The Journal of Supercomputing (2019) 75:5667–5687
https://doi.org/10.1007/s11227-019-02818-5

Cuckoo filter-based many-field packet classification using X-tree

A. A. Abdulhassan[1] · **M. Ahmadi**[1]

Published online: 21 March 2019

Abstract

Software-defined networking (SDN) is a new paradigm which emerged in the networking area. Packet classification is an interesting topic that has considered in both traditional and SDN networks. Packet classification involves inspection of multiple fields against a set of thousands of rules called rule-set. With the increasing throughput demands in modern networks and the growing size of rule-sets, performing wire-speed packet classification has become challenging and an important topic in recent years. Packet classification is called as many-field packet classification in the SDN because of increasing the number of header fields. In this paper, a scalable many-field packet classification by employing the extended tree (X-tree) integrated with an efficient probabilistic data structure called Cuckoo filter is proposed. X-tree has high performance from the lookup, insertion, and update aspects. However, X-tree has a high memory requirement, Cuckoo filter as a probabilistic data structure is integrated within each X-tree node to outperform memory requirements and providing more classification throughput. Our experiment results show that the proposed approach achieves high throughput while requiring low memory. In addition, the proposed approach improves latency 2.4×, 6.15× and 4.75× in comparison with DBAMCP, BSOL-RC and BF-AQT for 64 k rule-set, respectively.

Keywords Software-defined networking · OpenFlow · Many-field packet classification · X-Tree · Approximate membership query · Cuckoo filter

✉ M. Ahmadi
mahmadi1352@gmail.com; m.ahmadi@razi.ac.ir

A. A. Abdulhassan
Alaaabbas069@gmail.com

[1] Department of Computer Engineering and Information Technology, Razi University, Kermanshah, Iran

1 Introduction

Modern networking technologies such as software-defined networking (SDN) have provided high-quality network services such as firewalls, quality of service, and network security. To enable such services, SDN used a network kernel function called packet classification. With the fast evolution of Internet traffic technologies, packet classification has become one of the foundational techniques of modern networking devices. Packet classification is the process of categorizing the incoming packets into flows and assigning a specific action to each of these flows. The traditional packet classification techniques use 5-fields, source IP, destination IP, source port, destination port and protocol fields. While the SDN proposes the many-field packet classification which exploits many number of fields from 15 to 40 fields.

1.1 Problem statement

High-performance packet classification algorithms have gained a great interest from academics and industrialists. The packet classification assigns action according to highest priority rule selected among a set of predefined rules. OpenFlow [1] is an open standard communications protocol that provides the many-field packet classification. OpenFlow uses the rule-set to perform packet classification on the incoming packets. Rule-set of a classifier is a set of predefined rules. This set may consist thousands of rules, which each rule contains up to 15 matching fields in the newest OpenFlow version [2]. Each field may require different matching type according to the field's type (exact, prefix, and range) [3]. The main problems that make it as a challenge to perform the wire-speed packet classification are the growing size of rule-set, exponentially increasing throughput demands in modern networks, and high packet header dimensionality.

1.2 Contributions

Many hardware- and software-based packet classification solutions have been proposed. Decision tree-based algorithms are software-based solutions can provide high throughput by representing the rule-set as a tree data structure to limit the number of rule comparisons. The query speed depends on the height of decision tree. However, decision tree-based algorithms require an excessive amount of memory because it redefines the whole rules at each tree level. In this paper, we propose a novel decision tree-based approach called CX-tree that can achieve high classification throughput with low memory. The main idea is to use a space-efficient probabilistic data structure called Cuckoo filter (CF) [4] integrated with an extended node tree (X-tree) [5] for many-field packet classification. Our choice of X-tree came from that the X-tree has high query performance. This is because the following reasons:

1. At each tree level, only one branch of the X-tree needs to be expanded for each incoming packet.
2. X-tree is a balanced tree data structure that *M* rules can be indexed in each node. This means that it gives lower tree height.

Cuckoo filter is an approximate membership query (AMQ) data structure. AMQ data structures are used in many databases, networking, and memory systems to handle the large amounts of data. Integrating Cuckoo filter within the tree nodes can minimize the high memory requirement of X-tree and increases the lookup time. Cuckoo filter uses a small amount of memory to represent a large set of elements and supports a membership query with small false positive probability.

The proposed approach represents each rule from the rule-set as a minimum bounding rectangle (MBR) in X-tree by converting its source/destination IP addresses to range values. Then, it hashes the remaining fields from each rule in a Cuckoo filter integrated with each node. The experimental results show that the proposed data structure achieves high classification throughput with low latency in comparison with the most important works done in this area. The main contributions of this paper are as follows:

– Proposal of the CX-tree (integration of X-tree and Cuckoo filters) to organize the rule-set in an X-tree integrated with a Cuckoo filter.
– Using CX-tree for the first time for many-field packet classification.
– Perform the performance comparison of the proposed approach against our previous works [26, 27] and three modern many-field packet classification techniques including (DBAMCP, BSOL-RC and BF-AQT).

The rest of this paper is organized as follows. Section 2 reviews the related works. Section 3 presents the concept of X-tree and Cuckoo filter data structures. Section 4 describes CX-tree packet classification technique. Section 5 evaluates our proposed technique. Finally, we conclude the paper in Sect. 6.

2 Related works

In this section, the related works in many-field packet classification in SDN is reviewed. Many researchers have focused to address different problems in SDN. In [6], a compressive survey for multi-controller research in SDN is proposed. It introduced the overview of multi-controller, including the origin of multi-controller and its challenges, and then, classified multi-controller research into four aspects (scalability, consistency, reliability, and load balancing). In [7], presented an adaptive update mechanism based on the quality of service (QoS)-aware traffic classification and real-time network status. In [8], a multiagent-based service composition approach, using agent-matchmakers and agent-representatives for the efficient retrieval of distributed services and propagation of information within the agent network to reduce the amount of brute-force search is proposed. Other researchers

focused on one of the most important functions in SDN which is the packet classification. Packet classification has gained a big attention from the network developers. The classical packet classification is the network function that supported by the traditional network devices to deal with forwarding packet according to 5 packet header fields only. Many-field packet classification is the network function that used to classify packets into flows according to a large predefined set of rules called rule-set. Each rule in that rule-set contains up to 15 matching fields. In the following sections, the current classical and many-field packet classification techniques are described briefly.

2.1 Classical packet classification

Many of hardware and software solutions for the classical packet classification problem have been proposed. Hardware solutions such as Ternary Content Addressable Memory (TCAM) [9] and Bitmap Intersection [10, 11] provide a good classification performance by using parallel lookup but using hardware resources limit the size of the forwarding tables and rule-sets. Hence, the software solutions are required to solve the scalability problem. Software solutions use extendible data structures to store the rule-sets and forwarding tables. The current classical packet classification solutions can be classified in the following subsections.

