs have difficulties to achieve high throughput and support various kinds of rulesets at the same time. FPGA-based SmartNIC brings a new opportunity due to its flexibility and parallelism. In this paper, we propose FACL, a flexible and high performance ACL engine with the decision tree approach on FPGA-based SmartNIC. With the tree decomposition and the Network-on-Chip (NOC) pipeline scheduling, it is feasible for FACL to support all kinds of rulesets, as long as there is sufficient memory space. A compiler for FACL is also proposed, which maps decision trees to SRAM memory to optimize the throughput of a ruleset. FACL is implemented on Xilinx U250, a typical FPGA SmartNIC. According to the evaluation, FACL achieves up to 250 Mpps throughput with about 150 ns latency, when dealing with various 100 K ACL rulesets. The utilization of LUT/Register is only 10%/3.8%. With further decision tree optimization and engine parallelism, FACL has the potential to achieve higher throughput and support larger rulesets.**

***Index Terms*—SmartNIC, Packet Classification, FPGA, Parallel Processing**

I. Introduction

In modern networks, Access Control List (ACL) is an important pillar to support network security and guarantee performance isolation [34]. A typical ACL ruleset is shown in Table I. For each packet, the classifier engine searches on the ruleset and outputs the action of the matched highest-priority rule for the subsequent processing. With the trends of software-defined networking and programmable networking, SmartNIC [20] is becoming a promising unit to perform network functions between the server CPU and the switch. Cloud networks and 5G networks are deploying the SmartNICs for better performance [6] [40].

There are four main requirements for the high performance SmartNIC ACL engine: 1) *High capacity for rules*. There would be more than 100K rules to match considering the thousands of tenants in a cloud data center or the hundred thousands of user equipment in a carrier network. But the best Intrusion Detection System (IDS) product in Juniper only supports about 80K security policies [13]. 2) *Support for online rule update*. Due to the software-defined architecture in cloud and carrier networks, the ruleset is changing by programs instead of manual operations, and the rule update should take effect in sub-seconds. Therefore, the ACL engine must support incremental updates and ensure consistency during the update period[1]. 3) *High throughput for traffic*. The 25GbE NICs have been widely deployed in modern data centers, the share of the 100GbE NICs is expanding, and the 400GbE NIC has also been released [24]. To handle such a high load is a big challenge for the ACL engine because the low throughput would cause packet loss, severely degrading the performance of the upper-level applications. 4) *Low latency for classification*. The ACL engine is one step of the end-to-end network processing. Any delay caused by ACL matching would directly increase the completion time for the service.

TABLE I: Example ACL Rules

id	$addr_{src}$	$addr_{dst}$	$port_{src}$	$port_{dst}$	*action*
R_1	11****	*	80-MAX	*	a_1
R_2	001***	0010**	*	0-80	a_2
R_3	001***	0110**	*	80-150	a_3
R_4	0011**	0010**	*	200-400	a_4
R_5	0010**	0110**	*	400-MAX	a_3
R_6	01****	1101**	*	*	a_4
R_7	110***	1001**	*	*	a_4
R_8	100***	1011**	*	*	a_4

There are three main kinds of hardware compositions for all SmartNICs: specific ASICs, multi-core processors, and FPGA [21]. TCAM, a traditional ASIC solution on switches for ACL, is not suitable for the scenario. It achieves high throughput (˜100 Mpps) and ultra-low latency (˜10 ns), at the expense of limited capacity, high chip area, and high power consumption. Multi-core processors, such as the architecture of MIPS [9] or ARM [24], can update the ruleset and expand the capacity (100K+ rules) with the software library of DPDK or Hyperscan, but it costs too many CPU cores to handle 100 Gbps traffic, more than 30 cores even with the state-of-art algorithm [17], and DRAM bandwidth limits the throughput under large rulesets [28].

FPGA-based SmartNIC provides a good opportunity due to its flexibility and parallelism. There have been several FPGA designs for ACL classification; but previous BV-based [8]

This work is supported by National Natural Science Foundation (No. 61872212) and Industry-university-research Innovation Fund for Chinese Universities (No. 2021FNA04002).
*Chenglong Li is the corresponding author.

ISBN 978-3-903176-48-5

[1]The consistency means that the classification result of any packet is either from the ruleset before the update or from the updated ruleset.

TABLE II: ACL engine on FPGA

Architecture	*Rule*		*Classification*	
	Capacity	***Update***	***Throughput***	***Latency***
Fixed pipeline [7] [12]	Low	Hard	**High**	**Low**
Non pipeline [15] [39]	**High**	**Easily**	Low	Uncertain
FACL (flexible pipeline)	**High**	**Easily**	**High**	**Low**

[27] or hash-based [3] solutions only support ˜1K rules and need to reset the circuit for the update. Decision-tree-based algorithms are quite suitable for the scenario: large ruleset and high performance. Previous FPGA decision-tree designs can be divided into two categories: fixed-pipeline [7] [12] and non-pipeline [15] [39]. On the one hand, fixed-pipeline designs can not work if the number of nodes at a certain level exceeds the preallocation, which makes the rule update unavailable. On the other hand, non-pipeline would decrease the throughput sharply when the depth of trees is high. Essentially, to realize a high-performance FPGA decision-tree engine, we need to tackle the following challenges:

- *Structure volatility*. No decision tree algorithm can guarantee the limitation of the tree depth given an arbitrary ruleset. It makes the stage number of the pipeline design difficult.
- *Space consumption*. The calculated classifier is usually not a complete tree. Arranging the tree's nodes level by level in FPGA storage would waste space. It makes the data structure mapping process difficult.

In this paper, we propose a **F**lexible and high-performance ACL engine on **F**PGA-based SmartNIC, FACL, using the decision tree approach. By *flexible*, we mean that an arbitrary decision tree algorithm can be mapped into the FACL architecture, and an arbitrary ruleset is feasible to be implemented in FACL, as long as there is sufficient memory space. By *high performance*, we make a comparison of FACL and existing alternatives in Table II. FACL could achieve high rule capacity and high packet throughput simultaneously.

