Journal of Network and Systems Management (2022) 30:48
https://doi.org/10.1007/s10922-022-09661-9

2DNCF-PIT: Two-Dimensional Neighbor-Based Cuckoo Filter for Pending Interest Table Lookup in Named Data Networking

Arman Mahmoudi[1] · **Mahmood Ahmadi**[1]

Received: 11 June 2021 / Revised: 7 February 2022 / Accepted: 27 March 2022 /
Published online: 23 April 2022

Abstract

Named data networking is one of the proposed architectures for the future Internet. In this architecture, names play an essential role. Packets in named data networking have names that are used instead of IP addresses, and based on these names, packets are forwarded through the network routers. For this purpose, named data network routers have three data structures content store, pending interest table (PIT), and forwarding information base. In named data networking, the PIT table plays an important role. In this table, the information of all Interest packets waiting for Data packets is stored. The PIT should be able to search, delete, update information quickly, and in turn, take up little memory space. In this paper, a new variant of the Cuckoo filter to improve the performance of the PIT table called two-dimensional neighbor-based Cuckoo filter (2DNCF) is proposed. The proposed 2DNCF uses the physical neighbor of the selected bucket by the Cuckoo filter that increases the utilization of the neighbor buckets, as well as the performance of the proposed 2DNCF filter. In this data structure, which is essentially a two-dimensional Cuckoo filter, an attempt has been made to use the second hashing function less than the first one in the Cuckoo filter. Due to less use of the second hashing function in this filter, it is more efficient in inserting, deleting, and searching than the standard Cuckoo filter. The simulation results show that this filter has a lower false positive rate than the standard Cuckoo filter. Accordingly, it improves the insertion, deletion and lookup performance of the PIT table compared to the other solutions.

Keywords PIT · Pending interest table · Cuckoo filter · Two-dimensional neighbor-based Cuckoo filter · 2DNCF

✉ Mahmood Ahmadi
m.ahmadi@razi.ac.ir

Extended author information available on the last page of the article

1 Introduction

The new idea proposed for the future Internet is based on changing the current Internet host-to-host model to the content-to-content model [1]. This idea is known as the Information-Centric-Networking (ICN) [2]. ICN employs a model similar to a publish-subscribe model. In this idea, an attempt has been made to change the architecture of the Internet from address-oriented to content-oriented [3, 4]. To implement ICN, new architectures have been proposed, one of which is Named Data Networking (NDN) [3], in which information is sent, received, and searched based on names. In NDNs, instead of asking "where is the information," it asks "what information is needed" [5]. Names used in NDN networks are hierarchical, such as URLs, separated by "/" [1, 6, 7]. So names in NDN networks have variable lengths. This makes name lookup in these types of networks more complex than IP-based networks [1, 7], which has a significant impact on network performance. NDN routers contain three data structures including, Content Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB). CS is for storing, and caching Data packets, which have passed through the router, this data structure makes the network more efficient. FIB sends Interest packets. PIT is the location of Interest packets whose Data packet has not yet arrived at the current router. Since after arriving of each Interest packet or Data packet, a delete, insert or update operation in the PIT table is done, so this table must be able to perform this operation quickly [5]. In this paper, a new variant of the Cuckoo filter called two-dimensional neighbor-based Cuckoo filter to improve the performance of the PIT table is presented. The Cuckoo filter is an array of buckets that uses two hashing functions to insert items. If the location of the first candidate is empty, the item will be inserted into the filter. Otherwise, the item will be inserted in the location of the second candidate, which is determined by the second hashing function. If both candidate locations are full, the Cuckoo filter inserts an item into the filter using a special relocation algorithm [8]. Two important features of the Cuckoo filter in comparison to other filters, e.g., Bloom filter, Quotient filter are: low false positive rate, and faster lookup time.

The two-dimensional neighbor-based Cuckoo filter (2DNCF) is a two-dimensional Cuckoo filter that tries to use the physical neighbor of the selected bucket. To select the physical neighbor of each bucket, the last three bits (three least significant bits) of the fingerprint is selected which shows one out of eight neighbors of each bucket. The utilization of the neighbor bucket logically increases the bucket length and finally the 2DNCF performance. The use of a neighbor bucket also reduces the use of the second hashing function, which makes lookup faster, shorter insertion time, and shorter deletion time than standard Cuckoo and Bloom filters. A unique feature of this filter is that the potential false positive is significantly reduced compared to the standard Cuckoo filter. Due to the features of this filter, and the requirements of the PIT table, this filter can be used to implement the PIT table. The use of 2DNCF in PIT table design reduces the lookup time, inserting and deleting interest packets, which are important factors in NDN networks. The simulation results using the NDNSim simulator show that

the proposed 2DNCF filter improves 12%, 10%, and 40% lookup time, deletion time, and insertion time, in compared to the standard Cuckoo filter. In addition, 2DNCF-PIT outperforms other PIT approaches in terms of lookup time, insertion time, deletion time, and memory consumption.

The novelty of this paper is explained as follows:

- Proposal of two-dimensional neighbor-based Cuckoo filter (2DNCF) that works based on the standard Cuckoo filter.
- Proposal of a PIT table design based on the 2DNCF (2DNCF-PIT).
- Evaluation of the 2DNCF and 2DNCF-PIT based on the proposed variant of the Cuckoo filter.

The rest of this paper is organized as follows. In Sect. 2, related works are presented. In Sect. 3, the brief concept of name data networking and Cuckoo filter is described. In Sect. 4, the proposed 2DNCF, and 2DNCF-PIT are presented. The evaluation of the proposed method is discussed in Sect. 5. Section 6 concludes the paper.

2 Related Works

In this section, the related researches in PIT architecture are reviewed. In [9], an algorithm of combining Bloom filter and binary trie for IP lookup on IP-based networks is presented. The authors have tried to reduce false positives with the help of off-chip memory. In this algorithm, all nodes in the binary trie are programmed by a Bloom filter in on-chip memory and stored in a hash table in off-chip memory.

In [10], the authors explain a solution called the Longest aware Cuckoo filter (LACF) to reduce the false positive rate of Cuckoo filter when searching IP addresses in IP-based networks. In this solution, more hashing functions are suggested in some cases. In NDN networks, packet search plays a central role in the three tables CS, PIT, and FIB. In the PIT table, after receiving each Interest packet or Data packet, a search is performed on the whole network, which shows the importance of the search in the PIT table.

