IEEE Open Journal of the Computer Society

Received 15 October 2022; accepted 30 October 2022. Date of publication 4 November 2022;
date of current version 30 November 2022. The review of this paper was arranged by Associate Editor Tuo Shi.

Digital Object Identifier 10.1109/OJCS.2022.3219631

Towards Capacity-Adjustable and Scalable Quotient Filter Design for Packet Classification in Software-Defined Networks

MINGHAO XIE [1], QUAN CHEN [1] (Member, IEEE), TAO WANG [1], FENG WANG [2], YONGCHAO TAO [3], AND LIANGLUN CHENG [1]

[1]School of Computer Science and Technology, Guangdong University of Technology, Guangzhou 510006, China
[2]School of Information Engineering, Guangdong University of Technology, Guangzhou 510006, China
[3]Shenzhen Academy of Aerospace Technology, Shenzhen 518000, China

CORRESPONDING AUTHOR: QUAN CHEN (e-mail: quan.c@gdut.edu.cn)

This work was supported in part by the R&D Projects in Key Areas of Guangdong Province under Grant 2020B010164001, in part by the Key Program of NSFC-Guangdong Joints Funds under Grants U1801263 and U2001201, in part by the Projects of Science and Technology Plan of Guangdong Province under Grants 2022A1515011032, 2020A1515011132, and GDNRC[2020]024, in part by the Industry University Research Innovation Fund of Chinese Universities under Grant 2021FNA02010, and in part by the Guangdong Provincial Key Laboratory of Cyber-Physical System under Grant 2020B1212060069.

ABSTRACT Software defined networking (SDN), which can provide a dynamic and configurable network architecture for resource allocation, have been widely employed for efficient massive data traffic management. To accelerate the packet classification process in SDN, the hash-based filters which can support fast approximate membership query have been widely employed. However, the existing Quotient Filters are limited to fixed size and the number of elements has to be provided in advance. Thus, in this paper, we investigate the first capacity adjustable and scalable quotient filter for dynamic packet classification in SDN. Firstly, a novel Index Independent Quotient Filter (IIQF) is designed, which can adjust its capacity in a more precise level to support dynamic set representation. The algorithms for the operations of insertion, querying, deletion and capacity adjustment of IIQF are also given. Secondly, on the basis of IIQF, a Scalable Index Independent Quotient Filter (SIIQF) is designed to ensure the consistency of the designed quotient filter when adjusting its size. The theoretical performance of the proposed SIIQF, including the error rate, probability of collisions, and the time and space complexity are all analyzed. An instance of employing SIIQF for packet classification with tuple space searching algorithm is also introduced. Finally, the extensive simulations demonstrate the performance gains achieved by the proposed SIIQF compared with the baseline methods.

INDEX TERMS Capacity adjustable and scalable hashing, dynamic set representation, quotient filter, software defined network.

I. INTRODUCTION

Currently, the Software Defined Networking (SDN) technology, which can provide a dynamic and configurable network resource allocation scheme for supporting massive data traffic, has been widely used in data centers, cellular networks, Internet of things, etc [1], [2], [3]. Compared to the conventional network infrastructure, a set of switches is used in SDN for packet detection, classification, forwarding, etc., which are managed by a centralized controller with the OpenFlow protocol [4]. For packet classification, the headers of the packets are usually parsed to extract the relevant fields to compare with a set of rules in a flow table managed by the switches to find and apply the corresponding actions (i.e., forwarding rules) for such a packet. To accelerate the packet classification process, a smaller filter stored in a faster memory, which can support fast approximate membership query while the false positive probability is guaranteed, is widely employed to reduce the number of accesses to the full tables which are stored in a slower memory [5], [6].

As for filtering for packet classification, there are three widely-used filters designed for approximate membership query, i.e., Bloom Filter [7], Quotient Filter [8] and Cuckoo

Filter [9]. In Bloom Filter, a fixed-length array of bits is maintained and are initialized as 0 at the beginning. When coming an element, k independent hash functions are used to map the element to k positions in the array. After that, the bits in corresponding positions are set to 1 s. To query the membership of an element, one only needs to check whether all the bits in computed hash positions are all 1 s. If not, the queried element is definitely not in the set. Otherwise, the element belongs to the set with a high probability. Different from Bloom Filter, Cuckoo Filter maps the element to fingerprints and employs an array of buckets to store the fingerprint with the partial cuckoo hashing strategy [10]. There are two candidate buckets for each element, and it tries to store the fingerprint into one of the candidate buckets. If the fingerprint of the queried element is found in any of the two candidate buckets, then the element is in the set with a high probability.

Compared to Bloom filter and Cuckoo Filter, Quotient Filter partitions the fingerprint into two parts. The first part is used to decide the position to store the element, and the second part is the actual stored content in the position. Additionally, three extra bits are designed to assist in correctly inserting, querying and deleting the elements in the set. Note that, different from the Bloom filter and Cuckoo Filter, where several hash functions are required, only one hash function is used in Quotient Filter. On the other hand, Quotient Filter is more cache-friendly than Bloom Filter and Cuckoo Filter [11]. These enable it to have a much higher efficiency for inserting and querying, especially for the scenario with stringent latency requirements, such as deep packet inspection system [12] and node authentication [13]. To improve the efficiency of Quotient Filter, there have been several designs, such as Counting Quotient Filter [11], Quotient-based Cuckoo Filter [14], Dynamic Quotient Filter [15], Streaming Quotient Filter [16], etc.

However, all these quotient filters are designed with a fixed length, and the number of elements has to be provided in advance to determine the filter parameters. In SDN, the flow tables are usually dynamic changing, and the elements for packet classification are frequently inserted or removed from the filter. This requires the Quotient Filter to support membership query while the size of the set is changing. Meanwhile, to utilize the limited space in the switch efficiently, the capacity of the Quotient Filter should be adjustable.

To address the above issues, this paper proposes a capacity adjustable and Scalable Index Independent Quotient Filter for dynamic packet classification in SDN. Firstly, a novel Index Independent Quotient Filter (IIQF) is designed, which can adjust its capacity in a more precise level to support dynamic set representations. Secondly, a Scalable Index Independent Quotient Filter (SIIQF) is designed to handle the hard collision problem of the designed quotient filter and better respond to the dramatic growth of the size of the set. Our main contributions are listed as follows:

1) We propose the first capacity-adjustable Quotient Filter for packet classification in SDN, denoted as Index Independent Quotient Filter (IIQF), which employs a hash ring to arrange the bucket rows. The algorithms for insertion, query, deletion and capacity adjustment of IIQF are also given under such data structure.
2) On the basis of IIQF, a Scalable Index Independent Quotient Filter (SIIQF) is proposed, which can further expand the capacity to deal with the growth of the size of the set. Additionally, the method to handle the hard collision problem in IIQF is also given.
3) We theoretical analyze the performance of the proposed IIQF and SIIQF, including the time complexity, space complexity, error rate, probability of collisions and false positive rate.
4) Through extensive simulations, we demonstrate the high performance of the proposed SIIQF.

The rest of this paper is organized as follows. Section II introduces the related works. Section III presents the preliminaries and the definition of the problem. Section IV introduces the detailed design of the proposed IIQF and SIIQF. Section V discusses the theoretical analysis of the proposed data structures. Section VI describes the structure of the packet classification mechanism with the proposed SIIQF. The experimental results and the conclusion of this paper are shown in Sections VII and VIII, respectively.

II. RELATED WORKS

The Quotient Filter for set representation has been studied in the literature [8], [11], [12], [13], [14], [15], [16]. It is firstly proposed in [8], where each element is mapped to a p-bit fingerprint and the first few bits of the fingerprint are used to decide the position to store the rest bits of the fingerprint. In addition, three additional bits are designed in each bucket of the Quotient Filter. The bits are used to assist in inserting, querying and deleting the elements with a low false positive rate. To improve the efficiency of Quotient Filter, [11] proposed Counting Quotient Filter, which embeds variable-sized counters into the Quotient Filter, which can count the number of occurrences of elements in the set. In addition, the Counting Quotient Filter has a higher space efficiency than [8]. The authors in [14] proposed Quotient-based Cuckoo Filter (QCF) by combining Quotient Filter with Cuckoo Filter, which requires the same space with Cuckoo Filter but employs only two hash functions. It makes QCF faster than the Cuckoo Filter, and the authors employ QCF as an identification tool in the deep packet inspection system. In [15], the authors employed a multilevel structure for Quotient Filter and proposed a Concurrent Quotient Filter to reduce memory usage. The authors in [16] re-design the structure of Quotient Filter and propose Streaming Quotient Filter (SQF), which can detect the duplication in streaming data and avoid the time-consuming bit judgement process. But the random eviction mechanism of SQF brings false negative problem, which limits the practicability of SQF. The authors in [12] employ Quotient Filter for matching check in a deep packet inspection system. The authors in [13] proposed a privacy-preserving Quotient Filter to address the node authentication mechanism in vehicular networks. However, all these filters are designed with a fixed

length, which cannot utilize the limited space in the switch efficiently.