The main idea of FACL is decoupling the decision tree structure into sub-modules (pipeline or parallelism) and leveraging a Network-on-Chip (NOC) to flexibly schedule the classification process among the sub-modules. Therefore, FACL enables the recirculation of a pipeline to adapt to the variation of decision tree depth. Moreover, a compiler is designed to map the decision tree into sub-modules, and a greedy algorithm is presented to optimize the space consumption. The following summarizes our main contributions:

- We propose a new framework FACL with tree decomposition and NOC pipeline scheduling, enabling depth volatility of decision trees. It makes full use of the FPGA mechanism, that the BRAM/URAMs are distributed, hence FACL utilizes multiple memory interfaces to enlarge the memory access bandwidth.
- We develop a compiler for FACL that maps the decision trees into the different pipes and maps different linear search rule items to the parallel memory to save space.

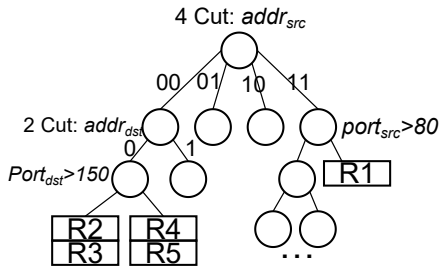


Fig. 1: Part of an example decision tree for Table I

- We implement FACL on a typical SmartNIC, Xilinx U250, and the evaluation demonstrates that the resource requirement of FACL is pretty low, and FACL achieves up to 250 Mpps throughput with various 100K rulesets.

The paper is organized as follows. We briefly introduce the decision-tree algorithm in §II. Then we elaborate on the design of FACL including FPGA architecture and rule compiler in §III. We illustrate some evaluation results to verify the feasibility and high performance of FACL in §V.

II. Background

For the decision tree classification algorithm, a corresponding typical decision tree is depicted in Fig.1 for the ACL ruleset in Table I. When the classifier wants to lookup the matched result of a packet, it would read the required field of the packet and decide where to go for the next node until it arrives at the leaf node. Then the classifier would search linearly against the leaf rules. For example, if a packet is with the header (001001, 001010, 20, 40), the classifier would read the first 2 bits of $addr_{src}$ according to the requirement of the root node. As it is '00', the classifier would go to the far left and then read the first bit of $addr_{dst}$ to judge the next direction. Finally, it compares the packet with $\{R_2, R_3\}$ to get the matched result with the highest priority, R_2.

Different algorithms would build different trees based on their heuristic observations and the operations at the nodes could be various. For example, HiCuts [10] chooses several continuous bits of one selected dimension to *cut* the packet header; HyperCuts [30] and EffiCuts [35] use the combination of a few continuous bits from several dimensions; while BitCuts [22] picks several discrete bits. HyperSplit [7] and SmartSplit [11] compare the field value with a specific value to *split* to reduce the size of the decision tree. ByteCuts [5] and CutSplit [16] allow the operations of *cut* or *split* in one node to expand the decision-making space further and build better trees with a lower depth.

As shown in Table III, even for the rule sets with the same number of rules, the decision trees built by the same algorithm differ largely. The HyperCuts trees for 'ACL 1(1K)' have about 25% more inner nodes than 'ACL 2(1K)' while the tree depth of the latter is more than 40% larger than the former. The diversity also exists for HiCuts, CutSplit, and other algorithms.

The uncertainty of the depth and the diversity for the node number in the tree are not a problem when we run it on

TABLE III: The volatility of decision trees

Ruleset (#rules)	Algorithm	#Trees	#Inner nodes	#Leaf nodes	Max tree depth	Trees' depth summary
ACL1 (1K)	HiCuts	1	659	3719	10	10
ACL1 (1K)	HyperCuts	2	446	3256	9	16
ACL1 (1K)	CutSplit	3	285	393	9	20
ACL2 (1K)	HiCuts	1	890	3107	14	14
ACL2 (1K)	HyperCuts	2	360	2075	13	21
ACL2 (1K)	CutSplit	3	290	419	13	29
ACL1 (10K)	CutSplit	3	1925	4271	11	25
ACL2 (10K)	CutSplit	3	2532	3243	14	36
ACL3 (10K)	CutSplit	3	3529	3936	15	40
ACL1 (100K)	CutSplit	3	17882	30379	14	34

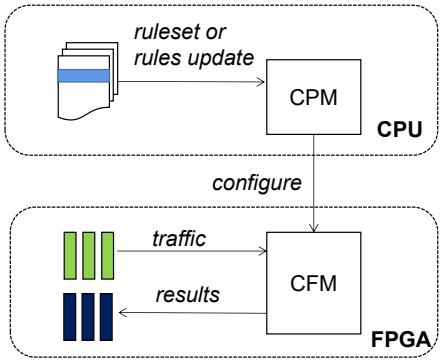


Fig. 2: The architecture of FACL.

the CPU-DRAM environment. We can dynamically allocate the memory for different levels of a tree. However, if we want to make full use of the parallel and pipeline processing features in the FPGA, diversity becomes a big problem. For example, ParaSplit [7] is a fixed-pipeline design on FPGA for HyperSplit; if the compiled tree depth is larger than preallocation, ParaSplit has to reconfigure the FPGA, which makes the ruleset update unavailable in this scenario. UTPC [15] is a non-pipelined design with all nodes of different levels stored in the same SRAM, but the throughput is limited as $\frac{B}{D}$PPS where B is the SRAM frequency and D is the tree depth, completely abandoning the FPGA advantages.

Our design FACL is aimed at mapping the decision trees to one fixed pre-configured FPGA circuit. For a specific decision-tree algorithm, the structures of multiple trees are different for different rulesets. They have diverse depths and various nodes at each level, but they all can be mapped in FACL.

III. System Design

As depicted in Fig.2, there are two modules for FACL: a classification module (CFM) and a compiler module (CPM).

- CFM works for the classification, with the input of a packet header and the output of matched rule ID as the result, which is the core module of FACL. CFM runs on the FPGA to provide high throughput.
- CPM is responsible for the rule compilation, with the input of original ruleset or update operations, and the output of memory (Register/BRAM/URAM) modification in CFM. CPM includes two parts: one is the previous algorithms to build the decision trees and the other is the mapping from trees to the CFM. CPM runs on the CPU because of the complexity; the CPU could be the host CPU or the SmartNIC CPU.