In [6], dai et al. used a method called Name Component Encoding (NCE). In this method, each name component of the Interest or Data packet is encoded by an NCE method, and then the generated code by the Encoding Name Prefix Trie (ENPT) method is converted to a Name Prefix Trie (NPT). This method has additional calculations, and the PIT table must also have a special architecture.

Another PIT design technique has been proposed in [11] called MaPIT. MaPIT consists of two components: Index Table, and Packet Store. Index Table also has two data structures, a standard Bloom filter, and a Mapping Array. Index Table is in on-chip memory, and Packet Store is in off-chip memory. Both the standard Bloom filter data structure and the Mapping Array are arrays of bits. The mapping array has n bits, and the standard Bloom filter is divided into n parts. To add an item to this data structure, the item using the k hashing function is first inserted into the Bloom filter. The number of bits that have been changed to bit one is entered in the Mapping Array. The number in the Mapping Array is where the item is stored in the

Packet Store. This data structure supports deletion, but has additional calculations when searching and inserting items, and also has a high false positive rate.

Another method used to reduce the lookup time of the three PIT, CS and FIB tables in NDN networks is to use the hash table presented in [12]. In this method, instead of tables in NDN networks, Chain Hash Table data structure is used, the size of each entry in the hash table is 64 bytes. The limitations of this method are high memory consumption and long search time.

In [2], a new method is used to implement the PIT table called FTDF-PIT. FTDF is a filter based on Bloom and Quotient filters. This method is a two-dimensional array of bits that uses a hashing function for items. It first obtains the row and column values using this hashing function. It changes its value to bit 1. Low memory consumption and low search time are some of the advantages of this method, but there is some false positive, and it does not support deletion.

In [13], PIT control management (PITCM) to mitigate PIT overflow and enhance the PIT utilization is proposed. The PITCM utilizes a thresholding mechanism in interest lifetime intelligently, a policy to determine the highest lifetime interest, and an adaptive virtualized PIT. The prediction of PIT conditions, e.g., early PIT overflow prediction and reaction, is the responsibility of the adaptive virtual PIT block. The smart thresholding of interest lifetime adjusts the lifetime value of received interest packets. Management of PIT entries in an efficient way is controlled by the highest lifetime latest request policy. The flow PIT sizing problem of the forwarding systems in the NDN networks is addressed by this work.

In [14], to increase PIT efficiency by reducing the residence time of non-responded PIT entries, round tripe time-aware pending interest PIT (RAPIT) is proposed. The round tripe time-aware pending interest table (RAPIT) presents an approach to increase the performance of PIT by decreasing the residence time of non-responded PIT entries. The authors used a technique to determine the residence time of PIT entries based on estimating the return time of data packets. To measure the return time of the data packet, they mark the initial request and the corresponding data packet, and then store the measured time in the forwarding information base (FIB) of the data packet for future use. In addition, they use a PIT replacement strategy when the PIT is completely filled.

In [15], the DiPIT approach to decrease the memory usage of the PIT table using the Bloom filter is proposed. In this data structure, for each router interface, a counting Bloom filter is installed, which each of the Interest packets will be inserted in the filters using the k hashing functions. This method speeds up the search, but one of the limitations of this method is the use of a counting Bloom filter to support deletion, a high false positive rate, and when the router receives a data packet, it must check all the Bloom filters to be able to send the desired interface.

In [16], the PIT table of a router is modeled using queuing theory to achieve the optimum size. They evaluate the optimal PIT size to trade-off the network performance and the cost of the PIT table. This model defines an objective function to minimize the PIT size subject to the interest drop probability as the upper bound.

In [17], an analytical model of two PIT replacement approaches is proposed. These approaches are called PIT replacement without reservation (PRWR) and PIT replacement with reservation (PRWR). The first approach utilizes the PIT

replacement, while the second approach uses the PIT entry reservation in conjunction with the first one. Both the approaches utilize Continuous Time Markov Chain (CTMC) and improve efficiency in comparison to the basic NDN. Table 1 depicts the comparison of different techniques used in IP and PIT table lookup.

This paper presents a two-dimensional filter that acts like a Cuckoo filter. In this filter, an attempt has been made to use the neighbor buckets for each selected bucket by the hashing functions of the Cuckoo filter. The use of these neighbor buckets causes that the second hashing function in 2DNCF to be used less than the second one in the Cuckoo filter, thus reducing the search, insertion, and deletion time. The false positive rate is also reduced in this proposed filter.

3 Named Data Networking and Cuckoo Filter

In this section, the brief concept of named data networking and Cuckoo filter is presented.

3.1 Named Data Networking

NDN network is a new paradigm that influences the future of computer networks and the Internet. NDN is presented in the direction of the information-centric networks (ICN) [18]. The packets of this network have a unique name, and routing is done based on them [18]. NDN networks organize component-based names in a hierarchical manner separated by "/" character. These names are like URLs that can be used to identify packets [6]. For example, "razi/ac/ir" has three components, which are "ir", "ac", and "razi". This is an example of a name on NDN networks. The hierarchy of names in this architecture causes a lot of memory to be consumed [19]. There are two types of packets in these networks: the Interest packet and the Data packet, both of which carry names and are identified by these names [6]. Figure 1 shows the structure of these packets.

When a client requests data, it creates an Interest packet, puts the name of the data in it, and sends it over the network. When the data producer receives an Interest packet, it creates a Data packet including names of the data and data. Puts the desired data packet and sends it through the network. Based on these names that network routing is done. Routers in NDN networks have three data structures: CS, Pending Interest Table (PIT), and Forwarding Information Base (FIB) [6]. CS is the location of data packets that have been forwarded by the current router. CS is also used to cache, temporarily, arriving Data packets depending on different caching mechanisms. When an Interest packet arrives at the router, this table is searched first. If there is a data packet corresponding to the Interest packet in this table, the Data packet will be sent back to the client. This table in NDN networks allows the network resources to be used optimally. FIB table is used for forwarding and routing the packets. FIB is the location of the information of interest packets that the router has not yet received. In this table, information such as the name of the interest packet, the input interface of the packet, the arrival time to set the timer in the