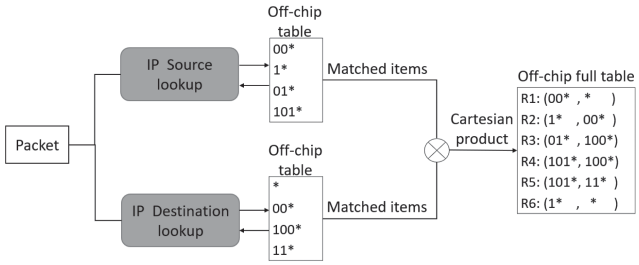
Besides Quotient Filter, Bloom Filter and Cuckoo Filter are two other probabilistic data structures used to accelerate the packet classification in SDN. There have been multiple variants of Bloom Filters [17], [18], [19], [20], [21] considering the static scenarios where there are no new elements. For the dynamic scenarios, the authors in [22] proposed Counting Bloom Filter (CBF), which supports deletion at the expense of accuracy and consumption of space. The authors in [23] proposed the Scalable Bloom filter (SBF), which maintains multiple Bloom Filters to dynamically adapt to the number of elements while controlling the false positive probability. Different from SBF, the authors in [24] proposed a Dynamic Bloom Filter (DBF), which maintains multiple homogeneous Bloom Filters, where the newly incoming element is inserted into an available Bloom Filter. And there is a union operation to merge two sparse Bloom Filters to increase the space efficiency.

Different from Bloom Filter, Cuckoo Filter first uses a hash function to generate the fingerprint of the element and then uses another hash function to derive out the two candidate buckets to store the fingerprint. If both the buckets are full, a randomly chosen fingerprint in the buckets will be kicked out. Then the victim will be moved to its the other candidate bucket. Cuckoo Filter supports deletion and constant-time query. Many efforts have been made to improve Cuckoo Filter [25], [26], [27], [28], [29], [30], [31], [32]. To adapt Cuckoo Filter to dynamic situations, [27] proposes Dynamic Cuckoo Filter (DCF), which supports elastic capacity and applies it to chunk de-duplication in file backup system. Likes the DBF, DCF also maintains multiple Cuckoo Filters. When the cardinality of the set varies, DCF will add or remove Cuckoo Filter to adapt to the cardinality. [28] employs a consistent hash ring and decouples the dependency between the length of the filter and indices of buckets. In this way, it can add or remove bucket instead of a whole filter. Additionally, [29] introduces Configurable-Bucket Cuckoo Filter (CBCF), which can configure the slot number in the spare bucket to reduce the false positive rate. However, Cuckoo Filter is not so cache-friendly as Quotient Filter [11]. And when the number of elements reaches to saturation, the insert time of Cuckoo Filter will heavily increase.

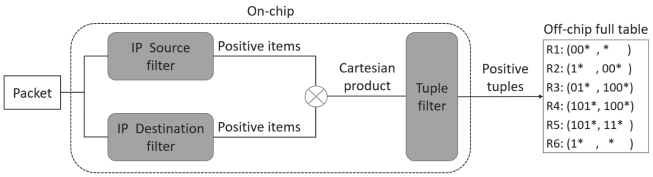
III. PRELIMINARIES AND PROBLEM FORMULATION

A. FILTERING FOR PACKET CLASSIFICATION

In SDN, the switches usually maintain a flow table to store the corresponding rules/actions for each data flow. Since there are so many entries in the flow table, the looking up time will be greatly large, considering the massive data flows. To accelerate the search time, the probabilistic data structure has been widely used to filter out unnecessary query. For example, Fig. 1 shows an example of employing a filter for Tuple Space Search (TSS) [33], which solves the prefix search problem of IP addresses and provides high search performance in



(a) The Tuple Space Search (TSS) algorithm.



(b) The Tuple Space Search with filter.

FIGURE 1. Speeding up Tuple Space Search with filtering.

packet classification. For simplification, the lengths of both IP addresses are assumed to be 4-bit. Let P_t denote the flow table composed of the IP source and destination prefix pairs, which need to be identified, which is set P_t ={R1(00*, *), R2(1*, 00*), R3(01*, 100*), R4(101*, 100*), R5(101*, 11*), R6(1*, *)}. Note that, 00$*$ means the IP address whose has the prefix "00," including "0000," "0001," "0011," etc. Let P_1 and P_2 be the set of source and destination prefixes, respectively, i.e., P_1 ={00*, 1*, 01*, 101*}, and P_2 ={*, 00*, 100*, 11*}. Note that, P_1, P_2 and P_t may be much larger and usually stored in the off-chip memory.

When a packet with a source and destination address pair (0100, 1001) comes, it will query the source and destination IP, respectively. As for the source IP address (0100), since the length of the IP prefixes in table P_1 is {1, 2, 3}, then it will query the 0*, 01* and 010* in P_1. And finally get the 01*. In the same way, it can get the possible destination prefixes, which are * and 100*. Then, the query results are merged into query tuples, i.e., (01*, *) and (01*, 100*), and will be queried in P_t. Finally, it gets the best matching tuple R3. Since the queries in P_1 and P_2 need to access the slow off-chip memory, which heavily degrades the search efficiency. Suppose each query requires one access to the off-chip memory. There are $3 + 2 + 2 = 7$ accesses to the off-chip memory in the above example.

To accelerate the searching phase, small but fast filters have been employed to reduce the access times to the off-chip memory [34]. For example, as in Fig. 1(b), which shows how filters are used for TSS. Initially, the sets P_1, P_2 and P_t are mapped to the filter IP Source filter, IP Destination filter and Tuple filter, which are stored in the on-chip memory. Note that, one can just merge the prefix of the source and destination IP to construct the tuple filter. As for the above example, the query in P_1, P_2 and the tuple filter can both be queried in

a constant time. Note that, although the filter is much more efficient, there may exist a query which does not belong to the set is identified (i.e., false positive). In the above example, while querying the source address, the IP Source filter may report the query 01* and 010* are positive, although 010* is actually not in the table P_1. In this case, for tuples, i.e., (01*, *), (01*, 100*), (010*, 10*), (010*, 100*), need to be queried in the Tuple filter and full table. Although the filters may produce some false positives, they can be excluded by subsequent queries, and there is only one access to the off-chip memory.

Since the flow table is usually dynamic, the rules/actions may be added into or removed from the flow table frequently. So the filter which is used in packet classification should support dynamic insertion and deletion of the elements in the represented set. On the other hand, the switches have limited on-chip memory resources, so it is also essential for the filter to be cache-friendly and support capacity adjustment.

In this paper, the Quotient Filter, which can support correctly inserting, querying and deleting the elements in the represented set, is employed. Note that, different from Bloom filter and Cuckoo Filter, where several hash functions are required, only one hash function is used in Quotient Filter. Although there have been many Quotient Filter designs, the existing quotient filters are limited with a fixed length, and the number of elements has to be provided in advance to determine the filter parameters, which makes it not suitable for the dynamic changing scenarios in packet classification for SDN. Thus, we try to propose a capacity adjustable and Scalable Index Independent Quotient Filter for dynamic packet classification in SDN.

B. PROBLEM FORMULATION AND QUOTIENT FILTER

Define S as a set including a sequence of elements that need to be filtered, i.e., $S = \{e_1, e_2, \ldots, e_N\}$, where N is the cardinality of the set. For each element e_i, it belongs to a finite universe set U. Thus, the filtering problem is defined as: Given a set $S = \{e_1, e_2, \ldots, e_N\}$ and a query element e_q, designing a probabilistic data structure (filter) which can check whether e_i is in S in a very short time. Note that, when the query element e_q belongs to S, i.e., $e_q \in S$, the filter should return True. Otherwise, if ($e_{query} \notin S$), the filter should return false with a high probability. Additionally, considering the dynamic feature of the flow table, the set S is dynamic changing here, which means S can add new elements or remove existing elements. And the proposed filter should alter its capacity according to the N. Table 1 shows the symbols used in this paper.