Because CFM is much complex, we give an overview in §III-A first and introduce the CFM modules respectively in §III-B,§III-C, and §III-D. CPM is shown in §III-E and some discussion about possible improvement for FACL is in §III-F.

A. CFM Overview

The FPGA architecture of CFM is depicted in Fig.3. There are three engines: Decision Tree Engine (DT), Linear Search Engine (LS), and Reorder Engine. These engines are connected by an on-chip interconnect network to switch the packet headers and calculation results.

When a new packet header is into CFM (Step ①), it first goes through the top trees in parallel to get the next addresses for classification (Step ②). Then the tuple of (*seq*, *header*, *nxtAddr*) would be switched to the corresponding DT/LS according to the value of *nxtAddr*. DT would modify the value of *nxtAddr* and resend the tuple to switch (Step ③-④). A tuple could go through different DTs by sequence, or pass through one DT several times to adapt to the depth of the corresponding leaf in the tree. Finally, the tuple arrives at the leaf node for linear search (Step ⑤-⑥). After that, the matching result would be led to Reorder Engine (Step ⑦). Due to the indeterminate depths of the leaf and unsteady latency from the switching, the order of results (⑦) could be different from the original header order (①). To guarantee the FIFO property, there is a buffer to reorder the sequence of results and aggregate the results from multiple trees to get the final one, i.e. the matched rule ID with the highest priority or unmatch notification (Step ⑧).

With the help of the NoC and Reorder Module, CFM decouples the structures of trees from the circuits of FPGA, by scheduling the packets freely without worrying about the out-of-order problem. If the depth of a leaf in the tree is larger than the preconfigured DT pipes, FACL could guide the corresponding packets to pass through another DT to extend the length of pipes, while other packets do not need.

B. Pipelined Decision Tree Engine

The architecture of DT is depicted in Fig.4, which is similar to the RMT [14] architecture.

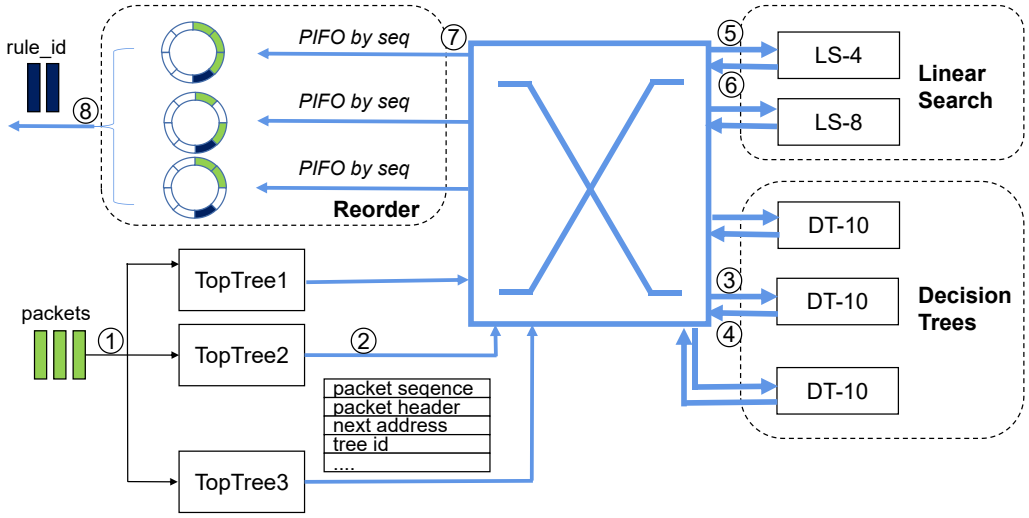


Fig. 3: CFM: the FPGA architecture of the packet classification. LS-**m**: Linear Search Engine with **m** rules per bucket which are compared with the input packet in parallel; DT-**n**: Decision Tree Engine with **n** stages.

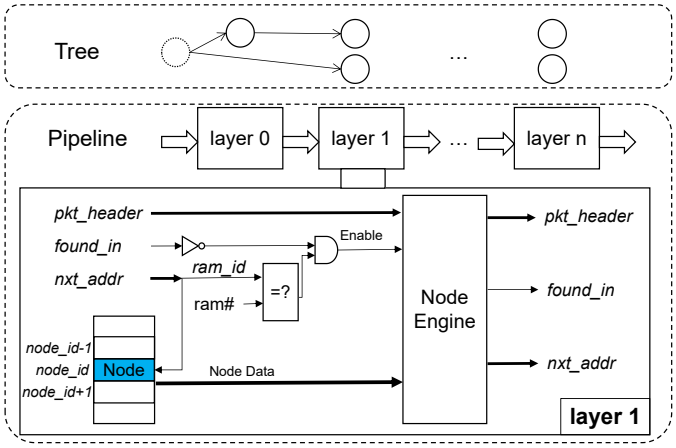


Fig. 4: The architecture of Decision Tree Engine.

A decision tree can be pipelined in the hardware for high throughput easily, with each level in the tree mapped as a pipe. As depicted in [7] [25], the first level of the tree is mapped as the first layer, and the second, the third until the last. However, the flexible scheduling as shown in §III-A brings the problem that the mismatching between the levels in the tree and the layers in the FPGA pipeline.

To solve the problem, every layer in all DTs has a unique ID $ram\#$ to judge whether to process the packet. For any layer in Fig.4, a tuple (*pkt_header*, *found_in*, *nxt_addr*) is transferred. The signal of *found_in* indicates whether the packet has arrived at the leaf for the corresponding decision tree. The *nxt_addr* consists of *ram_id*, which indicts the next layer, and *node_id*, which indicts the ram address of the next node. If the input $node\_id_1$ equals the layer ID $ram\#$, the node with an address of $node\_id_1$ would be read and use the node content to calculate the next address ($node\_id_2$ and ram_id_2); if not, the tuple (*pkt_header*, *found_in*, *nxt_addr*) would be kept for the next layer, of which the layer acts as a *nop* instruction. On the whole, all layers of DT together constitute a flexible pipeline to guarantee. A packet can be processed in the sequence of (layer 0, layer 1, layer 3, ..., layer n), or (layer 0, layer 2, layer3, ..., layer n) by skipping a certain layer; thus the nodes of the same level in the tree could be placed in the different layers of DT. As depicted in Fig.4, level-0 nodes in the tree are placed in both layer-0 and layer-1, and layer-1 in DT containers level-0 nodes and level-1 nodes. As a consequence, the preallocation SRAM on different layers/pipes could be used flexibly, without the limitations that the preallocated layer-*i* memory must be larger than level-*i* of the tree.