Table 1 Comparison of different techniques used in IP and PIT table lookup

Paper	Technique	Application	False positive rate	Memory consumption	Insertion time
[9]	Bloom filter and trie algorithm	IP lookup	N.A	High (off-chip and on-chip)	N.A
[10]	Length-aware Cuckoo filter	IP lookup	Lower than CF	Higher than CF	Higher than CF
[15]	Counting Bloom filter (DiPIT)	PIT lookup	Higher than CF	Higher than CF	Lower than BF
[11]	Mapping Bloom filter (MaPIT)	PIT lookup	High	High (Offchip and onchip)	High
[2]	Bloom and quotient filters	PIT lookup	Lower than Bloom filter	Low	Lower than BF

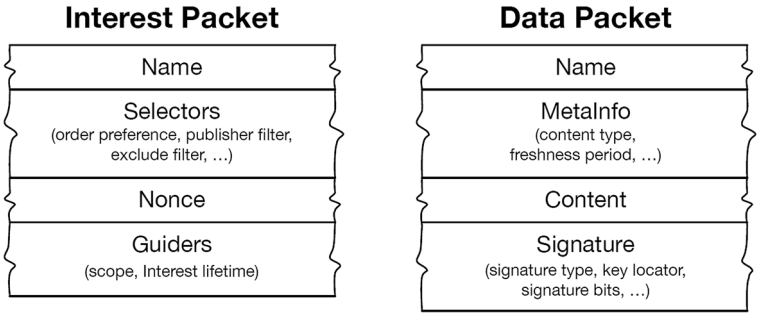


Fig. 1 Structure of the NDN network packet [1]

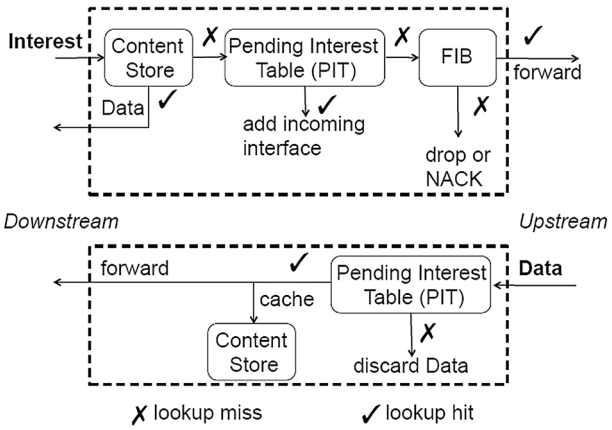


Fig. 2 NDN router strategy when interest and data packets [6]

network is stored. Figure 2 shows the router's reaction when it receives an Interest packet or Data packet [1, 6].

When the Data packet is received, the Data packet is stored in the CS table, and then the PIT table is searched. If the same name is found, the Data packet is sent to all the interfaces, and then the found entry is deleted.

3.2 Cuckoo Filter

A Cuckoo filter is a membership query data structure based on the Cuckoo hash table [8]. The Cuckoo filter is a bucket-based hash table that stores a small fingerprint of the item in the filter instead of storing it completely, resulting in less memory space [20]. The Cuckoo filter supports dynamic deletion and insertion and is also more efficient than the Bloom, and Quotient filters [8]. Unlike different types of Bloom filters that require more memory space to delete, the Cuckoo filter supports deletion without the need for more memory [10]. This filter uses a method called partial-key. Thus, if the fingerprint of item x is called $fingerprint(x)$, the first hashing

function is called $h_1(x)$ and the second hashing function is $h_2(x)$, Eq. 1 is used to obtain $h_2(x)$.

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \oplus hash[fingerprint(x)] \end{aligned} \tag{1}$$

In this regard, the second hashing function is obtained based on the first hashing function and fingerprint. This method helps the Cuckoo filter, items whose first and second candidate locations are full, to be inserted in the appropriate location by relocation.

To add an item, first, fingerprint $h_1(x)$ and $h_2(x)$ are calculated for item x. If bucket $h_1(x)$ or $h_2(x)$ have empty cells, the item is added; otherwise, one of the $h_1(x)$ or $h_2(x)$ buckets is selected randomly [8]. To delete an item, fingerprint $h_1(x)$ and $h_2(x)$, and *fingerprint*(x) are calculated. If fingerprints are found in the bucket with address $h_1(x)$, and $h_2(x)$, it is removed, otherwise, it does nothing.

To search for item x, *fingerprint*(x), $h_1(x)$ and $h_2(x)$ are calculated. If the *fingerprint*(x) is found in buckets with address $h_1(x)$, and $h_2(x)$, it returns True otherwise returns False. Each bucket has b entries, and instead of storing the item in the Cuckoo filter, f bit fingerprint is used. In the Cuckoo filter, like the Bloom filter, a false positive occurs. The false positive in the Cuckoo filter is calculated using Eq. 2.

$$fp_{\text{Cuckoo filter}} \approx \frac{2b}{2^f} \tag{2}$$

In Eq. 2, the b is bucket size (number of entries in a bucket), and f is the fingerprint length in terms of bits.

4 Two-Dimensional Neighbor-Based Pending Interest Table

In this section, the proposed two-dimensional neighbor-based Cuckoo filter (2DNCF) data structure and the two-dimensional neighbor-based Cuckoo filter Pending interest table (2DNCF-PIT) are presented.

4.1 Two-Dimensional Neighbor-Based Cuckoo Filter

In this section, the new 2DNCF filter is introduced. First, the architecture of this filter is explained, and then adding, deleting, and searching operations are presented.

Similar to the Cuckoo filter, 2DNCF is an array of buckets, except that 2DNCF uses a two-dimensional array. Each bucket is identified using the row and column addresses. The technique utilized in [21] is used to calculate the rows and column address of each bucket. This technique converts the value of the hashing function into two parts. The first part shows the row, and the second part shows the column numbers. This technique returns the q most significant bit as the first value and the remaining bits as the second value [1, 21]. The 2DNCF filter, like the Cuckoo filter, uses two hashing functions and a partial-key method. Equation 3 shows how to calculate hashing functions and fingerprints.

$$h_{1-2DNCF}(x) = hash_{2DNCF}(x)$$
$$h_{2-2DNCF}(x) = h_{1-2DNCF}(x) \oplus hash_{2DNCF}[fingerprint(x)] \quad (3)$$

In this filter, two candidate buckets are used. Each bucket has a list of neighbors. Figure 3 shows the neighbors and neighbor bucket numbering method for each bucket.