Before introducing the proposed method, we first introduce the basic Quotient Filter [8]. The Quotient Filter employs a hash function H to map all the elements in the universe U to a p-bit fingerprint f. The fingerprint is divided into two parts: remainder and quotient. The r least-significant bits are the remainder f_r, $f_r = f \bmod 2^r$. The rest $q = p - r$ most-significant bits are the quotient f_q, $f_q = \lfloor f/2^r \rfloor$. Both f_r and f_q can be acquired by an interception with f. Then Quotient

TABLE 1. Symbols and Descriptions

Symbol	Description
S	the represented dynamic set
e	the element in S
N	the cardinality of the set
H	the hash function
U	the finite universe set
f	the fingerprint of e
p	the length of fingerprint
f_r	the remainder of f
r	the length of the remainder
f_q	the quotient of f
q	the length of the quotient
k	the number of buckets in a row
r_i	the row whose index is i
M	the median of the offsets in a row
$offset_i$	the value of the offset in r_i
s	the number of IIQFs in SIIQF
$IIQF_i$	the i-th IIQF in SIIQF
m_i	the number of rows in $IIQF_i$
fp_i	the false positive probability of $IIQF_i$
n_i	the number of elements in $IIQF_i$
Ψ	the number of elements in a row
Φ	the number of elements have the same f_q

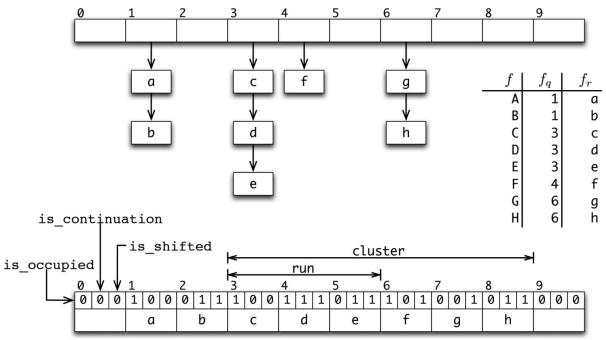


FIGURE 2. An example of quotient filter.

Filter maintains an array of slots A whose length is 2^q. As shown in Fig. 2, the length of each slot is $r + 3$ bits, which are used to store the remainder and three metadata bits.

When inserting an element e in Quotient Filter, it will store the corresponding f_r into slot $A[f_q]$. If the slot is not empty, which means there is a fingerprint having the same quotient. It is referred to as soft collision, which can be handled by linear probing. In this case, the f_r will be stored in a subsequent and empty slot. Meanwhile, Quotient Filter maintains a invariant that if $f_q < f'_q$, f_r is stored before f'_r. So the fingerprints with the same quotient will be stored in continuous slots, which are referred to as a *run*. Continuous runs are referred to as a *cluster*.

In the process of querying or deleting e, Quotient Filter first locates the slot $A[f_q]$. Then it searches forwards to find the beginning of the cluster and searches backwards to find the element's run with the assistance of three metadata bits in Quotient Filter. If f_r exists in the run, Quotient Filter considers e having been stored in the structure. Otherwise, e is not in the set S.

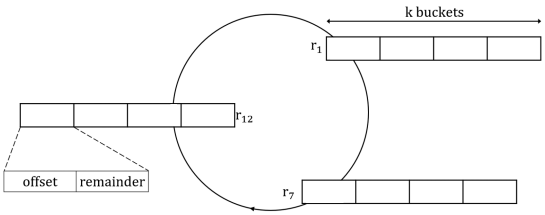


FIGURE 3. Structure of IIQF. $k = 4$, $q = 4$, indexes are 1, 7 and 12, which are not continuous but in a range from 0 to $2^4 - 1$.

The false positive of Quotient Filter occurs only when two elements generate the same fingerprint [8]. Considering an element $e \in S$ with a fingerprint f. If we query another element $e' \notin S$ having the same fingerprint f. Suppose n elements have been inserted in the structure. The probability of false positive is proved to be [8] :

$$1 - \left(1 - \frac{1}{2^p}\right)^n \approx 1 - e^{-n/2^p} \leq \frac{n}{2^p} \leq \frac{2^q}{2^p} = 2^{-r} \quad (1)$$

In the following, we will introduce a capacity adjustable and Scalable Index Independent Quotient Filter for dynamic packet classification in SDN.

IV. DESIGN OF SCALABLE INDEX INDEPENDENT QUOTIENT FILTER

Most existing filters have a strong relationship between the element and the position where the element is stored in the filter. For example, in Cuckoo Filter, the fingerprint of the element is stored in the position whose index is related to the calculated hash values [9]. Quotient Filter requires the fingerprint to be stored in the slot whose index is near the quotient [8]. This relationship limits the ability of the capacity adjustment of these filters. Thus, in this paper, we try to decouple this strong dependency between the fingerprint and index of unit (bucket or slot) which store the fingerprint, which is named as the Index Independent Quotient Filter (IIQF).

In this section, we first introduce the design and the algorithms of IIQF, which is capacity-adjustable and supports insertion and deletion on the run. Then, based on IIQF, we introduce Scalable Index Independent Quotient Filter (SIIQF) to avoid remapping all the elements when the size is changed.

A. INDEX INDEPENDENT QUOTIENT FILTER (IIQF)

Let U denote the universe of all the elements, IIQF employs a hash function $H : U \rightarrow [0, 2^p - 1]$ to map an element e into a p-bit fingerprint f, which is a bit array to identify the element. The r least-significant bits of f are extracted as the remainder. Let f_r denote the remainder, i.e., $f_r = f \bmod 2^r$. The left q ($q = p - r$) most-significant bits are extracted as the quotient, which is denoted as f_q, i.e., $f_q = \lfloor f/2^r \rfloor$. The structure of IIQF is shown in Fig. 3. IIQF consists of multiple rows, and each row has a unique index. Let r_i denote the row whose index is i. In the whole IIQF, the values of the indexes do not have to be continuous but must range from 0 to $2^q - 1$ ($\forall i, i \in [0, 2^q - 1]$). So the number of rows in an

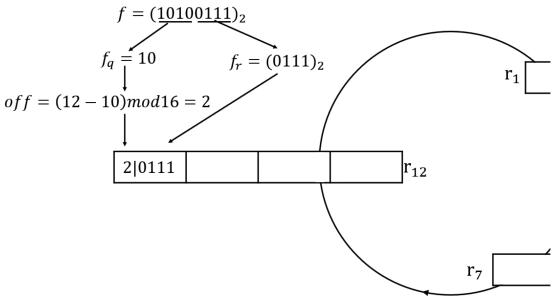


FIGURE 4. An example of inserting 10100111 to the IIQF. $p = 8$, $q = 4$, $r = p - q = 4$.

IIQF can vary from 1 to 2^q. For a fingerprint f with f_q, f is designated to be stored in the row whose index is closest to f_q. Let I denote the set of all the indexes in IIQF, and I^l denote the set of indexes which are larger than f_q. Assuming a fingerprint f is stored in the row r_i, the index i is given by:

$$i = \begin{cases} \min\limits_{j \in I^l} j & I^l \neq \emptyset \\ \min\limits_{j \in I} j & I^l = \emptyset \end{cases} \quad (2)$$

In this way, fingerprints are stored in the nearest row in a clockwise order, which behaves like a ring structure. Each row is further divided into k buckets. The bucket is the basic unit to store a fingerprint. Each bucket has two fields: remainder field and offset field. The remainder field is used to store the f_r of f. And the offset field is used to store the difference between f_q and the index of the row. With the assistance of the offset field, the fingerprint f does not have to be stored in the f_q row (*i.e.*, r_{f_q}). The relationship between f_q and the index of the row is decoupled. In other words, the fingerprint is independent of the index of the row to store the fingerprint. IIQF can smoothly add or remove rows without incurring any influence on the stored fingerprints. An inserted element can still be correctly queried after the capacity of the filter is adjusted.

In the following, we will show how to conduct the operations of insert, split, query, and merge in IIQF.

Insert Operation: Let e denote an element which needs to be inserted in the IIQF. To insert e, IIQF first generates the fingerprint of e. Let f denote the fingerprint of e. Then IIQF chooses the row whose index is nearest to f_q in a clockwise order, which is referred to as the *successor* of f_q. Let i denote the value of the index. If there is an empty bucket, the IIQF will start the store process. It first calculates the difference between the index of the row and f_q, and then stores the result in the offset field of the bucket: $offset = (i - f_q) \bmod 2^q$. Finally, the IIQF stores f_r in the remainder field of the bucket, and the insert operation terminates. The process can ensure the fingerprint f is completely stored in the structure. The detail is shown in Algorithm 1.

Fig. 4 shows an example of element insertion. Let the values of p and q be 8 and 4, respectively. So the value of r is $p - q = 4$. Assume that the fingerprint f of the inserted element is (10100111). So the remainder f_r is (0111), and

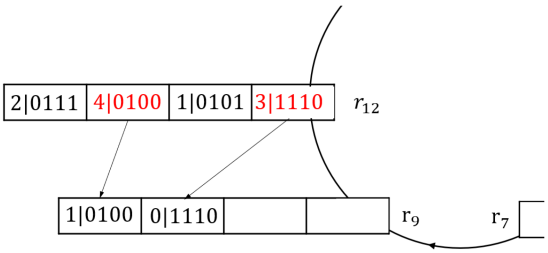


FIGURE 5. An example of releasing row r_{12}, $M = 3$. A new row r_9 was added, two fingerprints were transferred.

Algorithm 1: The Insert Operation in IIQF.