The node engine gets the next address from the input packet header and the node content. For the node, the meaning of each bit and supported operations can be encoded diversely for different decision tree algorithms, as the CPU instructions in RISC-V [38]. Users could modify the node engine without adjusting the other structure.

Because CutSplit has the most complex operations (*cut* and *split*) in the existing algorithms, we take CutSplit for IPv4 header as an example here and show the encoding of a node in Table IV. Each node is 4+32+32=68 bits and the beginning 4 bits (noted as OP) are used for the operation encode. CutSplit needs the support of *Cut* in src/dst IP and *Split* in all packet header fields. The operands in cut operation (StartInd, EndInd) and split operation (Value) are multiplexed to save space. Other algorithms could have different encoding methods. ByteCuts only allows 4/8/12 continuous bits *Cut* and the encoding can be easier without only 5 OPs. BitCuts would need a longer node because it uses arbitrary discrete bits and the information of the selected location must be stored in the node.

There are 4-bits of OP in our CutSplit node, allowable for 14 operations at most. All-zero value is reserved as invalid and the output is the same as the input, which is designed for the update process. All-one value is for the leaf node,

TABLE IV: Encoding of the node for CutSplit

Bit Fields	*OP*	Addr	Value
Bit Number	4	32	StartInd(5) EndInd(5) Other(22)

OP	Operations	Calculation
4'b0000	Invalid	nxt=node_addr
4'b0001	src IP (split)	nxt=Addr+[(Value $\geq$ srcIP)?1:0]
4'b0010	dst IP (split)	nxt=Addr+[(Value $\geq$ dstIP)?1:0]
4'b0011	src port (split)	nxt=Addr+[(Value $\geq$ srcPort)?1:0]
4'b0100	dst port (split)	nxt=Addr+[(Value $\geq$ dstPort)?1:0]
4'b1000	src IP (cut)	nxt=Addr+srcIP[StartInd:EndInd]
4'b1001	dst IP (cut)	nxt=Addr+dstIP[StartInd:EndInd]
4'b1111	leaf node	nxt=Addr, found_in=1

which cooperates with *found_in* signal to skip linear search procedure, as discussed in §III-F3. If there are more fields in the packet header, e.g. IPv6/OpenFlow [23] protocol, we can use more bits as the OP field to accommodate. It is worth noting that the bit length of one node had better not exceed the max width of a URAM, and it is 72bits in the case of Xilinx FPGA.

C. Parallel Linear Search Engine

Linear search can be implemented in FPGA with parallel comparison, as depicted in Fig.5. For each matching engine (LSE; the S_0, S_1, ...S_m in Fig.5), it would read one rule from its RAM and check whether the packet header is matched with the rule to output the stored rule ID, or a MAX_VALUE to indicate the mismatch. The aggregation of multiple LSE is completed by getting the minimum value of rule IDs (the highest priority ones) and if the final result is still the MAX_VALUE, it means that no rule matches the input packet.

There is a problem with which address is read for each LSE. The numbers of rules in different leaf nodes are not the same, so if we use the same node address for all engines, there would be much idle memory for LS. The waste does not exist for the previous designs [32], because they could carefully allocate the memory for a specified ruleset by reconfiguring FPGA circuits. FACL does not want to do so to accommodate the volatility. An easy way to avoid the memory waste in LS is to use different node addresses for different engines, but it would lead to the bit length of address from $log_2(K)$ to $m * log_2(K)$ where K is the rule number in one LSE. FACL solves the problem by adding the input of ram_id. Similar to DT in §III-B, each LSE has an id ram# too. The actual rule address is calculated by comparing the input ram_id with LSE $ram\#$ to choose between node_id and (node_id+1); thus the rules can be placed closely without waste of memory and the bit length of the address is only $log_2(K) + log_2(m)$.

For example, if a packet *pkt-1* is to search in R_1, R_2 at one leaf; and the other *pkt-2* in R_3, R_4, R_5 for LS-4, the align placement for LS-4 is as in Fig.6a: $\{R_1, R_3\}$ in S0, $\{R_2, R_4\}$ in S1, $\{R_5\}$ in S2. All LSE read from the same address *node_id*: 1 for pkt-1 and 2 for pkt-2. In contrast, different LSE could read from different addresses in our design. As in Fig.6b, we place $\{R_1, R_3\}$ in S0, $\{R_2, R_4\}$ in S1, $\{R_5\}$ in S2, $\{R_4\}$ in S3. For *pkt-1*, $node\_id$=1 and ram_id=1; for

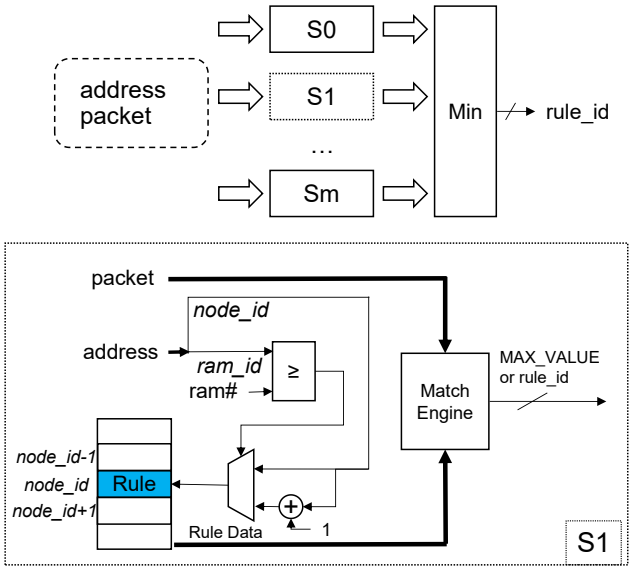
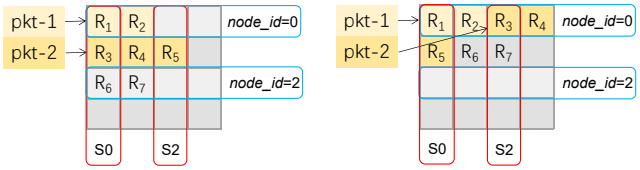


Fig. 5: The architecture of Linear Search Engine.