The last three bits of the fingerprint are used to select the neighbor bucket. For example, if the last three bits of the fingerprint show the number three, the third bucket of the neighbors is selected to insert the item. In this case, there are two situations:

- Bucket has eight neighbors: In this case, the first candidate bucket is checked. If the bucket has an empty entry, then the item is inserted into the filter. Otherwise, the filter first checks the neighbors of the first candidate bucket if they have an empty entry, then, the item is added to the filter. If the neighbors bucket does not have an empty entry, the filter uses the second hashing function to find the second candidate bucket, and checks it. If it has an empty entry, the item is added to the filter. If it does not have an empty entry, then its neighbors are checked, if they have an empty entry, the item is inserted. If both the first and second candidate buckets and their neighbors do not have an empty entry, then according to the Cuckoo filter method, one of the candidate buckets is selected randomly first. In the selected bucket, one of the entries is selected randomly, and a new item replaces it, then the filter inserts the item that was ejected from the bucket in the filter according to the same method.
- The bucket has 3 or 5 neighboring buckets: In cases where the first or second candidate bucket is in row 0, column 0, row $m-1$, and column $m-1$, then it has 3 or 5 neighbor buckets. In this case, if the last three fingerprint bits show a number that is not in the neighbor bucket list, the filter, like the standard Cuckoo filter, ignores the neighbor list and continues to work according to the standard Cuckoo filter.

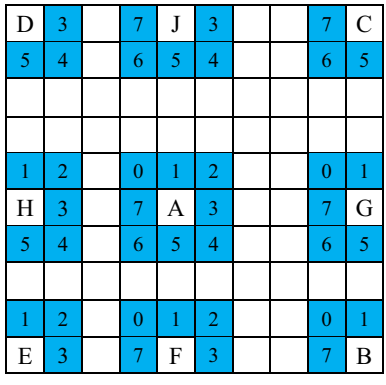
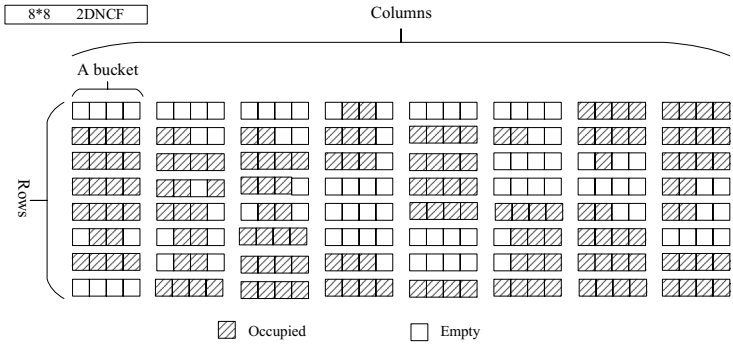


Fig. 3 Neighbors, and neighbor buckets numbering method in 2DNCF. The characters show the items and numbers show the neighbor buckets

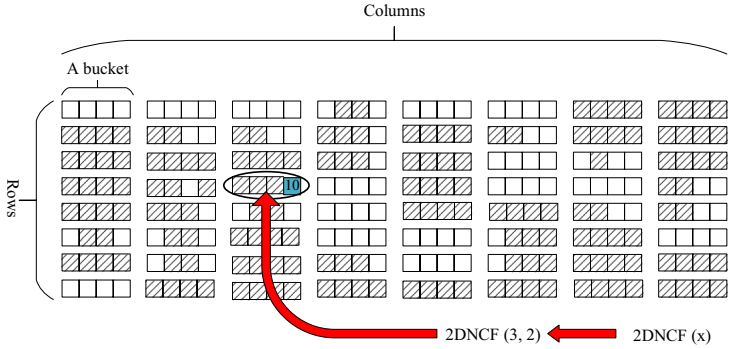
Figure 4(a) shows a 2DNCF filter with a size of 8×8. Each bucket also has four entries and a 4-bit fingerprint is provided for each item. Several items have been added to the filter. Now do this to insert the item x. The insertion of item x is as follows. If $fingerprint(x) = 10$, and $h_1(x) = 26$, then value of $h_1(x) = 26 = (11010)_2$. The row and column of the desired bucket is equal to $row = (11)_2 = 3$ and $col = (010)_2 = 2$. Bucket $C(3, 2)$ has an empty entry, so the fingerprint of item x is simply inserted in bucket $C(3, 2)$. Figure 4(b) shows this condition.

Now the item y is added to the filter. If $fingerprint(y) = 6$, and $h_1(y) = 26$, then $h_1(y) = 26 = (11010)_2$ and $row = (11)_2 = 3$, and $col = (010)_2 = 2$. Because $C(3, 2)$ does not have an empty entry, neighbors of bucket $C(3, 2)$ are checked. The value of fingerprint for item y is equal to 6, which according to the last three bits of fingerprint shows the number 6, so bucket number 6 is checked by neighbors. This bucket has an empty entry, so the fingerprint of item y, in neighbor number 6 of bucket $C(3, 2)$ is inserted. Now add the item z to the filter. If $fingerprint(z) = 15$, and $h_1(z) = 18$, then $h_1(z) = 18 = (10010)_2$, and $row = (10)_2$, and $col = (010)_2$. Bucket $C(2, 2)$ is checked, which has no empty entry, so bucket's neighbors are checked. According to the last three bits of the fingerprint, neighbor number 7 is selected. This bucket also does not have an empty entry, so the second candidate bucket which is selected by the second hashing function is used. Therefore, $h_{2-2DNCF}(z) = 21 = (10101)_2$, and $row = (10)_2 = 2$, and $col = (101)_2 = 5$. Bucket $C(2, 5)$ has an empty entry, so the fingerprint of item z is inserted in $C(2, 5)$. The steps for inserting the item $z$are shown in Fig. 4d. Now consider the item w. If $fingerprint(w) = 7$, and $h_1(w) = 18$, then the row and column of the desired bucket are $C(2, 2)$. Bucket $C(2, 2)$ is full, according to the amount of fingerprint, neighbor 7 is checked. This bucket is also full, therefore, the second candidate bucket is investigated. According to the amount $h_{2-2DNCF}(w) = 26 = (11010)_2$, $row = 3$, and $col = 2$ and bucket $C(3, 2)$ is selected which is full. So, according to the fingerprint, neighbor number 4 is checked. Neighbor number 4 of the bucket also has no empty entry. In this case, one of the candidate buckets is selected randomly. In this example, the candidate buckets for item w are $C(2, 2)$ and $C(3, 2)$. Bucket $C(3, 2)$ is selected. One of its entries is randomly ejected from the bucket, here is entry number 2, which contains item a. The fingerprint of item w is stored in entry number 2, and the above method must be performed to insert item a. Figure 4d shows the steps for adding the item w. Now suppose item v is inserted in the filter. If $fingerprint(v) = 14$, and $h_{2-2DNCF}(z) = 24$, then $row = 6$, and $col = 3$. This situation is the second state of the proposed filter. Bucket $C(6, 0)$ is full, so its neighbors are checked. Bucket $C(6, 0)$ neighborhood list has neighbors 1, 2, 3, 4, and 5. Figure 4e depicts this situation. According to the value of the fingerprint, the neighbor number 6 should be checked, considering that neighbor 6 is not in the neighborhood list of bucket $C(6, 0)$, so the algorithm checks the second candidate bucket and uses the same method above to add the item. In new situation, $h_{2-2DNCF}(z) = 22$ and $row2 = 2$, $col2 = 6$, therefore, $C(2, 3)$ is selected. This bucket has an empty entry, and the item is inserted in it.