Input: Element e, Hash function H

Generate the fingerprint $f \leftarrow H(e)$;
$f_q \leftarrow \lfloor f/2^r \rfloor$;
$f_r \leftarrow f \bmod 2^r$;
Find the successor r_i of f_q;
for each bucket b in r_i **do**
  if b is empty **then**
    $b.offset \leftarrow (i - f_q) \bmod 2^q$;
    $b.remainder \leftarrow f_r$;
    return True;
  end if
end for
Alert soft collision;
Split(r_i);

the quotient f_q is (1010), which is equal to 10. Based on the (2), the corresponding index is 12, i.e., $i = 12$. So the fingerprint will be stored in row r_{12}. Then IIQF calculates the offset, which is $(i - f_q) \mod 2^q = (12 - 10) \mod 16 = 2$. Finally, the offset 2 and f_r are stored in the offset field and remainder field, respectively. Note that if too many fingerprints are stored in a row, the buckets of the row can be used up, which is referred to as *Soft Collision*. To handle the soft collision problem, we propose a *split operation*, which is shown below.

Split Operation: If the number of elements inserted in a row r_i is larger than k, then IIQF will conduct a split operation to acquire additional space. First, the IIQF checks all the offsets in the row to obtain the median M. Then, a new row with index j ($j = (i - M) \bmod 2^q$) will be added into the IIQF. All the elements in row r_i whose offset is greater than or equal to M will be moved to row r_j. The offsets of the moving elements should be recalculated. Let off_i denote the previous offset and let off_j denote the re-calculated offset, i.e., $off_j = (off_i - M) \bmod 2^q$. The procedure is presented in Algorithm 2.

Fig. 5 shows an example of the split operation. Assume that IIQF is trying to split row r_{12}. The remainders in the r_{12} are (0111), (0100), (0101) and (1110). The offsets are 2, 4, 1 and 3, respectively. First, getting the median M in the offsets, which is 3. Then, a new row is added, and the index of the new row is $12 - M = 9$. In row r_{12}, the contents whose offset is larger or equal to M, i.e., (0100) and (1110), are moved to the

Algorithm 2: The Split Operation in IIQF.

Input: Filled row r_i

Find the median M of the offsets in r_i;
if $M = 0$ **then**
  Alert hard collision;
  Add a new IIQF;
  return False;
else
  $j \leftarrow (i - M) \bmod 2^q$;
  Add a new row r_j to IIQF;
  for each bucket b^i in r_i **do**
    if $b^i.offset \geq M$ **then**
      Find an empty bucket b^j in r_j;
      $b^j.remainder \leftarrow b^i.remainder$;
      $b^j.offset \leftarrow (b^i.offset - M) \bmod 2^q$;
    end if
  end for
  return True;
end if

new row r_9. The new offsets are $(4 - M) \mod 16 = 1$ and $(3 - M) \mod 16 = 0$, respectively. After the split operation, at least a half of the buckets in r_{12} will be released, such that more elements can be inserted in the future.

Query Operation: To query an element e, IIQF first generates the fingerprint f of e. Then, the IIQF finds the row whose index is nearest to f_q in clockwise order. Let i denote the nearest index, and r_i denote the row. Then IIQF goes through all the remainder fields of the buckets in row r_i. If there is a remainder equal to f_r, then the IIQF will calculate the difference between the index i and the corresponding offset to check whether the result is equal to f_q. If so, IIQF considers e has been inserted before and returns true. In other words, if there is a content in a bucket can satisfy the following two requirements: (1) $remainder = f_r$ and (2) $(i - offset) \bmod 2^q = f_q$, then e is an element belong to S. Otherwise, IIQF considers e is not in S and returns false. The detail is presented in Algorithm 3.

Delete Operation: If an element e needs to be deleted from IIQF, e should be queried first to find the fingerprint f of e. If the f is found, then the content in the corresponding bucket will be cleared. Otherwise, the IIQF returns false. Note that, frequent deletions usually incur multiple idle buckets in IIQF. For a higher space utilization, the IIQF performs the *merge operation* to merge two sparse rows and reuses the space of the released row. The merge operation starts with scanning through all the rows to acquire the number of occupied buckets of each row. Let r_i and r_j denote two adjacent rows, and r_j has the larger index. If the sum of occupation numbers of r_i and r_j is less than or equal to k, all the elements in r_i will be moved to r_j. Similar to the split operation, the offsets of the moving elements need to be adjusted. Let off_i denote the old offset and off_j denote the appropriate offset, so $off_j = (off_i + j - i) \bmod 2^q$. After the movements, the

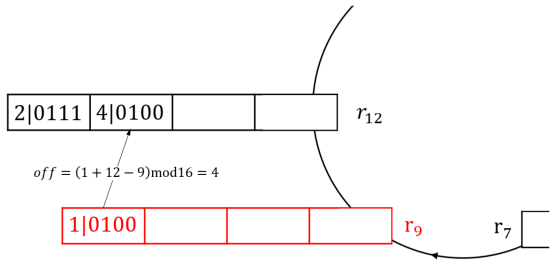


FIGURE 6. An example of merge operation, r_9 was merged into r_{12}.

Algorithm 3: The Query Operation in IIQF.

Input: Element e, Hash function H

Acquire the fingerprint $f \leftarrow H(e)$;
$f_q \leftarrow \lfloor f/2^r \rfloor$;
$f_r \leftarrow f \bmod 2^r$;
Find the successor r_i of f_q;
for each non-empty bucket b in r_i **do**
  if $f_r = b.remainder$ **then**
    if $f_q = (i - b.offset) \bmod 2^q$ **then**
      Report e is found;
      return True;
    end if
  end if
end for
return False;

Algorithm 4: The Merge Operation in IIQF.

Input: Current IIQF

for each row r_i in IIQF **do**
  Find the successor r_j of r_i;
  if $r_i.elements + r_j.elements \leq k$ **then**
    for each bucket b^i in r_i **do**
      Find an empty bucket b^j in r_j;
      $b^j.remainder \leftarrow b^i.remainder$;
      $b^j.offset \leftarrow (b^i.offset + j - i) \bmod 2^q$;
    end for
    Recycle r_i;
  end if
end for

IIQF recycles r_i to remove idle buckets. The pseudo-code is presented in Algorithm 4.

For example in Fig. 6, assume that r_9 is about to being merged into r_{12}, and in r_9, there is a fingerprint whose offset and f_r are 1 and (0100), respectively. First, IIQF calculates the difference between the indexes of two rows, which is $12 - 9 = 3$. Then, the new offset can be calculated as $(1 + 3) \bmod 16 = 4$. Finally, the new offset and f_r, which are 4 and (0100), will be stored in an empty bucket in row r_{12}. Note that, the merge operation adjusts the capacity with only local data movement and can still keep logically consistent of the whole structure.

B. SCALABLE INDEX INDEPENDENT QUOTIENT FILTER

In the proposed IIQF, it employs a hash ring structure to decouple the strong relationship between the quotient of the fingerprint and the row to store the fingerprint. While the IIQF is adding or removing a row, only the content in the adjacent row needs to move. In addition, the length of our filter does not have to be the powers of two. As the set size increases/decreases, the capacity of the IIQF can increase/decrease accordingly. This indicates that the proposed IIQF has a superior dynamic adaptation performance.

However, if there are k fingerprints with the same f_q having been stored in the IIQF, the row r_{f_q} can not conduct a split operation. If IIQF splits r_{f_q}, the index of the new row will be f_q, which violates the uniqueness of the row index. In this situation, if another fingerprint with the same f_q needs to be stored, there is no bucket for this new fingerprint. We refer to this problem as *Hard Collision* and propose a SIIQF to handle it. Its main idea is to maintain multiple IIQFs to resolve the hard collision problem, which does not need to remap all the elements when the size of the represented set is dramatically growing. The main operations of SIIQF are introduced as follows.

Expand Operation: When a hard collision occurs, the SIIQF expands its capacity by adding a new IIQF. Note that, adding a new IIQF to SIIQF can better address the increase in the size of the represented set. In SIIQF, the parameters of IIQFs can be variable. Different IIQFs could have different values of q, which means the range of row index in different IIQFs can be varied. This variety is useful since a longer range can contain more rows. On the other hand, a longer range incurs more space consumption. Also, different IIQFs could have different values of k, which is the number of buckets in a row. IIQF with a larger k can achieve a lower hard collision probability.

Insert Operation: For load balance, the SIIQF try to insert the new element in the IIQF, which has stored the smallest number of fingerprints. In the design, we monitor the number of elements inserted in each IIQF and select the top t IIQFs that have stored the smallest number of fingerprints as the active IIQF set. Let I^a denote the active IIQF set. When a new element e arrives, the SIIQF tries to insert e into one of the IIQFs in I^a one by one. If e is successfully inserted in an IIQF, the insertion of SIIQF will terminate. Otherwise, SIIQF has to add a new IIQF to insert e. The pseudo-code is shown in Algorithm 5.

Shrink Operation: If the set size is relatively small, the SIIQF can shrink its size by merging some sparse IIQFs to reduce space consumption. Due to the different parameter settings of the IIQFs, merging sparse IIQFs becomes a complex process. Let $IIQF_i$ denote the i-th IIQF in SIIQF, and q_i denote the length of the quotient in $IIQF_i$. Suppose a fingerprint f is stored in the row r_j of the $IIQF_i$. The restoration of f is expressed by:

$$f = (j - offset) \bmod 2^{q_i} || remainder \tag{3}$$

Algorithm 5: The Insert Operation in SIIQF.