2.1.1 Basic search algorithms

This group contains the simplest solutions for packet classification such as Linear Search, Hierarchical Tries (H-Tries), and Hierarchical Binary Search Tree. The linear search algorithm uses a simple list to store the rules in decreasing priority order and searches for matching in that list in a sequential manner. Hierarchical tries use the prefixes to recursively build two-dimensional trie, and then, for each incoming packet, it traverses that trie looking for matching rule. The solutions that belong to this group store the rule-sets in a basic data structure with a small amount of memory, but it suffers from the scalability problem.

2.1.2 Heuristic algorithms

This group contains solutions such as the Recursive Flow Classification (RFC) [12], and Distributed Cross-producing of Field Labels (DCFL) [13]. The classification query is divided into a number of exact match queries. RFC solution can be used for classical and many-field packet classification. It divides the rule-set into a smaller set and performs the same single-field matching over several phases. This group requires more preprocessing time to perform rule-sets and query dividing.

2.1.3 Decomposition-based algorithms

In this group, the rule-sets are divided into several single-field rule-sets, and then lookup for incoming packet fields performed separately. The results are then

combined from all fields. Field Split Bit-Vector (FSBV) [14] is an example of this group solution. This group spends more additional memory and processing time to merge the fields results in order to obtain the final result.

2.1.4 Decision tree-based algorithm

In decision tree-based solutions, each header field is defined as a search space, and then, these spaces are divided into smaller sub-spaces and so on. To classify incoming packet, it performs a linear search on these sub-spaces [15]. The HyperCuts [16, 17] algorithm generates a shorter decision tree by performing multiple field cutting per step to form shorter decision tree. It has high scalability and low rule-set dependency. The main decision tree-based solutions drawback is its high memory requirement.

2.2 Many-field packet classification

In recent years, the researchers have focused on many-field packet classification problems that classify the incoming packets according to the rule-sets with more than 15 fields. Following, we list a group of the most recent works that have tried to accelerate the packet classification from different directions and are related to our work. Several software-based many-field packet classification algorithms have been proposed. Most of these proposals use tree data structures to represent the rule-sets. Some of them use probabilistic data structures to accelerate querying these trees. The authors in [18] proposed a packet classification algorithm using a Bloom filter in a leaf-pushing area-based quad-tree. They hash the tree nodes in a Bloom filter stored in on-chip memories. Another Bloom search solution proposed in [19] which uses the GPU-accelerated implementation. The authors in [20] proposed a solution based on decision tree for many-field packet classification. They called their solution Binary search on levels (BSOL). They represent the rules as a hyper-rectangle in a two-dimensional address space, then generate a binary decision tree and associate each tree node with space. The space of each node is divided between its two children. They associate each node with a filter list. They concluded that their solution has a high classification speed. The authors of [21] employed a replication control scheme on the binary search on levels (BSOL) algorithm to minimize the memory requirements, and the processing latency, then call the new algorithm as (BSOL-RC). The memory requirement is still high because they have to store the entire rule-set plus many additional data in their tree. The authors in [22] proposed many-field packet classification approach using range-tree [23, 24] and hash table [25] to build an efficient search algorithm. Firstly, they construct a range-tree or a hash table for each filed according to the field's type. Subsequently, they query their data structure for each field of the incoming packets independently and then merge the partial searching results to compute the final result. This approach also requires large preprocessing time and needs high processing latency including the time required to reach in the range-trees, the time required to search in the hash tables and the merging time

In this work, the X-tree [5] is integrated with an approximate membership query (AMQ) data structure called Cuckoo filter to create high-performance packet classifier. The prefix fields of each rule are used to build the X-tree, then the remaining fields from each rule are hashed into Cuckoo filters and integrated them to the X-tree's nodes. The resulted CX-tree is a balanced tree which has high search speed and requires low memory due to integrating Cuckoo filters in its nodes. Compared to our previous works [26, 27], employing X-tree gives better processing latency and integrating Cuckoo filter with X-tree gives better memory utilization.

3 Cuckoo filter and X-tree data structures concepts

To design the proposed high-performance classifier, a new data structure is created and is called as Cuckoo filter-based X-tree (CX-tree). The CX-tree is an extended node tree (X-tree) [5] integrated with Cuckoo filters [4]. The CX-tree has been used to reorganize the rule-set in a tree data structure in order to accelerate the classification process and to minimize the memory requirements. In the next subsections, the X-tree and Cuckoo filter data structures are described in details.

3.1 Cuckoo filter

A Cuckoo Filter (CF) [4] is a probabilistic data structure. Its idea was originated from Cuckoo hashing, which was proposed by Pagh et al. [25]. CF can perform the usual probabilistic data structures operations (insertion, and query), as well as removing, re-size, and merging. An empty CF is an array of buckets. Each bucket can store K fingerprints. CF usually uses two hashing functions to generate a key and two bucket numbers. If the CF fails to insert the key in the first bucket (the slot already contains k keys), then it will try to insert it into the second bucket. If the alternative bucket is also full, then the CF uses specific operation called reallocation operation to deal with such situations. An advanced version of CF uses only one bucket number to insert the element's key, and if it fails to insert the key in that position, then it generates another bucket number using the key in order to accelerate the reallocation operation. CF has a high-speed query and low memory requirements.

Figure 1 depicts a Cuckoo filter with the size of 8 buckets, each bucket can store $k = 1$ item. Initially, the items b and c are already in the Cuckoo filter. In Fig. 1 (a) adding item "a", $h1(a) = 1$, and $h2(a) = 6$, both buckets are checked if one of them is empty then $fh(a)$ is added into that empty bucket. In this case, both $h1(a) = 1$, and $h2(a) = 6$ buckets are empty, then one of them randomly is selected (bucket(6)) and $fh(a)$ is inserted into it. In Fig. 1b adding item "d", $h1(d) = 2$, and $h2(d) = 5$. In this case, one bucket is empty (bucket(5)), then $fh(d)$ can be added into that empty bucket. In Fig. 1c adding item "e ", $h1(e) = 2$, and $h2(e) = 6$. In this case, both buckets are not empty, then it is added in one of those buckets (bucket 6), kicks out the existing item ("a"), and reinserts this old item to its own alternate location (bucket 1). The process of kicking out of existing items and reinserting them in their alternate locations called reallocation process. In some cases, reallocating the old item