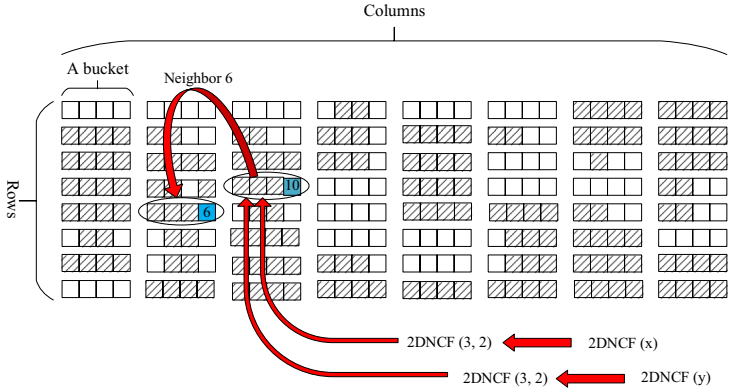
Algorithm 1, shows how to insert an item into a 2DNCF filter. This algorithm can execute the relocation function k times.



(a) An 8×8 2DNCF filter with bucket size 4.

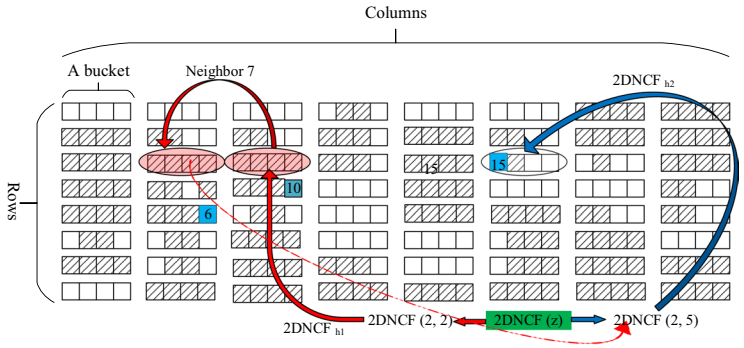


(b) $fingerprint(x) = 10$, $h_1(x) = 26 = (11010)_2$ $row = (11)_2 = 3$ and $col = (010)_2 = 2$, $fingerprint(x)$ is inserted in $2DNCF(3, 2)$

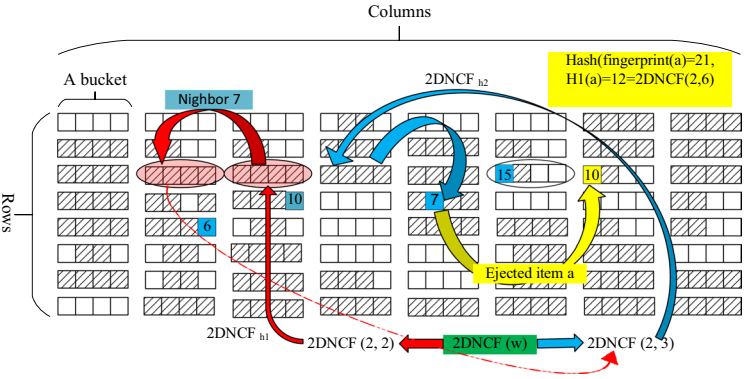


(c) $fingerprint(y) = 6$, $h_1(y) = 26 = (11010)_2$, $row = (11)_2 = 3$, $col = (010)_2 = 2$,$2DNCF(3, 2) = full$, $neighbor(6)$ is empty, $fingerprint(y)$ is inserted in $neighbor(6)$

Fig. 4 2DNCF filter. **a** Sample 2DNCF filter. **b** Insertion of item x. **c** Insertion item y. **d** Insertion of item z. **e** Insertion of item w with relocation



(d) $fingerprint(z) = 15$, $h_1(x) = 18 = (10010)_2$, $row = (10)_2 = 2$, $col = (010)_2 = 2$, $2DNCF(2,2) = full$, $neighbor(2DNCF(2,2)) = bucket(7) = full$, $h_{2-2DNCF}(z) = 21 = (10101)_2$, $row = (10)_2$, $col = (101)_2 = 5$, $2DNCF(2,5) = 15$, $fingerprint(z)$ is inserted in $2DNCF(2,5)$.



(e) $fingerprint(w) = 7$, $h_1(w) = 18 = (10010)_2$, $row = (10)_2 = 2$, $col = (010)_2 = 2$, $2DNCF(2,2) = full$, $neighbor(2DNCF(2,2)) = bucket(7) = full$, $h_{2-2DNCF}(w) = 26 = (11010)_2$ $row = (11)_2$, $col = (010)_2$, $2DNCF(3,2) = full$, $neighbor(2DNCF(3,2)) = bucket(4) = full$ $C(3,2) =$ select a random bucket and eject a.