Input: Element e, Hash function H

Find the top t IIQFs with least element: $\{I_1, I_2, \ldots, I_t\}$;
for each IIQF I in $\{I_1, I_2, \ldots, I_t\}$ **do**
  I.Insert(e, H);
  if Insertion is True **then**
    $I.counter + 1$;
    return True;
  end if
end for
Report an adding event;
Initialize a new IIQF I';
I'.Insert(e, H);
if Insertion is True **then**
  $I'.counter + 1$;
  return True;
end if

Algorithm 6: The Shrink Operation in SIIQF.

Input: Current SIIQF SI_c, Hash function H
Output: Shrunken SIIQF SI_s

$SI_s \leftarrow SI_c$;
while True **do**
  Find the least IIQF I_l in SI_s;
  Create a new SIIQF SI'_s;
  $SI'_s \leftarrow SI_s$ remove I_l;
  for each element e in I_l **do**
    SI'_s Insert(e, H);
    if Insertion is False **then**
      Report SI_s can not shrink anymore;
      return SI_s;
    end if
  end for
  $SI_s \leftarrow SI'_s$;
end while

The operator || means concatenating two bit-arrays. First, the SIIQF checks the number of elements of each IIQF to decide whether to start a merge process. Then, the SIIQF finds the IIQF that stores the smallest number of fingerprints (which we denote as I_l), restores all the fingerprints in I_l and inserts them into the rest of the IIQFs in SIIQF. If all the elements in I_l are successfully reinserted into the other IIQFs, then the merge process is successful, and I_l is finally deleted from the SIIQF. Otherwise, the SIIQF keeps I_l and withdraws the previous reinsertion. Algorithm 6 shows the shrink operation of SIIQF.

Query and Delete Operation: The SIIQF goes through all the IIQFs to query an element e. It first generates the fingerprint f of e. Then SIIQF checks the presence of f in all the IIQFs. If an IIQF returns true, the SIIQF terminates the query process and returns true. Otherwise, the SIIQF considers e is not in the set and returns false. This query process can be implemented in a parallel fashion, which can effectively improve the query performance. The delete operation is similar to the query operation. The SIIQF checks the presence of f in all the IIQFs. If f exists in an IIQF, the content in the corresponding bucket will be removed. Otherwise, the SIIQF considers e is not in the set and reports a failed deletion. After several removal operations, SIIQF will try to merge some sparse IIQFs to shrink the size and achieve a higher space utilization.

Now, the complete SIIQF design and its corresponding operations are introduced.

V. PERFORMANCE ANALYSIS OF SIIQF

In this section, we present the performance analysis for the proposed IIQF and SIIQF, including their error rates, collision probabilities, time and space complexity.

A. ERROR RATE OF SIIQF

For each element stored in a SIIQF, it will never report a false negative. As for false positive, it means there are two elements x and y sharing the same fingerprint. Assume x is inserted earlier. A query of y will return true, although it is not belonged to the set.

Consider a number of s IIQFs in the SIIQF. And each $IIQF_i$ ($i \in [1, s]$) has k_i buckets in a row. Let m_i denote the number of rows of $IIQF_i$. Let the length of the fingerprint be p. If the hash function is perfectly uniform, then the probability of two elements generating the same fingerprint is $1/2^p$. The lower bound of the probability that no fingerprint collision happen in all the k_i buckets is $(1 - \frac{1}{2^p})^{k_i}$. So, in an $IIQF_i$, the upper bound of the false positive probability is:

$$fp_i = 1 - \left(1 - \frac{1}{2^p}\right)^{k_i} \approx \frac{k_i}{2^p} \tag{4}$$

As a result, the upper bound of the SIIQF's false positive probability is:

$$fp = 1 - \prod_{i=1}^{s} (1 - fp_i) = 1 - \prod_{i=1}^{s} \left(1 - \frac{1}{2^p}\right)^{k_i} \tag{5}$$

If all the IIQFs are homogeneous, then the simplified upper bound of the false positive probability is given by:

$$fp = 1 - \left(1 - \frac{1}{2^p}\right)^{s \times k} \approx \frac{s \times k}{2^p} \tag{6}$$

B. PROBABILITY OF COLLISIONS

The soft collision occurs when a row is full of fingerprints, which leads to a row increment in an IIQF. Let n_i denote the number of elements that have been inserted in $IIQF_i$. Consider all the fingerprints inserted in IIQF are uniform. Since the size of $IIQF_i$ is m_i, the probability of a fingerprint being inserted into a specific row is given as $\frac{1}{m_i}$. Let $\Psi \in [0, n_i]$ denote the number of elements inserted to a specific row. Consider Ψ follows the binomial distribution, i.e., $\Psi \sim B(n_i, \frac{1}{m_i})$. The

probability that there are ψ elements being inserted in a row is computed as:

$$P(\Psi = \psi) = C(n_i, \psi) \times \left(\frac{1}{m_i}\right)^{\psi} \times \left(1 - \frac{1}{m_i}\right)^{n_i - \psi} \quad (7)$$

Then, the probability of the occurrence of a soft collision, i.e., p_{sc}^{i}, in $IIQF_i$ is obtained as:

$$\begin{aligned} p_{sc}^{i} &= P(\psi > k_i) = 1 - P(\psi \le k_i) \\ &= 1 - \sum_{\psi=0}^{k_i} C(n_i, \psi) \times \left(\frac{1}{m_i}\right)^{\psi} \times \left(1 - \frac{1}{m_i}\right)^{n_i - \psi} \end{aligned} \quad (8)$$

As for hard collision, it means the number of fingerprints with 0 offset exceeds the row size k_i in an IIQF. Assuming all the elements are uniformly distributed. The probability of two fingerprints with the same f_q is $1/2^q$. Let $\Phi \in [0, n_i]$ denote the number of elements generating the same f_q. Then the probability that ϕ elements generate the same f_q is calculated as:

$$P(\Phi = \phi) = C(n_i, \phi) \times \left(\frac{1}{2^q}\right)^{\phi} \times \left(1 - \frac{1}{2^q}\right)^{n_i - \phi} \quad (9)$$

And the probability of the occurrence of a hard collision in $IIQF_i$, i.e., p_{hc}^{i}, is computed as:

$$\begin{aligned} p_{hc}^{i} &= P(\Phi > k_i) = 1 - P(\Phi \le k_i) \\ &= 1 - \sum_{\phi=0}^{k_i} C(n_i, \phi) \times \left(\frac{1}{2^q}\right)^{\phi} \times \left(1 - \frac{1}{2^q}\right)^{n_i - \phi} \end{aligned} \quad (10)$$

When adding an IIQF in SIIQF, it means hard collision occurring in all the top t IIQFs, and its probability can be obtained as:

$$\begin{aligned} p_{add} &= \prod_{i=1}^{t} p_{hc}^{i} \\ &= \prod_{i=1}^{t} \left[1 - \sum_{\phi=0}^{k_i} C(n_i, \phi) \times \left(\frac{1}{2^q}\right)^{\phi} \times \left(1 - \frac{1}{2^q}\right)^{n_i - \phi} \right] \end{aligned} \quad (11)$$

C. TIME COMPLEXITY OF SIIQF

Firstly, we will consider the time complexity of querying in SIIQF. To query an element e, one first must find the successor row r_i. For simplicity, assume the row index in the hash ring is implemented by a binary search tree, and each row index is the node of the tree. In this case, the time complexity of finding row r_i in $IIQF_i$ is $O(\log m_i)$. The query operation of IIQF consists of two stages, *i.e.,* finding the row and finding an empty bucket. So the time complexity of query operation in $IIQF_i$ is $O(\log m_i + k_i)$. In the worst situation, a non-existent element is queried by SIIQF. The SIIQF has to go through all the IIQFs and check all the buckets in a row, the time complexity is $O(\sum_{i=1}^{s}(\log m_i + k_i))$. Let m, k respectively denote the largest m_i, k_i. The time complexity of the query operation can be computed as $O(s \times (\log m + k))$. As for the time complexity of the delete operation, one can see it is actually same to the one of the query operation, i.e., $O(s \times (\log m + k))$.

As for the time complexity of the insert operation, since one needs to find the corresponding row and an empty bucket in the row, then the time complexity of the insert operation in $IIQF_i$ can be computed as $O(\log m_i + k_i)$. In the worst situation, if all the t IIQFs in SIIQF are failed to insert the element, the time complexity of insert operation will be $O(\sum_{i=1}^{t} \log m_i + k_i) = O(t \times (\log m + k))$.

Then, we analyse the time complexity of the split operation in IIQF. Finding the median of the offsets in a full row is the first step of splitting. The divide and conquer algorithm is a fast way to acquire the median in an unordered list. The time complexity is $O(k_i)$, where k_i is the number of the offsets in a row of $IIQF_i$. Then all the k_i offsets are compared with the median, so the time complexity of split operation in $IIQF_i$ is $O(k_i + k_i) = O(k_i)$.