(a) Align (b) Our Design

Fig. 6: The placement of rule items in LS.

pkt-2, $node\_id$=1 and ram_id=3. Thus, *pkt-1* would search in $\{R_1, R_2, R_3, R_4\}$; and *pkt-2* would search in $\{R_3, R_4, R_5, R_6\}$. Other rules are added to the potential set but the false positive can be solved by the check because *pkt-1* does not match R_3 or R_4 by the check of LSE, and correctness is thus guaranteed. As a result, the align method needs 3*4=12 allocated slots with 5 unused; our design only needs 2*4=8 slots with only 1 unused. Compared with the align method, our design can save (1-8/12)=33% memory in this scenario because the rule items are closely packed.

D. Reorder Engine

As depicted in Fig.7, a reorder engine uses a variant of the PIFO (Push-In-First-Out) [31] [29] to reorder the matched rule results. A match result would carry the ID of the tree and the *seq* of the packet, which are inserted into the corresponding location. When match results from all decision trees are ready, the final result is got and the pointer moves to the next. Reorder Engine can guarantee that the result outputs per cycle.

There are finite packets buffered in FACL, thus the sequence is limited and the length of the SRAM array would not be large. There is a problem with the delay between address to the array and result back, which may raise the read/write consistency concern of *ind*. It can be solved with a go-back-N method to guarantee correctness.

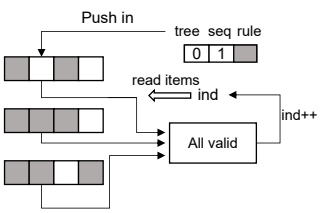


Fig. 7: Reorder Engine.

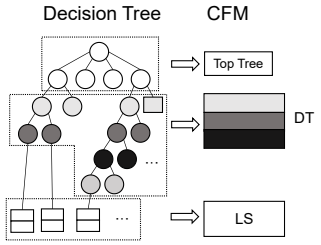


Fig. 8: CPM.

E. Compiler Module

CPM maps from the original decision trees to the FPGA memories of FACL. As depicted in Fig.8, the first few levels of a decision tree would be mapped to the *Top Tree* because it always is a full tree; the following inner nodes are mapped to the *DT* and the linear search of the leaves to the *LS*.

1) DT Mapping: There are three restrictions for the mapping from a tree to a pipeline in FACL.

- The siblings in a tree must be placed in the same layer and their addresses must be adjacent. It is because the calculation of the next address is from a base address and an offset, as shown in Table IV.
- The recirculation should be prevented. If node A is the posterity of node B and A.layer is larger than B.layer in one DT module, the recirculation would happen.
- If the height of a tree is larger than the layer number of the DT, the tree must be split into several subtrees to place each in the DT.

To meet these requirements, we show a greedy DT mapping in Algorithm 1. The main idea behind the algorithm is a Depth-First-Search (DFS). We first try to find the deepest leaf node (Line 15) and get all its ancestors which have not been mapped (Line 16-19). Then we try to place the ancestors and their siblings at the same layer (Line 22-25) in the ascending order of node depth. The failure and corner cases are carefully handled in our algorithm.

2) LS Mapping: As discussed in §III-C, if the total number of rule items is not larger than LS capacity, these items can be placed tightly in arbitrary order and the utilization of LS memory is 100% as depicted in Fig.6. Interestingly, there is a problem with how to arrange the order so that the movement cost for incremental updates is minimized. It is similar with the TCAM update problem [36], but leave it for future work because it is not a critical problem for FACL.

F. Some Discussions

1) DRAM Support: Although FACL is introduced with memory as SRAM on the chip above, it is easy for FACL to use DRAM as [26]. LS rather than DT should use DRAM, which leads to at most one DRAM read during the lookup in one decision tree. From the measurement of Shuhai [37], High Bandwidth Memory (HBM) could achieve 20Miops with 2Kb width in one channel. An IPv4 ACL rule size is about 200b, thus 10 rules could be loaded with one DRAM read, which is enough for most decision tree algorithms.

Algorithm 1 DT Mapping

Input: root node r, all leaf set $\mathbb{L}$, all tree node set $\mathbb{N}$, the max layer level D, spare space $S[D+1]$, and spare start address $A[D+1]$ for each layer.
Output: The map result $\mathbb{M} : \{(n, RamAddr) | \forall n \in \mathbb{N}\}$.

1: **if** $|\mathbb{N}| > \sum_{0 \le i \le D} S[i]$ **then**
2: **return** mapping fails due to not enough room.
3: **end if**
4: initial set $\mathbb{M} \leftarrow \varnothing, \mathbb{V} \leftarrow \varnothing, y \leftarrow 0$.
5: **while** $r \notin \mathbb{V}$ **do**
6: **if** $S[y] > 1$ **then**
7: place r at layer y: let $r.layer \leftarrow y$.
8: let $r.RamAddr \leftarrow (y, A[y])$.
9: let $A[y] \leftarrow A[y] + 1$, $S[y] \leftarrow S[y] - 1$.
10: let $\mathbb{V} \leftarrow \mathbb{V} \cup \{r\}$, $\mathbb{M} \leftarrow \mathbb{M} \cup \{(r, r.RamAddr)\}$.
11: **end if**
12: let $y \leftarrow y + 1$
13: **end while**
14: **while** $|\mathbb{V}| < |\mathbb{N}|$ **do**
15: Find a leaf node l_0 with $l_0 \in \mathbb{L} \setminus \mathbb{V}$ and $\forall l' \in \mathbb{L} \setminus \mathbb{V}$, $l_0.depth \ge l'.depth$.
16: $r_0 \leftarrow l_0.parent$, $\mathbb{P} \leftarrow \{l_0\}$.
17: **while** $r_0 \notin \mathbb{V}$ **do**
18: let $\mathbb{P} \leftarrow \mathbb{P} \cup \{r_0\}$, $r_0 \leftarrow r_0.parent$.
19: **end while**
20: let $x \leftarrow r_0$.
21: **while** $x! = l_0$ **do**
22: let $y \leftarrow x.layer + 1$, $d \leftarrow |x.children|$, $\hat{c} \leftarrow 0$.
23: **while** $S[y] < d$ and $c < D + 1$ **do**
24: let $y \leftarrow (y+1)\%(D+1)$, $c \leftarrow c + 1$.
25: **end while**
26: **if** $c >= D + 1$ **then**
27: **return** mapping does not succeed because we cannot find enough continuous space.
28: **else**
29: place $x.children$ in layer y and update their RamAddr, $\mathbb{M}$, $\mathbb{V}$.
30: $A[y] \leftarrow A[y] + d$, $S[y] \leftarrow S[y] - d$
31: **end if**
32: find $\hat{x}$ which $\hat{x}.parent = x$ and $\hat{x} \in \mathbb{P}$, let $x \leftarrow \hat{x}$.
33: **end while**
34: **end while**
35: **return** mapping succeeds and we get the result $\mathbb{M}$.