Fig. 4 (continued)

Algorithm 1 The pseudo code of Insert algorithm

```
2DNCF_insert(item) { item is string} { C is 2DNCF}
f=h_fingerprint(item)
h_{1-2DNCF}=hash_function(item)
(row1,col1)=getrowcol(h_{1-2DNCF})
if C(row1,col1) has an empty entry then
    C(row1,col1).insert(f)
    return Done
else
    select_neighbor=getfingerprintselector(f) {get 3 bits for selected neighbor}
    if select_neighbor in list_neighbor(C(row1,col1)) then
        (rowneighbor, colneighbor)=getrowcolneighbor(row1,col1,f)
        if C(rowneigbor, colneighbor) has an empty entry then
            C(rowneighbor,colneighbor).insert(f)
            return Done
        end if
    end if
end if
h_{2-2DNCF} = h_{1-2DNCF} XOR hash_function(f)
(row2,col2)=getrowcol(h_{2-2DNCF}) {retrieve row and col of h_{2-2DNCF}}
if C(row2,col2) has an empty entry then
    C(row2,col2).insert(f)
    return Done
else
    select_neighbor=getfingerprintselector(f)
    if select_neighbor in list_neighbor(C(row2, col2)) then
        (rowneighbor, colneighbor)=getrowcolneighbor(row2, col2, f)
        if C(rowneigbor, colneighbor) has an empty entry then
            C(rowneighbor, colneighbor).insert(f)
            return Done
        end if
    end if
end if{h_{2DNCF} select randomly h_{1-2DNCF} and h_{2-2DNCF}} {k is maximum number of
hits}
for i=0 ;i ≤ k; k++ do
    row,col=getrowcol(h_{2DNCF}) {retrieve row and col of h_{2DNCF}}
    Select randomly e entry of C(row,col)
    Swap f and fingerprint stored in e
    h_{2DNCF} = h_{2DNCF} XOR hash_function(f)
    if C(row,col) has an empty entry then
        C(row,col).insert(f)
        return Done
    else
        select_neighbor=getfingerprintselector(f) {get 3 bits for selected neighbor}
        if select_neighbor in list_neighbor(C(row2,col2)) then
            (rowneighbor, colneighbor)=getrowcolneighbor(row2, col2, f)
            if C(rowneighbor, colneighbor) has an empty entry then
                C(rowneighbor, colneighbor).insert(f)
                return Done
            end if
        end if
    end if
end for
return failure
```

To lookup for an item in 2DNCF, the first hashing function for the item fingerprint is calculated. If a fingerprint is found in the desired bucket, the search will be successful.

Algorithm 2 depicts the pseudo code of the lookup operation.

Algorithm 2 The pseudo code of the lookup algorithm.

```
1: 2DNCF-lookup(item) { item is string} { C is 2DNCF}
2: f=h_fingerprint(item)
3: h_{1-2DNCF}=hash_function(item)
4: (row1,col1)=getrowcol(h_{1-2DNCF}) {retrieve row and col of h_{1-2DNCF}}
5: if C(row1, col1) has f then
6:     return C(row1,col1)
7: else
8:     select_neighbor=getfingerprintselector(f) {get 3 bits for selected neighbor}
9:     if select_neighbor in list_neighbor(C(row1,col1)) then
10:        (rowneighbor,colneighbor)=getrowcolneighbor(row1,col1,f)
11:        if C(rowneighbor,colneighbor) has f then
12:            return C(rowneighbor,colneighbor)
13:        end if
14:    end if
15: end if
16: h_{2-2DNCF} = h_{1-2DNCF} XOR hash_function(f)
17: (row2,col2)=getrowcol(h_{2-2DNCF}) {retrieve row and col of h_{2-2DNCF}}
18: if C(row2,col2) has f then
19:    return C(row2,col2)
20: else
21:    select_neighbor=getfingerprintselector(f) {get 3 bits for selected neighbor}
22:    if select_neighbor in list_neighbor(C(row2, col2)) then
23:        (rowneighbor,colneighbor)=getrowcolneighbor(row2, col2, f)
24:        if C(rowneigbor,colneighbor) has f then
25:            returnC(rowneighbor, colneighbor)
26:        end if
27:    end if
28: end if
29: return false
```

Algorithm 3 shows the removal from the 2DNCF filter. To remove an item, the filter first is searched for it. If the desired item is found, it removes it from the filter.

Algorithm 3 The pseudo code of delete algorithm.

```
2DNCF-Delete(item) { item is string} { C is 2DNCF}
f=h_fingerprint(item)
h_{1-2DNCF}=hash_function(item)
(row1, col1)=getrowcol(h_{1-2DNCF}) {retrieve row and col of h1-2DNCF}
if C(row1,col1) has f then
    remove f from bucket
    return true
else
    select_neighbor=getfingerprintselector(f) {get 3 bits for selected neighbor}
    if select_neighbor in list_neighbor(C(row1, col1)) then
        (rowneigbor, colneighbor)=getrowcolneighbor(row1 ,col1, f)
        if C(rowneigbor, colneighbor) has f then
            remove f from bucket
            return True
        end if
    end if
end if
h_{2-2DNCF} = h_{1-2DNCF} XOR hash_function(f)
(row2, col2)=getrowcol(h_{2-2DNCF}) {retrieve row and col of h_{2-2DNCF}}
if C(row2,col2) has f then
    remove f from bucket
    return True
else
    select_neighbor=getfingerprintselector(f) {get 3 bits for selected neighbor}
    if select_neighbor in list_neighbor(C(row2,col2)) then
        (rowneigbor,colneighbor)=getrowcolneighbor(row2,col2,f)
        if C(rowneigbor,colneighbor) has f then
            remove f from bucket
            return True
        end if
    end if
end if
```

4.2 Two-Dimensional Neighbor-Based Pending Interest Table

This section introduces a 2DNCF-based data structure for the PIT table in NDN networks. In the PIT table, each entry has four fields (content_name, list_interface, list_nonces, expiration) [11, 22]. The proposed PIT table focuses on reducing search time and memory usage. Fig. 5 shows a general view of a named data networking router with its tables.