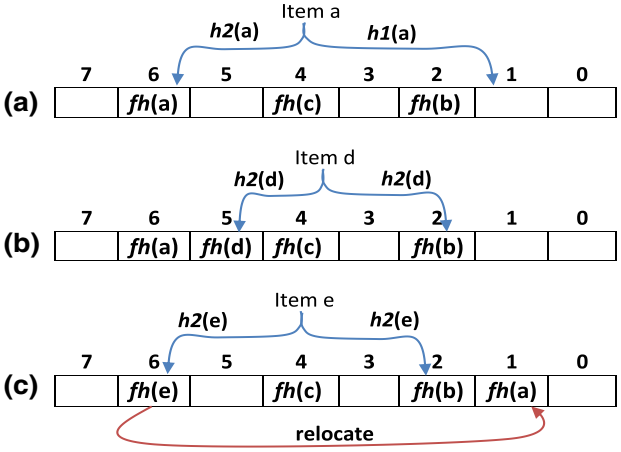

Fig. 1 **a** Inserting "a" at bucket (6), **b** inserting "d" at bucket (5), **c** inserting item "e" at bucket(6)

may also require kicking out another existing item, and this procedure may repeat until a vacant bucket is found or the maximum number of reallocation is reached (e.g., 150 times in our implementation). Even with a long sequence of reallocation process, Cuckoo filter still has $O(1)$ insertion time and high space occupancy.

To lookup Cuckoo filter for a given item x, first, calculate x's fingerprint $fh(x)$ and two candidate buckets $h1(x)$, and $h2(x)$, and then read these two buckets. If there is fingerprint match $fh(x)$ in these two buckets then returns true otherwise, returns false. The Cuckoo filter has more advanced features so that Cuckoo filter ensures no false negatives as long as bucket overflow never occurs, has better lookup performance, and supports deleting items dynamically [28].

3.2 eXtended node tree (X-tree)

The X-tree (eXtended node trees) is a hierarchical data structure [5] as an improved version of R*-tree [9]. X-tree has been used widely to dynamic organization of the high-dimensional data and geographical coordinates objects as rectangles or polygons. X-tree is a balanced tree which describes multi-dimensional geographical objects as items of nodes by generating minimum bounding rectangle (MBR) for each item and then describes each group of nearby MBRs as bigger MBR in the upper levels. The main improvement points of the X-tree are the overlap-free split and the supernodes. The overlap-free split means that the search space in X-trees can be divided into disjoint areas.

X-trees can organize its objects automatically so that the reorganization after the insertion or deletion is not necessary. Each internal node in X-tree of M degree can be either normal directory nodes or supernode. Each directory node can store m ($\frac{M}{2} \leq m \leq M$) entries unless it is a supernode (can store more than M entries). Each entry is described as MBR together with a pointer to a node in next level.

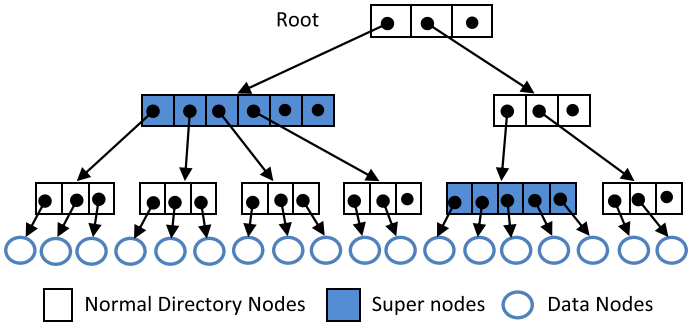

Fig. 2 The overall structure of the X-tree

Each MBR describes a group of sub-MBRs at the next lower level located within its bounded area. Figure 2 presents the overall structure of the X-tree.

The supernodes are extended oversized nodes used to avoid the overlap among MBRs. The highly overlapped MBRs cause a visit to more than one branch of the tree for similar queries. To avoid such problem, X-tree maintains the split history of each node and uses it when an internal node overflows. X-tree does not perform split and creates a supernode as an alternative solution in case of an unbalanced split (after the split, one of the resulted nodes is full and the other is semi-empty). Each data node in X-tree contains MBR and a pointer to the real data objects.

4 The proposed cuckoo filter-based X-tree

CX-tree likes X-tree consists of three different types of nodes: data nodes, normal directory nodes, and supernodes. The data nodes contain minimum bounding rectangles (MBRs) integrated with Cuckoo filters. Each MBR is represented as a node entry in CX-tree and contains a pointer to a real data in the database. The directory nodes contain MBRs integrated with Cuckoo filters and pointers to store the addresses of the next level nodes that belong to its directory. The supernodes are oversized directory nodes used to avoid unbalanced splits in the directory that would result in an inefficient directory structure. The root node can store between (2 to M) entries unless it is a leaf (can store zero or a single entry). The internal nodes and the data nodes can store m ($\frac{M}{2} \leq m \leq M$) unless it is a supernode (can store more than M entries).

CX-tree can organize elements dynamically and can store elements with multi-dimensional information such as rectangles or polygons and geographical objects, where each object contains p multi-dimensional attributes and q identities. The root and internal nodes contain entries in the form of (MBR, CF, $Pointer$) where MBR is the d-dimensions bounded rectangle which covers the p multi-dimensional attributes. CF is a Cuckoo filter which stores the q identities whose multi-dimensional attributes are represented by the MBR, and the Pointer stores the address of the next level. The data nodes contain entries in the form of (MBR, CF, Pointer) where MBR

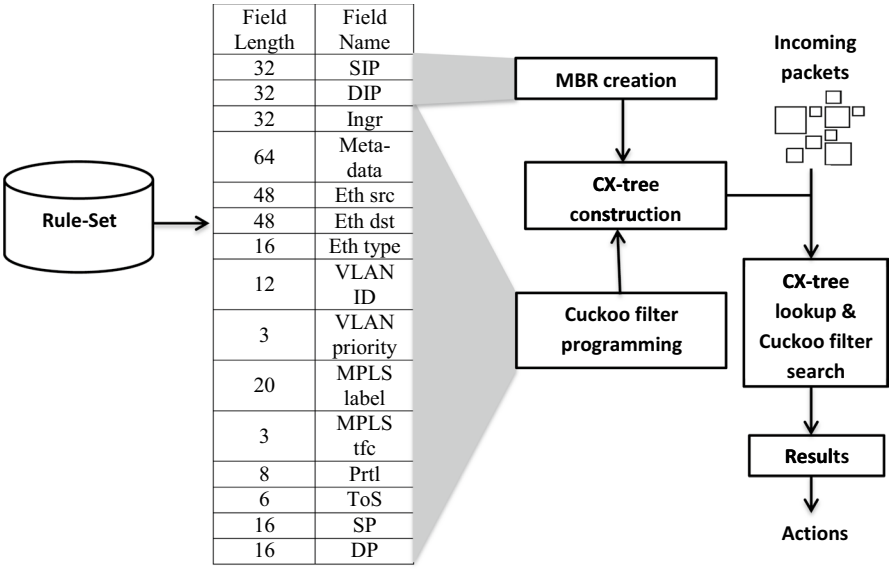

Fig. 3 Block diagram of the proposed system

is a d-dimensions bounded rectangle which covers the p multi-dimensional attributes. CF is a Cuckoo filter which stores the q identities, and the Pointer is an address of data items in the database.