2) Incremental Update: FACL could accommodate different trees without reconfiguring FPGA circuits, so we could use the ping-pong method to do the update when the users want to use a new ruleset easily. However, for the incremental update that only a few rules are changed, ping-pong is a little expensive.

For the scenario, CPM can get a new memory mapping result and the modification is carried out according to the

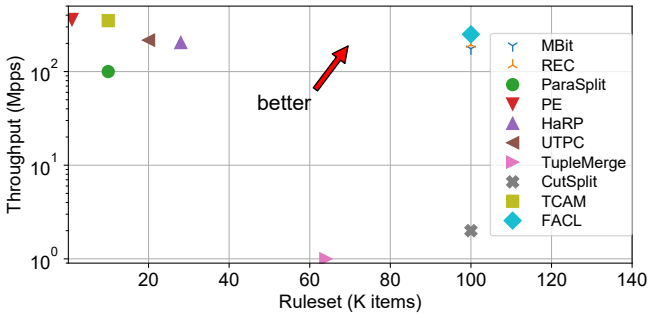
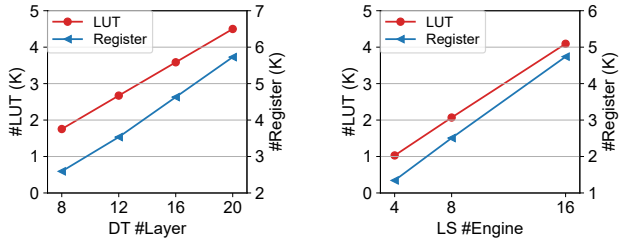
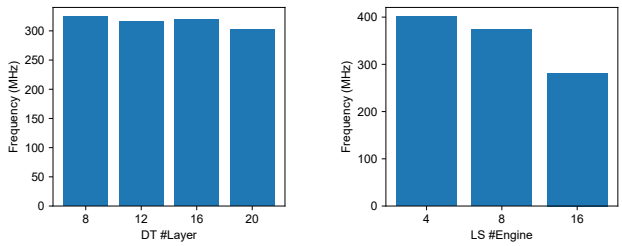


Fig. 9: FACL and alternatives.



(a) Decision Tree (b) Linear Search

Fig. 10: The FPGA resource usage for modules.



(a) Decision Tree (b) Linear Search

Fig. 11: The max FPGA frequency for modules.

difference between the original memory map and a new one by inserting write bubbles into CFM. If there is enough spare space for the new nodes, they are directly written to the spare space. After writing the new ones, we change the parent's pointer to the new node address and the update is done. If space is not enough, we could reuse the space of its posterity after marking a node as invalid as discussed in §III-B. The drawback is that the packets, which need the node information to process, have to be circulated and packet loss happens if too many packets are recirculated and the buffer is full. Luckily, ACL rules do not update frequently and the alterations are few.

3) Skip Linear Search: In the tree node encoding of FACL, there is an OP for the leaf node which returns the rule ID directly. As depicted in §II, we can skip the linear search to know that the packet matches no or one rule. In this case, DT would set *found_in* to inform the following stages to skip the process and save the rule id or MAX_VALUE in the nxt_addr. After leaving DT, the packet would be scheduled by NoC to the Reorder Engine because the nxt_addr does not belong to the LS, which saves the processing capability of LS.

IV. Implementation

We implement CFM with Verilog (∼1000 lines) based on the design of PANIC [19] and CPM with Python (∼1000 lines) based on NeuroCuts [18]. The rulesets are from the classical open-source benchmarks [33].

For CFM, we add the valid signal and ready signal to wrap our modules, thus each engine is in AXI format. We use Vivado 2020.01 and Virtex UltraScale+ VCU118 Evaluation Platform (xcvu9p-flga2104-2L-e) to evaluate the FPGA resources usage. The platform has the same Lookup Tables (LUTs) and Registers resources as Xilinx typical SmartNIC, U200, with 1,182K LUTs and 2,364 Registers. For the next generation Xilinx SmartNIC U250, there are 1,728K LUTs and 3,456K Registers, much more than U200.

For CPM, we run on a server with 32G DDR3 1600MHz DRAM and *Intel(R) Core(TM) i7-4790@3.60GHz* CPU with Python 3.8 with a single process.

V. Evaluation

First, we summarize the performance of FACL and other alternatives in Fig.9. ParaSplit [7], PE [27], HaRP [3], UTPC [15], MBit [32] and REC [2] are FPGA designs while TupleMerge [4] and CutSplit [16] are the latest software implementations with more than 10 physical CPU cores. FACL achieves 250 Mpps for 100K ruleset simultaneously, which is the best among them. Most FPGA-based solutions perform with 100x higher throughput than software ones, which reveals the great advantage of FPGA. MBit is the latest FPGA design on the DPU, which achieves 248 Mpps throughput for 100K ACL ruleset. FACL performs similarly to MBit but accommodates a variety of trees while MBit must reconfigure the FPGA circuits to ensure that the SRAM allocation is suitable for the input ruleset.