D. SPACE COMPLEXITY OF SIIQF

In $IIQF_i$, the length of a bucket depends on the lengths of the remainder and offset, which are r and q, respectively. In addition, the cost of binary search tree in $IIQF_i$ is $m_i \times q_i$, where m_i is the number of the nodes and q_i is the length of the node. Therefore, the space consumption of $IIQF_i$ is $m_i \times k_i \times (q_i + r_i) + m_i \times q_i = m_i \times (k_i \times p + q_i)$. As for the whole SIIQF, its whole consumed space is computed as:

$$\sum_{i=1}^{s} m_i \times (k_i \times p + q_i) \quad (12)$$

Next, we will investigate the expected space complexity in SIIQF. Let $m_{i,j}$ be the number of rows in $IIQF_i$, where $m_{i,0}$ is the initial size of $IIQF_i$, and $j \in [0, 2^{q_i} - m_{i,0}]$ means the increased number of rows. Apparently, $m_{i,j}$ is computed as:

$$m_{i,j} = m_{i,0} + j \quad (13)$$

Let $p_{sc}^{i,j}$ be the probability of soft collision happen in $IIQF_i$ when increasing j rows. Based on (8), the value of $p_{sc}^{i,j}$ is calculated as:

$$p_{sc}^{i,j} = 1 - \sum_{\psi=0}^{k_i} C(n_i, \psi) \times \left(\frac{1}{m_{i,j}}\right)^{\psi} \times \left(1 - \frac{1}{m_{i,j}}\right)^{n_i - \psi} \quad (14)$$

This indicates that when the size of $IIQF_i$ is $m_{i,j}$, the probability of soft collision is $p_{sc}^{i,j}$. If the final size of the $IIQF_i$ is $m_{i,0}$, there is no soft collision, and the probability is $1 - p_{sc}^{i,0}$. If the final size grows to $m_{i,j}$ $(j > 0)$, the probability is given by:

$$\prod_{k=0}^{j-1} p_{sc}^{i,k} \times (1 - p_{sc}^{i,j}) \quad (15)$$

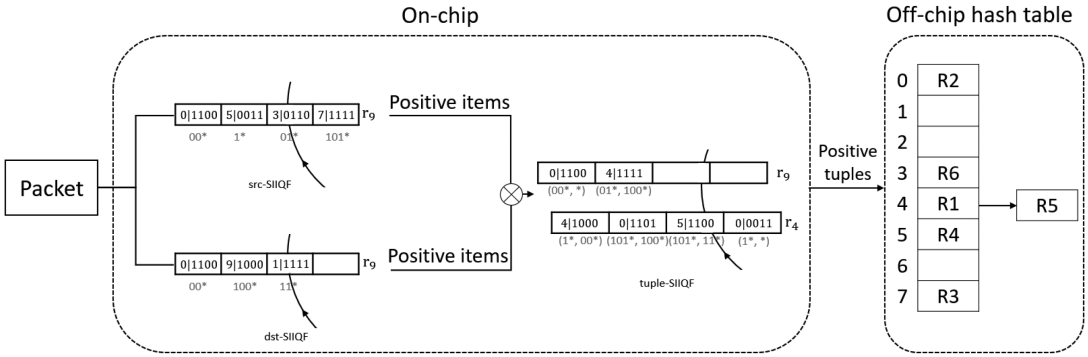


FIGURE 7. The example of employing SIIQF for TSS in packet classification.

Let $E(IIQF_i)$ be the expected space complexity of $IIQF_i$. Then, it can be computed as:

$$E(IIQF_i) = m_{i,0} \times (k_i \times p + q_i) \times (1 - p_{sc}^{i,0}) + \sum_{j=1}^{2^{q_i} - m_{i,0}} m_{i,j} \times (k_i \times p + q_i) \times \prod_{k=0}^{j-1} p_{sc}^{i,k} \times (1 - p_{sc}^{i,j}) \quad (16)$$

With the expected length of $IIQF_i$, we can further compute the space complexity of the whole SIIQF. Assume that SIIQF checks all the IIQFs while insertion. We make some adjustments to (11) as follows:

$$p_{add}^{s} = \prod_{i=1}^{s} \left[1 - \sum_{\phi=0}^{k_i} C(n_i, \phi) \times \left(\frac{1}{2^q}\right)^{\phi} \times \left(1 - \frac{1}{2^q}\right)^{n_i - \phi} \right] \quad (17)$$

where p_{add}^{s} means the probability of adding IIQF while the SIIQF has s IIQFs. With the expected length of each $IIQF_i$, the expected space complexity of the whole SIIQF, i.e., $E(SIIQF)$, is given by:

$$E(SIIQF) = E(IIQF_1) \times (1 - p_{add}^{1}) + \sum_{i>1} \left[\sum_{j=1}^{i} E(IIQF_j) \times \prod_{s=1}^{i-1} p_{add}^{s} \times (1 - p_{add}^{i}) \right] \quad (18)$$

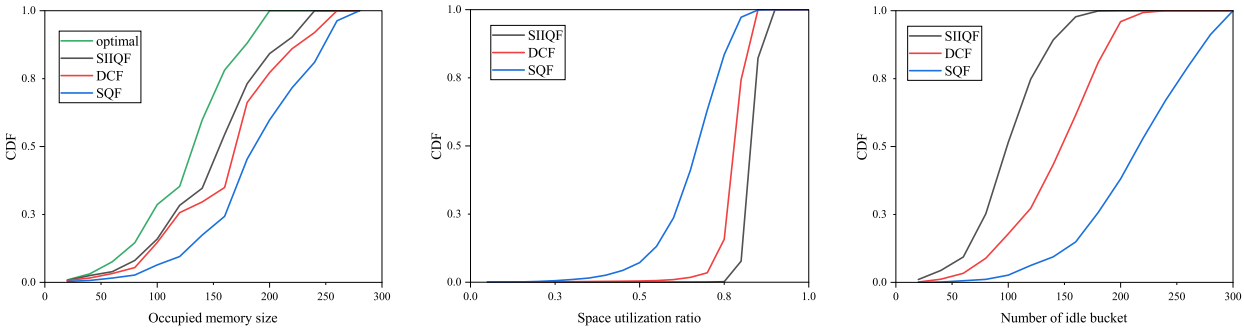
VI. SIIQF BASED TUPLE SPACE SEARCH

In this section, we show how to employ SIIQF for the TSS algorithm. As the example shown in Fig. 7, all the prefixes of the IP source and destination address which need to be identified, are mapped to two IIQFs, i.e., src-SIIQF and dst-SIIQF, respectively. And the source-destination pairs (by combing the prefixes of the IP source and destination addresses) are mapped into a tuple-SIIQF. All the SIIQFs are deployed in the on-chip memory. Table 2 gives an example of the elements and their corresponding fingerprints. Before querying in SIIQF, there is a length check to build the queried element. For example, assuming the possible lengths of the IP source address are 1, 2 and 3, so the first 1-bit, 2-bit and 3-bit of the IP source address are respectively extracted to be queried in the src-SIIQF. The rules in off-chip memory are organized in a hash table with chaining. Let the last three bits of the fingerprint be the hash value to store the rule. For example, in Table 2, the fingerprint of R1 is (10011100). The hash value is (100), which is equal to 4. So R1 is stored in the position whose index is 4.

TABLE 2. Mapping Relationship Between Elements and Fingerprints

Source address			
Element	Fingerprint		Length
	Binary	Decimal	
00*	10011100	156	2
1*	01000011	67	1
01*	01100110	102	2
101*	00101111	47	3
Destination address			
Element	Fingerprint		Length
	Binary	Decimal	
00*	10011100	156	2
100*	00001000	8	3
11*	10001111	143	2
Tuple			
Element	Fingerprint		Length
	Binary	Decimal	
(00*, *)	10011100	156	(2, 0)
(1*, 00*)	00001000	8	(1, 2)
(01*, 100*)	01011111	95	(2, 3)
(101*, 100*)	01001101	77	(3, 3)
(101*, 11*)	11111100	252	(3, 2)
(1*, *)	01000011	67	(1, 0)

As for the query example in Section III-A, we will show how to handle it with SIIQF. Assume the packet with the pair (0100, 1001) is the input, which is the source and destination IP address, respectively. Firstly, the source address (0100) and destination address (1001) will be queried by src-SIIQF and dst-SIIQF, respectively. In the query of source address, since the lengths of the source address prefix are 1, 2 and 3 (Table 2), then the first 1-bit, 2-bit and 3-bit of (0100), which are 0*, 01* and 010*, will be extracted to query in src-SIIQF. One can find that 01* is the positive element, and the possible length of the source address prefix is 2. In the same way, 100* is the positive element in dst-SIIQF, and the possible length is 3. Then, the corresponding tuple (01*, 100*) is built to be queried in tuple-SIIQF. Since the fingerprint of



(a) The CDF of occupied memory size. (b) The CDF of space utilization ratio. (c) The CDF of idle bucket.