In this figure, the architecture of the 2DNCF-PIT table is shown. As shown in this 2DNCF-PIT table, a two-dimensional neighbor-based Cuckoo filter is used. Figure 5 shows the fields of an entry in the 2DNCF-PIT table. In this implementation, instead of a content_name, the packet name fingerprint is used. When the router receives the interest packet, if the Data packet is found in CS, the packet is sent to the data requester, otherwise the item is first searched based on content_name according to Algorithm 2 in the 2DNCF-PIT. This means that, the content_name is given as input to the algorithm. If the search is successful, the input interface number is added to the list_interface field of the same entry. If the search is unsuccessful, and the

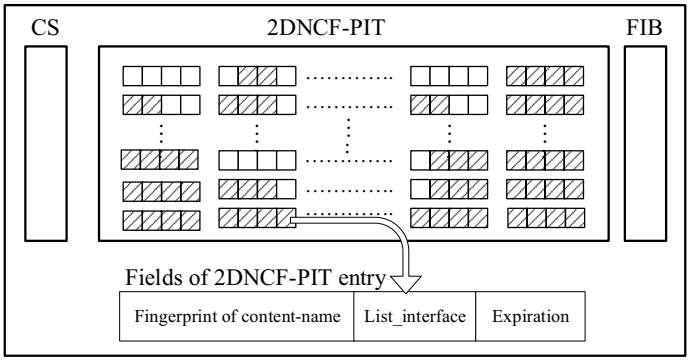


Fig. 5 NDN router architecture based on 2DNCF-PIT, and The fields of 2DNCF-PIT's entries

interest packet with the same name as the received interest packet is not found in the 2DNCF-PIT table, then according to Algorithm 1, the interest packet is added to the 2DNCF-PIT table, the input interface number is added to list_interface. Algorithm 4 depicts the pseudo code of interest packet routing.

Algorithm 4 The pseudo code of interest packet routing.

```
Input: interest packet as I
if search in CS(content_name(I)) then
    return Data Packet
else
    if 2DNCF_lookup(content_name(I) then
        update(list_interface)
    else
        2DNCF_insert(content_name(I))
        insert(list_interface )
    end if
end if
```

Algorithm 5 depicts receiving data packet by the router. When a router receives a data packet, it first stores it in the CS table, and searches for the received data packet in 2DNCF-PIT based on the content_name. If the search is successful, it sends the data packet to the entire list_interface, and according to Algorithm 2, it deletes that entry, and if the search fails, it ignores the data packet.

Algorithm 5 The pseudo code of receiving data packets by the router.

```
Input: interest packet as D
save D
if 2DNCF_lookup(content_name(D)) then
    forward to all list_interface
else
    drop D
end if
```

To store the list of interfaces, n bits are used for each entry. For example, a router with 16 interfaces is considered 16 bits per entry in 2DNCF-PIT. When the interest packet is entered from the interface of the number m, the bit of the number m in the desired entry changes to one. The 2DNCF-PIT uses a central timer. The 2DNCF-PIT uses a timer to delete all expired interest packets. Time To Live (TTL) equals $2r$, where r is the average response time [1]. In this condition, two states are occurred.

1. The router receives the corresponding Data packet before the end of the TTL time, in which case the Interest packet is removed from the 2DNCF-PIT table.
2. The router will receive the Data packet after the TTL expires. Because after the expiration of the TTL time, the Interest packet stored in 2DNCF-PIT will be deleted [1].

5 Simulation Results and Evaluation

In this section, first, the proposed 2DNCF filter is compared with the Cuckoo filter in terms of lookup time, insertion time, deletion time, and load factor. Subsequently, the proposed 2DNCF-PIT is compared with HT-PIT, FTDF-PIT, NCF, and DiPIT approaches in terms of lookup time, insertion time, deletion time, and memory consumption.

5.1 Simulation Parameters and Configurations

Table 2 shows the simulation parameters and system specifications used in this work.

To implement 2DNCF-PIT, an NDN network is considered with three routers, 8 ports, four nodes, and the incoming packet rate for each port is assumed between 10 Mpcs (Mega packets per second) and 100 Mpcs. It should be noted that the network topology is useful when the routing algorithms need to be evaluated. In this case which a table like PIT is evaluated a huge number of names is required. Therefore, we used a simple network topology with three routers. The size of 2DNCF-PIT contains the numbers of 8000 to 8,000,000 entries based on incoming bandwidth. Finally, the intended RTT is 80 ms in these experiments [3]. The used dataset for the experiment of the CNC dataset is the one used in [6]. The specifications of the system used in this work are: AsusX299 Delux h1

Table 2 Simulation parameters

Number of routers	3
Number of ports in each router	8
Packet rate of each port	10–100 Mpcs
Number of entries of DiCuPIT	8000–8,000,000
Round trip time (RTT)	80 Msec
Dataset	CNC dataset [10]
System specification	AsusX299 Delux h1
	Equipped with Core i-9 7900 CPU
	64 GB DDR4 3200 g-skill (4*16)RAM
	1080 11 GB DDR5 GPU
Programming language	Python
Bucket size	4 entries [18]
Fingerprint size	6 bits [18]
Expiration size	16 bits
Hash function of chain hashing	MurMur (32 bits)
Number of hashing functions in DiPIT	4
Number of bits in DiPIT counting Bloom filter	4 bits
Number of bits to show the interface numbers	8 bits
NDNSim version	2.6

equipped with Core i-9 7900 CPU, 64 GB DDR4 3200 g-skill (4*16)RAM, and 1080 11 GB DDR5 GPU. The Python programming language has been used in the implementation. The bucket and fingerprint sizes have been set to 4 entries and 6 bits [17], respectively.

Figure 6 depicts the router architecture with eight interfaces that utilized 2DNCF.

In Fig. 6, in the first step, an interest packet enters from an interface, in the second step, it is checked against the CS, if a match is found the re quested data packet is fetched and the NDN node responds by sending back the data packet through the interface which the interest packet has received. Otherwise, in the third step NDN name is added to the 2DNCF-PIT table and in the fourth step Interest packet will be sent to the FIB table, and in the last step it is forwarded to its destination. This process continues for every incoming Interest packet from any interface.

5.2 Comparison of 2DNCF with Cuckoo Filter

In this subsection, the 2DNCF filter is compared to the Cuckoo filter. For implementation, the Python programming language on an 18.04 LTS Linux operating system is used. The standard Cuckoo filter uses 6-bits of fingerprint, and each bucket also has four entries [8].