Next, we discuss the performance of FACL from two aspects: the consumption of FPGA resources and the mapping results of decision trees.

A. FPGA resources

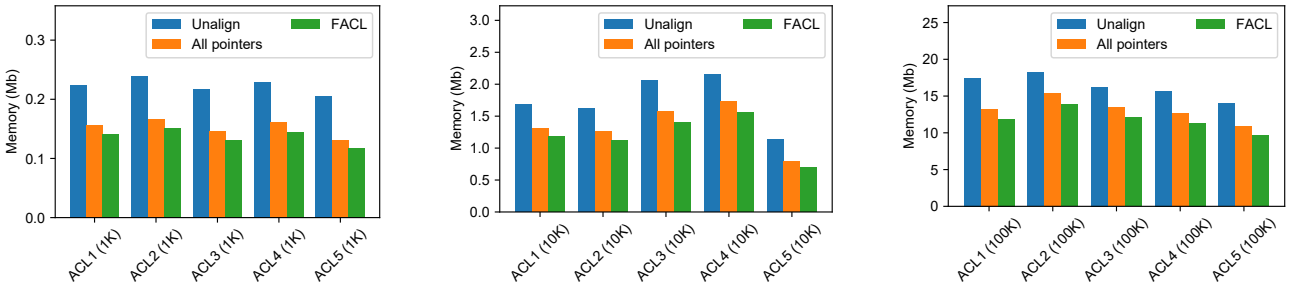
As depicted in Table V, we implement a DT with 20 layers and there are 4K nodes space in each layer. The depth of most trees is not larger than 16, shown in Table III, so DT-20 is enough to handle most cases, where the packets reach LS after leaving the DT module without the need to go through a DT again. An LS allows for 8K*8=64K rules and Reorder Engine has 1K item buffer.[2] They all occupy very few LUT and register resources: less than 1% of the board. We connect four DT-20(4K) modules, three LS-8(8K) modules, and one Reorder Engine to the PANIC framework, which means 320K tree nodes and 192K rule items in total, and the resource usage is still few, less than 20%. The most occupied resource is RAM

[2]The number of '4K' or '8K' is because a URAM is 72b*4K in Xilinx and we want to make full use of the parallel ability of URAM.

TABLE V: The FPGA resources usage. The clock is 250 MHz

	CLB$^+$ LUT	CLB Register	CARRY8	CLB	BRAM	URAM	WNS$^+$ (ns)	WHS$^+$ (ns)
Board	1182240	2364480	147780	147780	2160	960		
DT-20 (4K)	4481	5694	180	1384		20	0.441	0.003
LS-8 (8K)	2083	3890	110	623		48	1.330	0.009
Reorder (1K)	90	260	4	43	3		2.034	0.012
FACL-on-PANIC*	134756	108476	1428	25040	118.5	224	0.046	0.010
Ratio	11.4%	4.6%	1.0%	16.9%	5.5%	23.3%		

* FACL-on-PANIC has 4*DT-20(4K), 3*LS-8(8K) and 1*Reorder(1K) on PANIC.
$^+$ CLB=Configurable Logic Block. WNS = Worst Negative Slack. WHS = Worst Hold Slack.



(a) 1K ruleset (b) 10K ruleset (c) 100K ruleset

Fig. 12: The memory reduction for the placement of rules in LS.

(BRAM&URAM), but the board has about 300Mb, enough for even 1M rules.

For LUT and register resources, the PANIC framework utilizes a lot due to its complex switching and scheduling policies. The modules we design occupy a little. As depicted in Fig.10, we also evaluate the DT(4K) with 8, 12, 16, and 20 layers respectively; LS(4K) with 4, 8, and 16 parallel engines. The resources increase linearly as expected, always no more than 5K LUTs and 5K registers.

B. Throughput and Delay

From the WNS[3] and WHS in Table V, we can know that FACL could run at 250Hz, which means that the throughput is about 250 Mpps, 173Gbps for even the smallest 64B packets, enough for the U200 ports. The throughput of FACL is limited by the PANIC, which employs a complicated full-mesh switch scheduling on the FPGA. To reveal the throughput limit of FACL, we vary the configuration of DT and LS, calculating their maximum operating frequency in Fig.11. The frequency is calculated by $f = \frac{1}{T-WNS}$ where T is the clock period [39], i.e. 4 ns for the 250 MHz clock.

As depicted in Fig.11a, we vary the layer number from 8 to 20 for the decision tree module. The max frequency decreases a little, from 320 MHz to 300 MHz. The pipeline architecture of DT works very well. As depicted in Fig.11b, we vary the number of LSE from 4, 8, to 16. The max frequency is 400 MHz, 374 MHz, 281 MHz respectively, going down fast from 8 to 16 parallel engines. The reason is that all results from the parallel engines should be aggregated, which brings many difficulties to the layout of engines and routing of lines on FPGA. We can also change the structure of LS to pipeline instead of parallelism, but it is unnecessary because 8 parallel engines can work at 300 MHz and the pipeline structure of LS would introduce additional delay.

Multiple engines for one SRAM can speed up further, with more LUTs and registers consumed. The maximum operating frequency for URAM/BRAM is 600 MHz [1], and thus we could reach 600 Mpps at most for an FPGA ACL engine, but the asynchronous sequential circuits are much more complex and 600 MHz is a too ideal result for FPGA SRAM to realize, thus we keep FACL synchronous circuits and 300 Mpps could be easily achieved with better Network-on-Chip.

When it comes to the delay of classification, the latency for one SRAM read is 2 cycles for better routing in FACL, and we have 20 SRAM read on average. With the additional delay between modules and from the calculation, the common latency for a match is about 50 cycles, 200 ns, which is a little larger than TCAM, but acceptable for our scenario and much better than the μs delay of software solutions [16]. Reorder engine (1K) allows the maximum processing delay jitter of matching as 1K cycles, which meets the requirement very well.