4.1 Many-field packet classification using CX-tree

In this work, the X-tree is used to represent the rule-set by changing the source/destination IP addresses from each rule into a range value, and then these ranges are used to create MBRs to build the X-tree. After that, instead of storing the remaining fields of each rule, they are hashed into a Cuckoo filter and are integrated in the corresponding data node for improving memory utilization and search performance. Subsequently, these fields are hashed in the bigger Cuckoo filters in the next upper levels. Figure 3 depicts the block diagram of the proposed system.[1]

In MBR creation, to create the MBR, the SIP and DIP addresses are selected from each rule. In Cuckoo filter programming, the remaining fields are hashed to the Cuckoo filters, and then in CX-tree construction the resulted MBRs and Cuckoo filters are selected to construct the CX-tree. To query the resulted CX-tree, the incoming packets one by one is checked in the CX-tree lookup and Cuckoo filter search to get the matched rules as results and apply the related actions. In the next subsections, the proposed packet classification approach using CX-tree is described in details.

[1] The codes is available in: https://github.com/AladdinAbdulhassan/CX-tree.

4.1.1 MBRs creation and X-tree construction

To construct the CX-tree, the X-tree is built from the rule-set by creating MBRs using the source IP (SIP), and destination IP (DIP) addresses. To create MBR for a rule, its prefixes is converted into range values. To understand the MBRs creation, a simple example is described in Table 1. Table 1 shows rule-set with 14 rules with rule IDs (M-Z). Each rule contains two multi-dimensional attributes (SIP & DIP) and q identities (not displayed in the table). Every two prefixes of a rule are converted into MBR containing two ranges. Each range is described by its start and end values (R.start, R.end). The R.start and R.end are the minimum and maximum possible values that the IP prefix can represent, respectively. After creating MBRs, these MBRs are inserted one by one into an empty X-tree. The insertion operation of X-tree is described in the algorithm in Fig. 4 and the final resulted X-tree of ($M = 3$) degree is depicted in Fig. 5. The graphical representation of the corresponding X-tree for the rules in Table 1 is depicted in Fig. 6. Each MBR has been drawn as a rectangle in geometric coordinates. The filled rectangles represent the real rules (X-tree data nodes). The others rectangles represent the directory and supernodes that is generated during X-tree construction operation.

The insert operation of X-tree is performed by creating MBRs from the rules, and then traversing the X-tree from the root node to the data nodes at each level. The X-tree selects the node entry that can cover the new MBR with minimum area and moves to its children until reaching the data nodes. At the corresponding data node, if that node can accommodate the new MBR, then inserts the MBR into that data node and updates the MBRs in the path from the root to the data node so that all of them should cover the new MBR. In case of overflow, after inserting the new entry to a full node, the selected node will contain $M + 1$ entries. In some cases when the node entries are highly overloaded, the split operation is not required even when the selected node is already full. X-tree changes the overflowed node to supernodes. The

Table 1 A simple example of a small rule-set of 14 rules, and the converting process to the range values

RID	SIP	R. Start	R. End	DIP	R. Start	R. End
M	10*	128	191	00011*	24	31
N	101*	160	191	00000*	0	7
O	110*	192	223	10*	128	191
P	010*	64	95	01101*	104	111
Q	10101*	168	175	00*	0	63
R	111*	224	255	00110*	48	55
S	011*	96	127	100*	128	159
T	000*	0	31	0101*	80	95
U	10011*	152	159	100*	128	159
V	01*	64	127	10100*	160	167
W	1001*	144	159	00011*	24	31
X	001*	32	63	0100*	64	79
Y	11*	192	255	01101*	104	111
Z	01*	64	127	00*	0	63

```
Input: Type Rule R, Type Node RN;
/* Inserts a new rule R in an X-tree with root node RN*;
Search in X-tree from the root level to the leaves level.;
At each level, choose node entry that can cover R.mbr with minimum area.;
if the chosen leaf L can absorbs R.mbr then
    Insert R into L;
    Update all the nodes in the path from L to RN ;
    /*so that all of them cover R.mbr*/;
else
    /* L is already full */;
    if Split(L, R.mbr)== True then
        /* topological split was successful */;
        Let ε ← the set of the old entries of L plus R;
        Create two new nodes L1, L2;
        Set two entries from ε to L1, and L2 so that the distance between them is the
        maximum among all other pairs;
        Assign the remaining entries of ε to L1, and L2 according of which of L1, and
        L2 can cover with minimum area;
    else
        /* there is no good split */;
        Change L stat from Normal_node to Super_node;
        Insert R into L;
for MBRs of nodes that are in the path from RN to L1, and L2 do
    Update(MBR) to cover L1, and L2;
    Perform splits at the upper levels if necessary;
    if RN has to be split then
        Call create (new root);
        Tree height←Tree height+1;
```

Fig. 4 Proposed CX-tree insertion algorithm

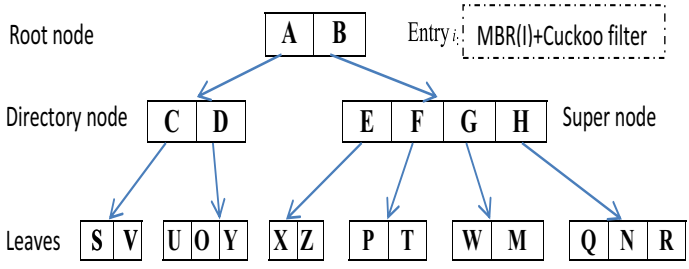

Fig. 5 The corresponding X-tree for the 14 rules from Table 1

supernodes are extended oversized nodes that can store more than M entries. When the overflowed node can be split without overlapping, two new nodes should be created. Subsequently, two entries are selected as seeds so that they have the maximum distance among all. Then two new nodes are created and assigned these seeds to them. After that, assign the remaining $M-1$ entries to node 1 and node 2 according to which of node 1 or node 2 can cover the entry with minimum area. This is depended on the distance between the entries and the seed that firstly inserted to the nodes.