FIGURE 8. Comparison between SIIQF, DCF and SQF.

the queried tuple exists in the tuple-SIIQF, (01*, 100*) is a positive element. Then the last three bits of the fingerprint are extracted as the hash value to indicate the position of the rule. Since the hash value is (111), which is equal to 7, the pair (01*, 100*) is directly queried in the 7-th position of the hash table and get the matching rule R3.

VII. EVALUATION

In this section, we evaluate the effectiveness of the proposed SIIQF through extensive simulations. For comparison, the following two baseline algorithms are implemented and evaluated.

Firstly, the existing Streaming Quotient Filter (SQF) [16] is implemented and compared. Note that, SQF employs a hash table and a signature technique to store the element. Note that, in SQF, when a row is full, it will randomly empty a bucket in the row for capacity constraint, which means there exists a false negative rate in SQF.

Secondly, the recently proposed Dynamic Cuckoo Filter (DCF) [27] is also implemented and compared. DCF maintains multiple Cuckoo Filters and utilizes a monopolistic fingerprint for representing an item. So DCF supports dynamic set representation and reliable delete operation.

In the experiments, all the algorithms are run on a PC with a 1.8 Ghz Intel Core 8250 U CPU and an 8 GB DDR4-2133 RAM. The dataset used for comparison is from the WIDE MAWI [35], which records the trace in WIDE from 14:00 to 14:15 on March 31, 2022. There are a total of 80,221,555 packets in the dataset, and we extract the first 15,000 TCP packets for our experiments. The IP source address, IP destination address, source port number, destination port number and the protocol name of the packet are extracted to construct the element. In the experiments, the value of q in SIIQF is 4, and the lengths of one single filter in DCF and SQF are also set 16. For fairness, the k of SIIQF and SQF is set 4, and the number of slots in a bucket in DCF is also set 4. The lengths of the fingerprint of each method are all set 8.

A. PERFORMANCE UNDER THE WIDE MAWI DATASET

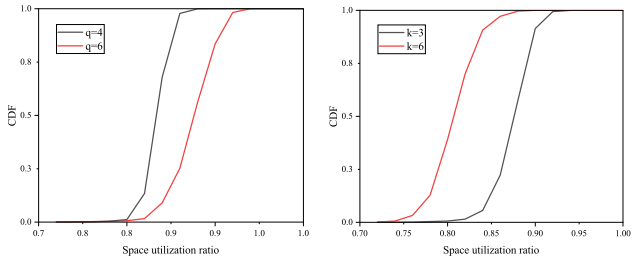
First, we compare the occupied memory size of three methods. Fig. 8(a) shows the cumulative distribution function (CDF) of three different structures. Note that, the optimal required memory size is also compared, which is denoted optimal in Fig. 8(a). One can first observe that the occupied memory size of the proposed SIIQF is closest to the optimal one. This is mainly due to the proposed SIIQF having more precise size modification than DCF and SQF. The maximum occupied memory size of the optimal methods is 199, while the ones of SIIQF, DCF and SQF are 236, 256 and 400, respectively. As one can see, SIIQF requires less space than DCF and SQF to store the same set, which demonstrates its higher space efficiency.

Next, we evaluate the space utilization ratio of SIIQF, DCF and SQF in Fig. 8(b), which is a ratio of the number of elements to the number of buckets/slots in the structure. The space utilization ratio of the proposed SIIQF in the whole process is ranged from 0.825 to 0.850. And the space utilization ratio of DCF and SQF range from 0.775 to 0.800, and from 0.675 to 0.700, respectively. The average space utilization ratio of SIIQF is 0.82913, which is larger than the one of DCF and SQF (The average space utilization ratio of DCF and SQF are 0.77327 and 0.65427, respectively). Additionally, at the early stage, the space utilization ratio of SIIQF can even reach 1, while the highest utilization ratio of DCF and SQF are 0.82813 and 0.86094, which demonstrates the efficiency of the proposed method.

Third, we evaluated the number of idle buckets of these three structures in Fig. 8(c). During the whole process, the number of idle buckets of SIIQF is always lower than 183, while the number of idle buckets in DCF reaches 236, and the one of SQF even exceeds 300. The average number of idle buckets in SIIQF, DCF and SQF are 98, 141 and 257, respectively. Note that, in the experiments, the number of idle buckets in SIIQF is lower than 180 in almost 99.92% time, which demonstrates that SIIQF generates fewer redundant buckets than DCF and SQF in the same situation.

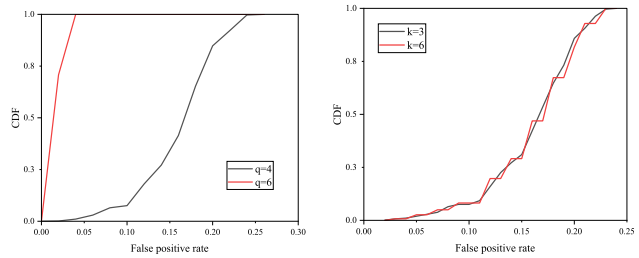
B. PERFORMANCE UNDER DIFFERENT PARAMETERS AND PERFORMANCE

In this group of experiments, we evaluate the performance of the proposed SIIQF under the different values of q and k. The performance of the space utilization ratio, false positive rate, insertion time and query time are evaluated.



(a) The utilization ratio of different q. (b) The utilization ratio of different k.

FIGURE 9. Space utilization ratio of different *q* and *k*.



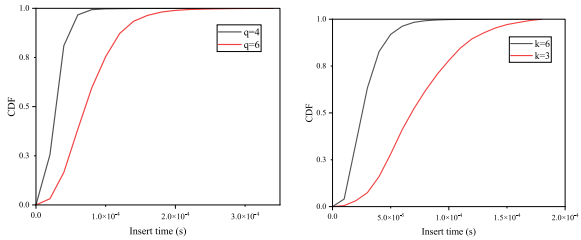
(a) The false positive rate of different q. (b) The false positive rate of different k.

FIGURE 10. False positive rate of different *q* and *k*.

Fig. 9 shows the performance of the space utilization ratio of SIIQF under different q and k. As one can see in Fig. 9(a), when q increases from 4 to 6, the average space utilization ratio also increases from 0.83361 to 0.87478. This is because a larger q in SIIQF will result in a lower probability of collision. Fig. 9(b) shows the space utilization ratio of SIIQF under different k. When k increases from 3 to 6, we can see that the space utilization ratio decreases. Because SIIQF with a larger k will produce more empty space in the splitting and adding process, which leads to a lower space utilization ratio.

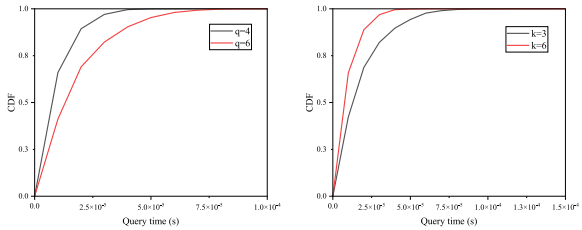
Fig. 10(a) displays the performance of the false positive rate of each method when q is increased from 4 to 6. One can first observe that the false positive rate of the proposed SIIQF decreases as q increases. This is mainly due to a larger q may result in SIIQF implying a longer length of fingerprint, which can significantly decrease the false positive rate of SIIQF. Fig. 10(b) shows the false positive rate of SIIQF under different k. With the increase of k, the false positive rate of SIIQF does not change much. This is because the occurrence of false positive is due to the elements sharing the same fingerprint, which is not relevant to the parameter k.

Fig. 11 shows the performance of the insert time of SIIQF under different q and k. As one can see in Fig. 11(a), when q goes up, the average insert time increases from 2.96×10^{-5}s to 7.69×10^{-5}s. This is because it takes more time to find the corresponding row in the SIIQF under a larger q. Fig. 11(b) shows the insert time of SIIQF when k varies from 3 to 6. When k is increased, SIIQF achieves a shorter insert time. This is due to SIIQF maintaining fewer IIQFs under a larger k. As a result, it can find the positions for the inserted element fast, which helps decrease the insert time.



(a) The insert time of different q. (b) The insert time of different k.

FIGURE 11. Insert time of different *q* and *k*.



(a) The query time of different q. (b) The query time of different k.

FIGURE 12. Query time of different *q* and *k*.

Fig. 12 shows the query time of SIIQF under different q and k. As one can see in Fig. 12(a), the query time increases as q increases. This is because a larger q leads to more rows in an IIQF. In the query process, it will take more time to find the target row. As shown in Fig. 12(b), as k is increased from 3 to 6, we can see the query time decreases. This is because a larger k in SIIQF will result in fewer IIQFs to query.