C. Compiler

To make the decision trees easier to place on CFM, we limit the maximum children number of a node to 128 for those decision tree algorithms. With CutSplit, we compile a 10K ACL ruleset to three trees, with inner node and rule item as 3,361/8,998 respectively for LS-4, while a 100K ACL ruleset as 23,530/61,894. They do not cause any recirculation for DT-20(4K) with our Algorithm 1 by pushing some inner nodes to the layers behind.

[3]WNS points to the path having the maximum negative slack. If it is positive, the modules can work without any violations at the clock frequency.

We further explore how much memory is saved by our LS placement design. Apart from our design and unalign method in Fig.6, there is also an *All pointers* method with $m*log_2(K)$ cost per node, introduced in §III-C. We compare their memory usage in Fig.12. FACL saves 24%-42% memory than *Unalign* method and 10% memory than *All pointers* method for all rulesets, with the number of rules 1K, 10K, and 100K.

VI. Conclusion

FACL, a flexible and high-performance ACL engine on FPGA-based SmartNIC, could achieve 250 Mpps for 100 K ACL rulesets and support the arbitrary rule update because it could allow any depth of decision trees with a small sacrifice of average throughput. We demonstrate the feasibility of our FPGA design and the memory savings of our compiler through experiments. Future work will design a less complex network topology and thus a new compiler algorithm to further improve FACL by reducing the cost of its full-mesh NoC.

References

[1] A. Alyushin, S. Alyushin, and V. Arkhangelsky. High-Speed Pattern Matching Architecture on Limited Connectivity FPGA. In *AICT*, 2017.

[2] Y.-K. Chang, H.-C. Chen, and G. Parr. Fast Packet Classification using Recursive Endpoint-Cutting and Bucket Compression on FPGA. *The Computer Journal*, 62(2):198–214, 2019.

[3] C.-H. Chou, F. Pong, and N.-F. Tzeng. Speedy FPGA-based packet classifiers with low on-chip memory requirements. In *FPGA*, 2012.

[4] 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 ToN*, 2019.

[5] J. Daly and E. Torng. Bytecuts: Fast packet classification by interior bit extraction. In *INFOCOM*. IEEE, 2018.

[6] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud.

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

[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*, 2014.

[9] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. New York, NY, USA, 2020. Association for Computing Machinery.

[10] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[11] P. He, G. Xie, K. Salamatian, and L. Mathy. Meta-algorithms for software-based packet classification. In *ICNP*, 2014.

[12] W. Jiang and V. K. Prasanna. Large-Scale Wire-Speed Packet Classification on FPGAs. In *FPGA*, FPGA '09, 2009.

[13] Juniper. SRX4600 SERVICES GATEWAY Datasheet. Website, 2022. https://www.juniper.net/content/dam/www/assets/datasheets/us/en/security/srx4600-services-gateway-datasheet.pdf.

[14] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. *ASPLOS*, 2016.

[15] A. Kennedy and X. Wang. Ultra-high throughput low-power packet classification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2013.

[16] W. Li, X. Li, H. Li, and G. Xie. CutSplit: a decision-tree combining cutting and splitting for scalable packet classification. In *INFOCOM*. IEEE, 2018.

[17] W. Li, T. Yang, O. Rottenstreich, X. Li, G. Xie, H. Li, B. Vamanan, D. Li, and H. Lin. Tuple space assisted packet classification with high performance on both search and update. *IEEE Journal on Selected Areas in Communications*, 38(7):1555–1569, 2020.

[18] E. Liang, H. Zhu, X. Jin, and I. Stoica. Neural Packet Classification. Association for Computing Machinery, New York, NY, USA, 2019.

[19] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *OSDI*, 2020.

[20] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE*, 107(4):746–764, 2019.

[21] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. New York, NY, USA, 2019. Association for Computing Machinery.

[22] Z. Liu, S. Sun, H. Zhu, J. Gao, and J. Li. BitCuts: A fast packet classification algorithm using bit-level cutting. *Computer Communications*, 2017.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[24] Mellanox. NVIDIA Announces Mellanox InfiniBand for Exascale AI Supercomputing. Website, 2022. https://nvidianews.nvidia.com/news/nvidia-announces-mellanox-infiniband-for-exascale-ai-supercomputing.

[25] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna. Multi-dimensional packet classification on fpga: 100 gbps and beyond. In *FPT*. IEEE, 2010.

[26] Y. Qu and V. K. Prasanna. High-performance pipelined architecture for tree-based IP lookup engine on FPGA. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 114–123. IEEE, 2013.

[27] Y. R. Qu and V. K. Prasanna. High-performance and dynamically updatable packet classification engine on fpga. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):197–209, 2016.

[28] A. Rashelbach, O. Rottenstreich, and M. Silberstein. A computational approach to packet classification. In *SIGCOMM*, 2020.

[29] V. Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 367–379, New York, NY, USA, 2019. Association for Computing Machinery.

[30] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. New York, NY, USA, 2003. Association for Computing Machinery.

[31] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *SIGCOMM*, 2016.

[32] J. Tan, G. Lv, Y. Ma, and G. Qiao. High-performance pipeline architecture for packet classification accelerator in dpu. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–4, 2021.

[33] D. E. Taylor and J. S. Turner. Classbench: A packet classification benchmark. *IEEE/ACM transactions on networking*, 2007.

[34] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang, D. Yu, C. Tian, H. Zheng, and B. Y. Zhao. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. New York, NY, USA, 2019. Association for Computing Machinery.

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

[36] Z. Wang, H. Che, M. Kumar, and S. K. Das. CoPTUA: Consistent policy table update algorithm for TCAM without locking. *IEEE Transactions on Computers*, 2004.

[37] Z. Wang, H. Huang, J. Zhang, and G. Alonso. Shuhai: Benchmarking high bandwidth memory on FPGAs. In *FCCM*. IEEE, 2020.

[38] A. S. Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[39] Y. Xin, W. Li, Y. Wang, and S. Yao. An FPGA-based High-Throughput Packet Classification Architecture Supporting Dynamic Updates for Large-Scale Rule Sets. In *INFOCOM WKSHPS*. IEEE, 2021.

[40] Y. Yan, A. F. Beldachi, R. Nejabati, and D. Simeonidou. P4-enabled smart nic: Enabling sliceable and service-driven optical data centres. *Journal of Lightwave Technology*, 38(9):2688–2694, 2020.