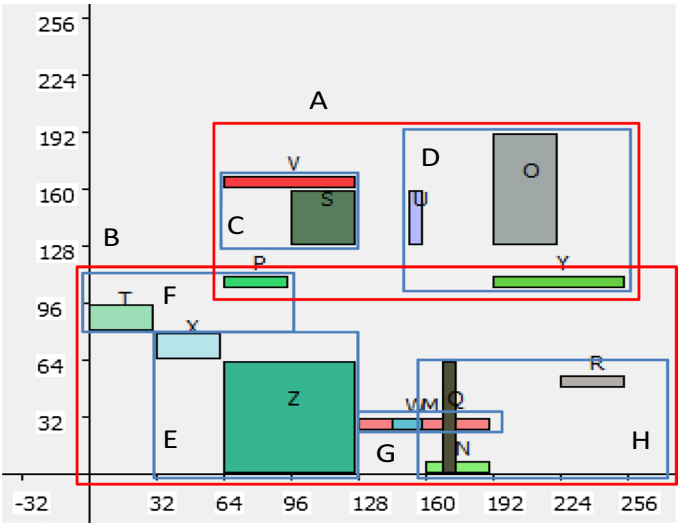

Fig. 6 The graphical representation for the MBRs resulted from 14 rules from Table 1

After that, X-tree will perform splitting into the upper levels if necessary. If splitting is required on the root node, then it will split the root node into two new nodes and generates a new root with two entries to cover the newly generated nodes, and increases the tree height by one.

4.1.2 Cuckoo filters programming

When the construction of X-tree is completed, the integrating of Cuckoo filters is started. The Cuckoo filter programming is a bottom-up operation so that integrating the Cuckoo filters are started from the data nodes and traverse up to the root. In the data nodes, all the remaining fields of each rule are hashed to an empty small Cuckoo filter and integrate it with the node's MBR in the data node's entry. In the next upper levels, at each level, all the Cuckoo filters that are related to the same entry are merged to create a bigger Cuckoo filter and are integrated together with the node's MBR in the directory or supernode's entry. At each level, the node entries will contain Cuckoo filters and fingerprints for all the remaining fields of the rules that is bounded by the corresponding MBR. Figure 7 depicts the process of integrating Cuckoo filters to the X-tree from Fig. 5.

4.1.3 CX-tree lookup and cuckoo filter search

To classify an incoming packet (test whether it matches a rule in the rule-set) using CX-tree, two queries are required: X-tree query and Cuckoo filters query. These queries are performed simultaneously in the CX-tree. Firstly, it creates a query MBR from the queried packet, generates the fingerprint and bucket locations of the remaining fields that are required for Cuckoo filter query. Once the queries

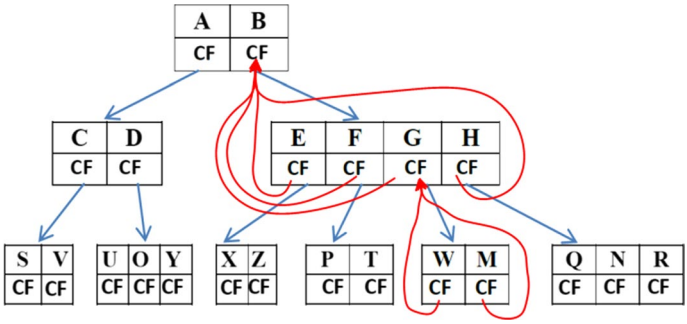

Fig. 7 Integrating CFs to the X-tree

Table 2 A simple example to show the size and value 8 bits fingerprint $fh(x)$, and a variable size candidate bucket $h1(x)$ when hashing the word "TCP" in a data node at the proposed CX-tree with 5 levels using our hashing function

Tree level	fh(x) size (bits)	h1(x) size (bits)	fh(x) value	h1(x) value
Root	8	16	01001010	1101000011110100
Level 2	8	13	11101000	1100110100111
Level 3	8	11	11010010	10100011010
Level 4	8	8	10100001	10100111
Data node	8	6	01010110	101010

parameters are generated, the query process starts by traversing the CX-tree from the root to the data nodes. At each level, it selects appropriate node entry that its MBR can accommodate the query MBR with minimum area and its Cuckoo filter gives positive results for all the remaining fields queries. Then, travels to that node entries' children.

At the data nodes, if there are more than one node entry that match the query MBR and its Cuckoo filter, the rule with the highest priority as a result is reported. To search for a given item x in a CF, first calculates x's fingerprint $fh(x)$ and two candidate buckets $h1(x)$, and $h2(x)$, and then read these two buckets. If there is fingerprint match $fh(x)$ in these two buckets then return true, otherwise, return false. The used Cuckoo filters are longer in the upper levels than the lower levels. They have the same number of buckets if they belong to the same tree level. For query purpose, we build a special hashing function to get a static length fingerprint $fh(x)$ with variable length buckets $h1(x)$, and $h2(x)$. The special hashing function is a modified version of Murmurhashing2 that generates variable fingerprint size in different level of X-tree. That means, the number of bits returned from the hashing function at level L is more than the number of bits returned from the same hashing function at level $L+1$. The same hashing function is used to hash the remaining fields to the Cuckoo filters and to query them. In this way, we can query a set of Cuckoo filters with different size. Table 2 shows a simple example of the hashing function results at different tree level when hashing the word "TCP" for Cuckoo filters with different size. Note that $h2(x)$ can be generated using the exclusive-or between $fh(x)$ and $h1(x)$.

4.2 Theoretical analysis

In this section, the query time, and memory consumption of the proposed data structure is analyzed theoretically. Query time (T) is the amount of the time that the classifier needs to classify single packet theoretically. The maximum height (h) for an X-tree that stores N items (rules) and has m as minimum number of entries at each node is:

$$h_{\max} = \lceil \log_m N \rceil - 1 \tag{1}$$

To query an X-tree, it is needed to visit one branch only at each level. Since one Cuckoo filter at each node's entry is used, then it is needed to query m filter at each level. Cuckoo filter requires a fixed number of memory accesses (at most two memory accesses) [16]. The total query time for CX-tree is

$$T_{\text{CX-tree}} = O((\log_m N) - 1). \tag{2}$$

Memory consumption (M) is the amount of the memory needed to store the rule-set in the proposed data structure. Since the whole rule-set fields in Cuckoo filters at each tree level is stored, and the X-tree has $\log_m N$ levels. To store rule-set with N rules, each rule has K fields, which the average field's size is L bits. Storing one item in Cuckoo filter requires C bits [28].