VIII. CONCLUSION

To accelerate dynamic packet classification in SDN, we investigate the first capacity adjustable and scalable quotient filter in this paper. Firstly, by combining the consistent hash with the Quotient Filter, an Index Independent Quotient Filter (IIQF), which can support precise capacity adjustment for dynamic set representation, is designed. Then, a Scalable IIQF is further proposed to ensure the consistency when resizing. The theoretical performance of the proposed SIIQF is also analyzed. To demonstrate the feasibility of SIIQF, an instance of employing SIIQF for packet classification with the TSS algorithm is also given. Finally, the extensive evaluations demonstrate the effectiveness of the proposed SIIQF.

REFERENCES

[1] G. Yu, R. Liu, Q. Chen, and Z. Tang, "A hierarchical SDN architecture for ultra-dense millimeter-wave cellular networks," *IEEE Commun. Mag.*, vol. 56, no. 6, pp. 79–85, Jun. 2018.

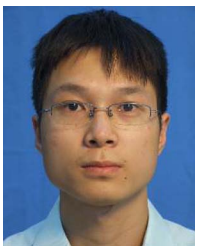
[2] O. Salman, I. Elhajj, A. Chehab, and A. Kayssi, "IoT survey: An SDN and fog computing perspective," *Comput. Netw.*, vol. 143, pp. 221–246, 2018.

[3] S. Rawas, "Energy, network, and application-aware virtual machine placement model in SDN-enabled large scale cloud data centers," *Multimedia Tools Appl.*, vol. 80, no. 10, pp. 15541–15562, 2021.

[4] N. McKeown et al., "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[5] C. Li, T. Li, J. Li, Z. Shi, and B. Wang, "Enabling packet classification with low update latency for SDN switch on FPGA," *Sustainability*, vol. 12, no. 8, 2020, Art. no. 3068.
[6] M. Yang, D. Gao, C. H. Foh, Y. Qin, and V. C. M. Leung, "A learned bloom filter-assisted scheme for packet classification in software-defined networking," *IEEE Trans. Netw. Serv. Manag.*, early access, 2022, doi: 10.1109/TNSM.2022.3181063.
[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, Jul. 1970.
[8] M. A. Bender et al., "Don't thrash: How to cache your hash on flash," in *Proc. VLDB Endow.*, Jul. 2012, vol. 5, pp. 1627–1637.
[9] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Experiments Technol.*, 2014, pp. 75–88.
[10] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 371–384.
[11] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 775–787.
[12] M. Al-Hisnawi and M. Ahmadi, "Deep packet inspection using quotient filter," *IEEE Commun. Lett.*, vol. 20, no. 11, pp. 2217–2220, Nov. 2016.
[13] S. Goudarzi et al., "A privacy-preserving authentication scheme based on elliptic curve cryptography and using quotient filter in fog-enabled VANET," *Ad Hoc Netw.*, vol. 128, 2022, Art. no. 102782.
[14] M. Al-Hisnawi and M. Ahmadi, "QCF for deep packet inspection," *IET Netw.*, vol. 7, no. 5, pp. 346–352, 2018.
[15] R. Williger and T. Maier, "Concurrent dynamic quotient filters: Packing fingerprints into atomics," Dissertation, Dept. Inform., Karlsruher Inst. fúr Technol., Karlsruhe, Germany, 2019.
[16] S. Dutta, A. Narang, and S. K. Bera, "Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams," *Proc. VLDB Endowment*, vol. 6, pp. 589–600, Jun. 2013.
[17] D. Guo, Y. He, and P. Yang, "Receiver-oriented design of bloom filters for data-centric routing," *Comput. Netw.*, vol. 54, pp. 165–174, Jan. 2010.
[18] D. Li, H. Cui, Y. Hu, Y. Xia, and X. Wang, "Scalable data center multicast using multi-class bloom filter," in *Proc. 19th IEEE Int. Conf. Netw. Protoc.*, 2011, pp. 266–275.
[19] F. Angius, M. Gerla, and G. Pau, "BLOOGO: BLOOm filter based GOssip algorithm for wireless NDN," in *Proc. 1st ACM Workshop Emerg. Name-Oriented Mobile Netw. Des. Architecture Algorithms Appl.*, 2012, pp. 25–30.
[20] T. M. Graf and D. Lemire, "Xor filters: Faster and smaller than bloom and cuckoo filters," *ACM J. Exp. Algorithmics*, vol. 25, Mar. 2020, Art. no. 1.5.
[21] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos, "Rosetta: A robust space-time optimized range filter for key-value stores," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2071–2086.
[22] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.
[23] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," *Inf. Process. Lett.*, vol. 101, no. 6, pp. 255–261, 2007.
[24] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 120–133, Jan. 2010.
[25] A. A. Abdulhassan and M. Ahmadi, "Cuckoo filter-based many-field packet classification using X-tree," *J. Supercomputing*, vol. 75, no. 9, pp. 5667–5687, 2019.
[26] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," *ACM J. Exp. Algorithmics*, vol. 25, Mar. 2020, Art. no. 1.1.
[27] H. Chen, L. Liao, H. Jin, and J. Wu, "The dynamic cuckoo filter," in *Proc. IEEE 25th Int. Conf. Netw. Protoc.*, 2017, pp. 1–10.
[28] L. Luo, D. Guo, O. Rottenstreich, R. T. Ma, X. Luo, and B. Ren, "The consistent cuckoo filter," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 712–720.
[29] P. Reviriego, J. Martínez, D. Larrabeiti, and S. Pontarelli, "Cuckoo filters and bloom filters: Comparison and application to packet classification," *IEEE Trans. Netw. Serv. Manag.*, vol. 17, no. 4, pp. 2690–2701, Dec. 2020.
[30] M. Wang and M. Zhou, "Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters," *Proc. VLDB Endowment*, vol. 13, pp. 197–210, 2019.
[31] A. D. Breslow and N. S. Jayasena, "Morton filters: Fast, compressed sparse cuckoo filters," *VLDB J.*, vol. 29, no. 2, pp. 731–754, 2020.
[32] P. Fu, L. Luo, S. Li, D. Guo, G. Cheng, and Y. Zhou, "The vertical cuckoo filters: A family of insertion-friendly sketches for online applications," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst.*, 2021, pp. 57–67.
[33] H. J. Chao, "Next generation routers," *Proc. IEEE*, vol. 90, no. 9, pp. 1518–1558, Sep. 2002.
[34] H. Lim and S. Y. Kim, "Tuple pruning using bloom filters for packet classification," *IEEE Micro*, vol. 30, no. 3, pp. 48–59, May/Jun. 2010.
[35] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "MAWILab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking," in *Proc. 6th Int. Conf.*, 2010, Art. no. 8.

MINGHAO XIE received the B.S. degree from the School of Computer Science and Technology, Guangdong University of Technology, Guangzhou, China, where he is currently working toward the master's degree. His research interests include software-defined network and edge computing.



QUAN CHEN (Member, IEEE) received the B.S., master's, and Ph.D. degrees form the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China. He is currently an Associate Professor with the School of Computers, Guangdong University of Technology, Guangzhou, China. Previously, he was with Georgia State University, Atlanta, GA, USA, as a Postdoctoral Research Fellow. His research interests include IoT networks, wireless communication and edge AI. He was the recipient of the Hong Kong Scholar Award, ACM SIGCOM CHINA Excellent Doctoral Dissertation and the CCF Excellent Doctoral Dissertation Nomination Award. He was also the Technical Program Committe member for several referred conferences, and the Technical Reviewer for several IEEE transactions.

TAO WANG received the Ph.D. degree from the School of Information Science and Technology, Sun-Yat-Sen University, Guangzhou, China, in 2010. He is currently an Associate Professor with the School of Automation, Guangdong University of Technology, Guangzhou, China. His research interests include smart manufacturing, software-defined network, time-sensitive network, industrial intelligence, and high-performance computing.

FENG WANG received the Ph.D. degree from Fudan University, Shanghai, China, in 2016. In 2017, he was a Postdoctoral Research Fellow with the Singapore University of Technology and Design, Singapore. He is currently a Hong Kong Scholar Fellow with the Hong Kong University of Science and Technology, Hong Kong. He is also an Associate Professor with the Guangdong University of Technology, Guangzhou, China. His research interests include signal processing for wireless communications, mobile-edge computing and intelligence, and applications of optimization algorithms. Dr. Wang was the recipient of the Examplary Reviewer for IEEE WIRELESS COMMUNICATIONS LETTERS in 2020. He was a member of the Technical Program Committees for several IEEE conferences and a reviewer for several IEEE journals.

YONGCHAO TAO received the B.S. and master's degrees from the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China. He is currently an Senior Engineer with the Shenzhen Academy of Aerospace Technology, Shenzhen, China. His research interests include wearable and ubiquitous computing, and IoT networks.

LIANGLUN CHENG received the M.S. degree from the Huazhong University of Technology, Wuhan, China, and the Ph.D. degree in control science and engineering from the Chinese Academy of Sciences, Beijing, China. He is currently a Professor with the School of Computers, Guangdong University of Technology, Guangzhou, China. He is also the Executive Director of Chinese Robot Automation Association. His research interests include wireless sensor network and Internet of Things, cyber-physical system, industrial Big Data, modeling, and optimal control of complex systems.