$$C = \frac{f}{\alpha}, \tag{3}$$

Where f is the fingerprint size, and α is the load factor. To store N rules with K fields for each rule, then the total space size that is required to store the CX-tree in the memory is:

$$M_{\text{CX-tree}} = (K(\log_m N) - 1)) \times \frac{f}{\alpha} \tag{4}$$

5 Performance evaluation

The performance evaluation has been provided using simulated rule-sets created by Classbench tool [29]. We have generated three types of 15 fields rule-sets: access control list (ACL), IP chain (IPC), and firewall (FW) with different sizes of each type (100 rule − 1 M rules). Also, the generated rule-sets have different specifications as depicted in Table 3. We have extended these rule-sets from 5-fields to 15-fields rule-sets by generating more fields for each rule. All the generated fields are exact fields and required only exact matching. Since the proposed method does not depend on the properties of the remaining fields. We have created X-tree for each of rule-set size, and the degree of the tree have set to $M = 6$. Therefore, the maximum supernode size is set to 12, and four supernodes in each tree level as maximum. We have provided performance comparison against the many-field packet classification in [22] (DBAMCP), [21] (BSOL-RC), and (BF-AQT) [18]. We have

Table 3 The simulation rule-sets type, size, and specifications

Rule-set type	No. of rule	Specifications
ACL 100	100	Less than 1% wild-card
ACL 1 K	916	Ratio in SIP & DIP
ACL 5 K	4417	Source port fields are all wild-cards
ACL 10 K	9602	More than 80 % distinct field value in SIP & DIP
ACL 100 K	96,000	Low overlapping rules
ACL 1 M	972,000	
IPC 100	100	More than 6% wild-card
IPC 1 K	937	Ratio in SIP & DIP
IPC 5 K	4459	More than 38 % distinct field value in SIP & DIP
IPC 10 K	9037	Shorter prefixes length than ACL
IPC 100 K	96,000	and more overlapped rules than ACL
IPC 1 M	980,048	
FW 100	100	60% wild-card ratio in SIP
FW 1 K	790	25% wild-card ratio in DIP
FW 5 K	4652	26% distinct field value in SIP & DIP
FW 10 K	9311	Destination port fields are all wild-cards
FW 100 K	96,000	and highly overlapped rules
FW 1 M	963,042	
Input tracing	100, 1 k, 10 K	
Packets	100 k, 1 M	

re-coded and implemented the compared approaches to get a fairer comparison by testing it in our system and using our rule-sets. The same settings have been applied to the compared approaches when applicable. All experiments are conducted on Asus X299 delux h1 equipped with Cor i-9 7900 CPU, 64 GB DDR4 3200 g-skill (4*16) RAM, and 1080 11GB DDR5 GPU.

5.1 Classification throughput

The classification throughput is defined as the average number of packets that our system can classify in a single second. The classification throughput is evaluated using rule-sets with the size of (100–10,000 rules). The results of the proposed approach show that it can classify packets in up to 4.5 Million PPS in small rule-sets and up to 3.48 Million PPS in large rule-sets as can be seen in Fig. 8. The simulated rule-sets have different characteristics from each other, hence the performance differs when working on different types of the rule-sets. The X-tree's lookup time grows logarithmically with the rule-set size that gives X-tree the scalability to very large rule-sets sizes [5]. Comparing to the Bloom filter [30], using Cuckoo filter gives better results with the aspect of different rule-sets type. The classification throughput is decreased significantly when we remove the integrated Cuckoo filters and the rule's fields themselves in the leaves is stored as can be seen in Fig. 9.

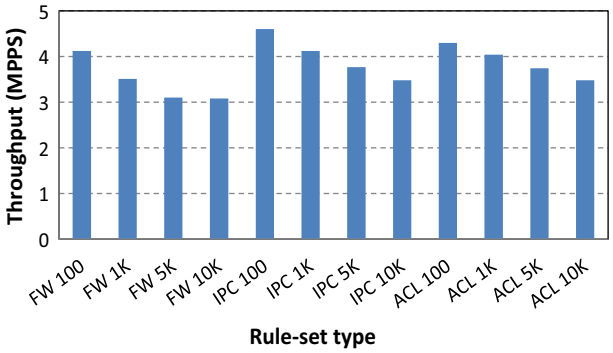

Fig. 8 The proposed system throughput with respect of different type of rule-sets

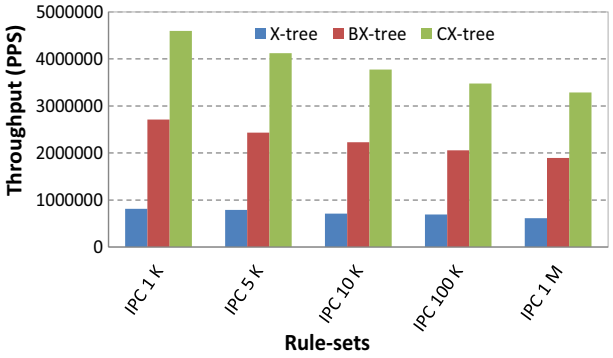

Fig. 9 The proposed system throughput compression against its modified version with removing the Cuckoo filter and replacing it with Bloom filter

Integrating Cuckoo filters in the internal nodes prevents the unnecessary tree search (in the case of MBR success matching and Cuckoo filter query fail). Also, storing the rules in the leaves requires more matching processes and memory accesses.

5.2 Processing latency

The processing latency is defined as the average time that is required to classify a single packet measured by μs. To measure the processing latency, rule-sets size in the range of (1 K–64 K) is used. The processing latency of our approach is compared against our previous work (AMQ-R-tree) [27] and three many-field packet classification approaches (A Decomposition-Based Approach for Scalable Many-Field Packet Classification on Multi-core Processors (DBAMCP) [22], Binary Search on Levels with Replication Control (BSOL-RC) [21], and Packet Classification Using a Bloom Filter in a Leaf-Pushing Area-based Quad-Trie (BF-AQT) [18]). These algorithms have been chosen because they are related to our proposed algorithm so that they

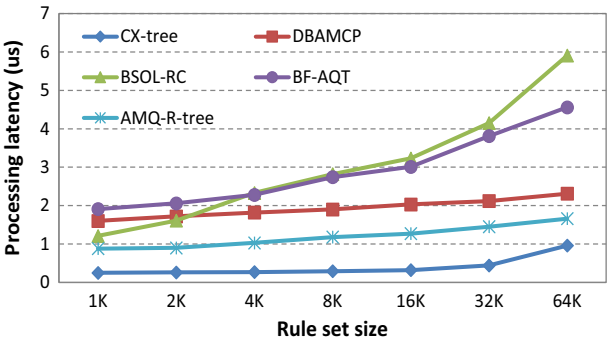

Fig. 10 Average processing latency comparison against AMQ-R-tree [27], BSOL-RC [21], DBAMCP [22], and BF-AQT [18]

use tree data structures in their solutions [18]. Uses a tree data structure combined with a Bloom filter. All these approaches use 15 field rule-sets generated by the same tool (Classbench tool) that we use to generate our rule-sets. The DBAMCP uses range-tree and hashing to search the fields of the input packet header in parallel. After that, it merges the partial results from all the fields in rule-ID sets to produce the final match results. BSOL-RC enhances the binary search on levels in BSOL [20] by employing a replication control scheme. The R-tree in our previous work has more overlapped MBRs than the X-tree in the proposed approach so that in the X-trees, instead of splitting the nodes that have overlapped MBRs, the overlapped MBRs have been gathered by employing supernodes. The range-tree in DBAMCP has empty nodes and more tree levels than CX-tree in our approach. Also, it uses a separated filter to store the remainder fields and a lot of hash tables. The processing latency of DBAMCP contains range-tree search, filter and hash tables query, and partial results merging process. Hence, as can be seen in Fig. 10, the proposed approach latency outperforms the DBAMCP approach by more than $(2x)$ times. The BSOL-RC constructs an unbalanced binary search tree and uses a set of hash tables to store the leaf nodes for every tree level contains a leaf (the leaves are not at the same level). Moreover, in BSOL-RC, the rule-set may be divided into more than two sub-rule-sets because that single decision tree may not be scalable for large rule-sets. BSOL-RC stores replicated filters in a new decision tree to result in fewer memory requirements. Hence, as can be seen in Fig. 10 increasing the rule-sets size increases BSOL-RC latency rapidly. Also, the proposed approach outperforms BSOL-RC with respect to various rule-set sizes. The BF-AQT builds a leaf-pushing area-based quad-Tree by assigning codewords for *SIP&DIP* fields. For *SIP&DIP* that have varied lengths, they determine the codewords by the shorter length. For example, for given rule with $SIP\&DIP = (11*, 101*)$, since DIP is longer, they extend the SIP to three bits, $110*$ and $111*$, then they generate two codewords for each one combined with DIP. For *SIP&DIP* that are varied in length with large number of bits, the BF-AQT generates a huge number of codewords and stores these codewords in a hash table. Hence, it requires more processing latency.

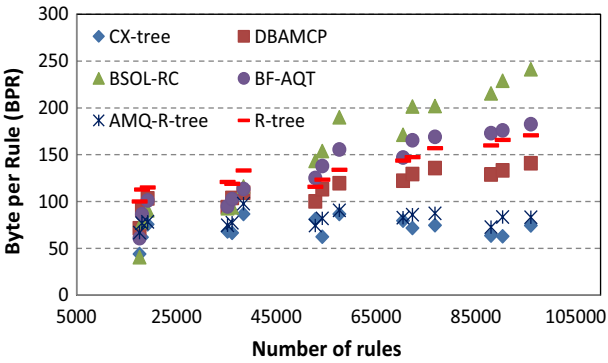

Fig. 11 Memory requirements in Bytes per rule (BpR) comparison against our previous works [26, 27] and BSOL-RC [21], DBAMCP [22], and BF-AQT [18]

5.3 Memory consumption

The proposed classifier's memory consumption has been enhanced significantly when X-tree is integrated with Cuckoo filter. Since our experiments use rule-sets with different sizes. We use Bytes per rule (BpR) to provide a better evaluation of proposed data structure memory consumption. BpR is equal to the total required memory to represent the entire rule-set in the proposed data structure divided by the number of rules. Figure 11 shows the required Bytes per rule (BpR) for the proposed data structure and comparison against our previous works (R-tree [26], AMQ-R-tree [27]), and other three many-field packet classification algorithms including (DBAMCP [22], BSOL-RC [21], and BF-AQT [18]). As can be seen, the proposed data structure requires 40–96 Bytes to store each rule for rule-sets of size 20 K to 100 K. As a result, the proposed data structure does not incur any exponentially increased memory. Moreover, the average required memory of the proposed data structure is only 67 Bytes for rule-sets with the size of up to 100 K rules that means it supports high scalability from the memory aspect. As a comparison, the proposed approach outperforms the DBAMCP, BSOL-RC and BF-AQT more than (1.5×) times. Comparing to our previous works, in [26] the R-tree have been built using SIP & DIP addresses. It stores the whole rule-set in every level in the R-tree, and integrating Cuckoo filters to store the remaining fields of the rules causes the proposed approach achieves better memory utilization. In [27] the R-tree has been used instead of the X-tree in the proposed approach, using the supernodes in X-trees makes X-trees to have less tree levels than R-trees. Less tree levels means less memory consumption.

5.4 Classification accuracy

Using the Cuckoo filter in the proposed system gives a small false positive probability. In addition, it does not provide false negatives. In other words, when querying

Table 4 False positive rate comparison

Approach	False positive probability				False positive
	1 K	10 K	100 K	1 M	
BF based X-tree	2	24	231	2137	0.00215
CF based X-tree	0	2	14	158	0.000156

such data structures, the result will be either "possibly in the set" or "definitely not in set". To hash n elements to a Cuckoo filter, if k is the number of hash functions, m is the number of bits in the filter array, f is the fingerprint bits, and b is bucket size, then:

$$\text{CF false positive probability} = 1 - \left(1 - \frac{1}{2^f}\right)^{2b} \simeq \frac{2b}{2^f} \tag{5}$$

As can be seen from CF false positive probability equation (Eq. 5) that the probability of false positive is decreased as f is increased (the number of bits in the fingerprint), and is increased as b is increased (the bucket size). Due to the use of Cuckoo filter in each X-tree level, the false positive rate cannot be theoretically calculated because that the X-tree is dynamically originated and the size of Cuckoo filter is variable at each level. The false positive probability has been calculated experimentally using proposed classifier. It uses a variable number of the incoming packets that have been generated especially to match the existing rules, and then the ratio between the incoming packets that have been classified as expected and those that failed to find its corresponding rules. The resulted false positive probability has been compared against the false positive rate when integrating the Bloom filter in our proposed classifier instead of Cuckoo filter. The false positive probability of Bloom filter is:

$$\text{BF false positive probability} = (1 - e^{\frac{-kn}{m}})^k \tag{6}$$

As can be seen from BF false positive probability equation (Eq. 6) that the probability of false positives decreases as m increases, and increases as n and/or k increases. In the experimental implementation, the size of fingerprints f in CF is set to 2 Byte in average depending on the size of rule-set and the X-tree node level, bucket size b to 4 slots. The BF array size m and the number of hashed elements in each BF n are set to variable size depending on the corresponding X-tree node level, and four hashing function are used k.

The results show that the proposed classifier has less false positive rate when using the Cuckoo filter as can be seen in Table 4. The proposed solution has false positive rate $15 * 10^{-5}$. This rate includes false positive from all levels and the X-tree itself. Noted that selecting high-quality hashing function and appropriated bucket length for the filter decrease the false positive rate. We have used simple hashing functions such as (Murmurhashing2 [31], and CRC32 [32]). CF has very small false positive rate because it uses the reallocation technique that prevents inserted elements collision.

6 Conclusion

In this work, we proposed a new data structure that called CX-tree. The CX-tree is an X-tree that includes Cuckoo filters in its nodes. The proposed CX-tree data structure is used in many-field packet classification as an important task in software-defined networking. The X-tree needs more memory than the original rule-sets themselves. Hence, we integrate an approximated membership query data structure named Cuckoo filter in the X-tree nodes as a query data structure. Cuckoo filters require a smaller querying time comparing with other approximated membership query data structures. Cuckoo filter had been used to increase the classification throughput as well as to decrease the memory requirements. We concluded from the results that the proposed data structure decreases the required memory to store the rule-set to less than half compared to our previous work [26], which stored the rule-sets directly in R-tree without the use of Cuckoo filters. We have also concluded that the proposed method has reduced the processing latency by half in compared to our previous work AMQ-R-tree [27]. As well as the proposed approach has outperformed the other compared many-field packet classification algorithms from classification throughput, processing latency, and memory consumption aspects.

References

1. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J (2008) OpenFlow: enabling innovation in campus networks. SIGCOMM Comput Commun 38(2):69–74
2. Specification, OpenFlow Switch 1.4 (2019) https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf. Accessed 1 March 2019
3. Ganegedara T, Jiang W, Prasanna VK (2014) A scalable and modular architecture for high-performance packet classification. IEEE Trans Parallel Distrib Syst 25(5):1134–1144
4. Walia GS, Kapoor R (2013) Particle filter based on cuckoo search for non-linear state estimation. In: Advance computing conference (IACC), 918–924
5. Berchtold S, Keim DA, Kriegel HP (1996) The X-tree: an index structure for high-dimensional data. In: Proceeding VLDB '96 proceedings of the 22th international conference on very large data bases, pp 28–39
6. Hu T, Guo Z, Yi P, Baker T, Lan J (2018) Multi-controller based software-defined networking: a survey. IEEE Access 6:15980–15996
7. Yu C, Lan J, Guo Z, Hu Y, Baker T (2019) An adaptive and lightweight update mechanism for SDN. IEEE Access 7:12914–12927
8. Kendrick P, Baker T, Maamar Z, Hussain A, Buyya R, Al-Jumeily D (2018) An efficient multi-cloud service composition using a distributed multiagent-based. Memory-driven approach. IEEE Trans Sustain Comput Early Access
9. Liu AX, Meiners CR, Torng E (2016) Packet classification using binary content addressable memory. IEEE/ACM Trans Biol Cybern 24(3):1295–307
10. Taylor DE (2005) Survey and taxonomy of packet classification techniques. J ACM Comput Surv 37(3):238–275
11. Lakshman TV, Stiliadis D (1998) High-speed policy-based packet forwarding using efficient multi-dimensional range matching. ACM SIGCOMM Comput Commun Rev 28(4):203–214
12. Gupta P, McKeown N (1999) Packet classification on multiple fields. ACM SIGCOMM Comput Commun Rev 29(4):147–160
13. Taylor D, Turner J (2005) Scalable packet classification using distributed crossproducing of field labels. INFOCOM 2005. In: Proceedings of the 24th annual joint conference of the IEEE computer and communications societies, 269–280

14. Jiang W, Prasanna VK (2009) Field-split parallel architecture for high performance multi-match packet classification using FPGAs. In: Proceedings of the twenty-first annual symposium on parallelism in algorithms and architectures, pp 188-196
15. Dong X, Qian M, Jiang R (2018) Packet classification based on the decision tree with information entropy. J Supercomput, pp 1–15
16. Singh S, Baboescu F, Varghese G, Wang J (2003) Packet classification using multidimensional cutting. In: Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications, pp 213–224
17. Wee J-H, Pak W (2017) Fast packet classification based on hybrid cutting. IEEE Commun Lett 21(5):1011–1014
18. Lim H, Byun HY (2015) Packet classification using a bloom filter in a leaf-pushing area-based quad-trie. In: Proceedings of the eleventh ACM/IEEE symposium on architectures for networking and communications systems, 183–184
19. Varvello M, Laufer R, Zhang F, Lakshman TV (2016) Multilayer packet classification with graphics processing units. IEEE/ACM Trans Netw 24(5):2728–2741
20. Lu H, Sahni OS (2007) O(logW) multidimensional packet classification. IEEE/ACM Trans Netw 15(2):462–472
21. Cheng Y-C, Wang P-C (2015) Packet classification using dynamically generated decision trees. IEEE Trans Comput 64(2):582–586
22. Qu YR, Zhou S, Prasanna VK (2015) A decomposition-based approach for scalable many-field packet classification on multi-core processors. Int J Parallel Program 43(6):965–987
23. Warkhede P, Suri S, Varghese G (2004) Multiway range trees: scalable IP lookup with fast updates. Comput Netw 44(3):289–303
24. Zhong P (2011) An IPv6 address lookup algorithm based on recursive balanced multi-way range trees with efficient search and update. In: Proceedings of the international conference on computer science and service system (CSSS), 2059–2063
25. Pagh R, Rodler FF (2001) Cuckoo hashing. Springer, Berlin
26. Abdulhassan A, Ahmadi M (20017) Parallel many fields packet classification technique using R-tree. In: New trends in information and communications technology applications (NTICT), 274–279
27. Abdulhassan A, Ahmadi M (2018) Many-field packet classification using AMQ-R-tree. J High Speed Netw 24(3):219–241
28. Gupta V, Breitinger F (2015) How cuckoo filter can improve existing approximate matching techniques. In: International conference on digital forensics and cyber crime, vol 157, pp 39–52
29. Taylor DE, Turner JS (2007) ClassBench: a packet classification benchmark. IEEE/ACM Trans Netw 15(3):499–511
30. Kaler T, Cache efficient bloom filters for shared memory machines. https://pdfs.semanticscholar.org/465b/0d7872764bba2a6daadffa9a169d056b8c7c.pdf
31. Tanjent (tanjent) (2018) MurmurHash first announcement. https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/MurmurHash.html. Accessed 1 March 2019
32. Mitra J, Nayak T (2017) Reconfigurable very high throughput low latency VLSI (FPGA) design architecture of CRC 32. Integr VLSI J 56:1–14