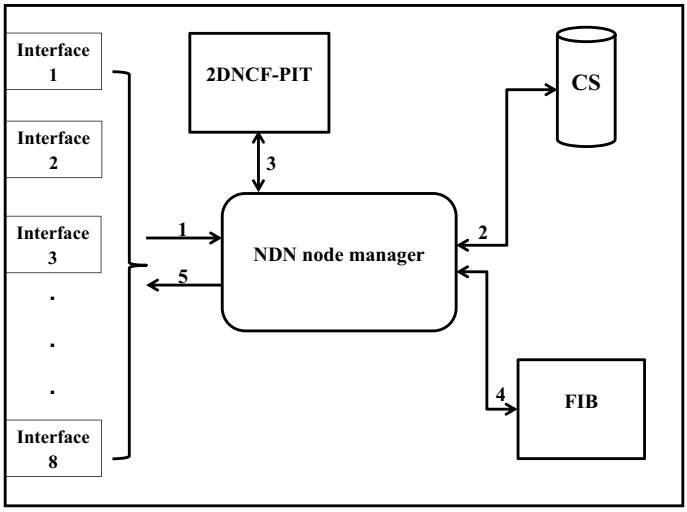


Fig. 6 The router architecture with eight interfaces that utilized 2DNCF

5.2.1 Load Factor

Load factor is the rate of the number of items to the number of addresses in the filter which is defined as Eq. 4.

$$\text{Load factor} = \frac{n}{m} \tag{4}$$

In Eq. 4, n shows the number of items that have been inserted into the filter successfully, and m shows the size of the filter or the total number of addresses. To obtain the amount of load factor, if the insertion in the filter is successful, it is added to the value of n one unit, and finally, the number n is divided by the size of the filter (m). Figure 7 depicts the load factor of both Cuckoo and 2DNCF filters in terms of the different numbers of items.

Comparing the load factor of both filters shows that the proposed 2DNCF filter has a higher load factor. This is because the Cuckoo filter examines the first and second candidate buckets. If both the first and second candidate locations are full, one of the buckets is selected randomly. It selects an entry from the selected bucket randomly and replaces the selected entry with a new item. The Cuckoo filter now tries to relocate the items, k times to insert the ejected item into the filter and also to retain the existing items, otherwise, one of the items will not be inserted in the Cuckoo filter. In the 2DNCF filter, in addition to being selected by its address, each bucket can be chosen by other addresses as neighbors. That is, each address is chosen by 8 adjacent neighbors. Therefore, the load factor of the proposed 2DNCF filter is higher than the Cuckoo filter. According to [8] in this experiment, the fingerprint size is 6 bits, and each bucket also has 4 entries. This experiment was performed with 100,000, 200,000, 500,000, 1,000,000, 2,000,000 and 50,000,000 items. To perform this experiment, first a filter of the size $\frac{m}{4}$ of buckets is made, because each

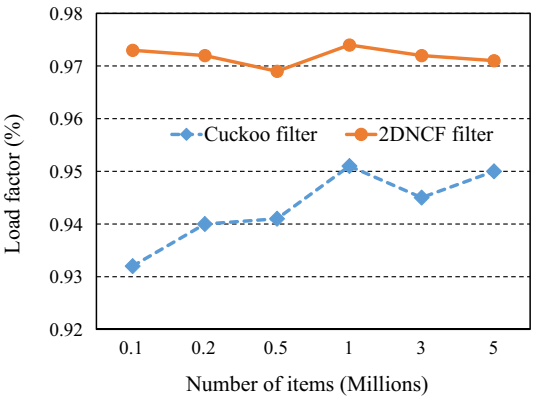


Fig. 7 Load factor of both Cuckoo and 2DNCF filters

bucket has 4 entries. According to [9], the number of bits per item is equal to $C = \frac{f}{Load\ factor}$. Because the load factor of the proposed filter is higher, so this consumes less memory than the Cuckoo filter. It can be concluded that the proposed filter uses fewer bits per item.

5.2.2 False Positive

To calculate the false positive rate, we search for values that are not in the filter. If the filter returns a successful search, then a false positive has occurred. In this experiment, the Alexa dataset with one million items with load factors between 0.01 and 0.95% was performed. Figure 8 depicts the false positive rate for the Cuckoo filter and 2DNCF filter.

As shown in Fig. 8, the 2DNCF filter has a lower false positive rate than the Cuckoo filter in the same conditions and load factor. The average improvement of the 2DNCF in comparison to the standard Cuckoo filter is 48%. This is because, the 2DNCF utilizes a neighbor bucket which increases the length of the bucket, and finally the chance of collision and false positive is decreased.

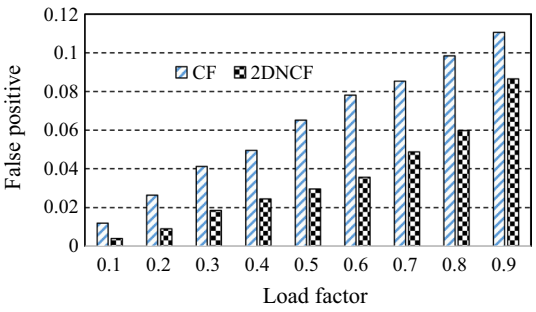


Fig. 8 False positive rate of both Cuckoo and 2DNCF filters

It should be noted that in the Cuckoo filter and other probabilistic filters e.g., Bloom and quotient filters, the false negative cannot exist. In other words, a query returns either "possibly in set" or "definitely not in set".

5.2.3 The Time of Insertion, Lookup, and Deletion

In this subsection, the time of insertion, lookup, and deletion operations is compared to other approaches. One reason for introducing the 2DNCF filter is that the second hashing function is less used than the Cuckoo filter. This reduces the time to insert, search and delete items or relocation between items because the calculation of the second hashing function for relocation between filter items is time-consuming. Therefore, first it is checked whether 2DNCF uses the second hashing function less or not, and then it is checked how many items use the relocation of items in the filters. To do this, the number of m items is entered in the Cuckoo filter and the proposed 2DNCF filter, each of which has an $\frac{m}{4}$ bucket. In the insertion phase, the number of items that used the first hashing function is stored in the variable *firsthash*, the number of items that used the second hashing function in the *secondhash* and the number of items that use the relocation operation is stored in the relocation variable. Figure 9 depicts the rate of hashing functions utilization, and relocation in 2DNCF, and Cuckoo filter.

According to Fig. 9, it can be concluded that the proposed 2DNCF filter uses the second hashing function and relocation less than the Cuckoo filter. In this experiment, the number of items is 500,000, 1,000,000 and 2,000,000, and because each bucket has 4 entries, the size of the filters is 125,000, 250,000, and 500,000 buckets, respectively. Based on Fig. 9, in the 2DNCF filter, the use of the first hashing function has increased by 17%, while the use of the second hashing function has decreased by 5%, and the number of the relocation operations has also decreased by 5%.

Figure 10 depicts the required time of insert operation in both 2DNCF, and Cuckoo filters.

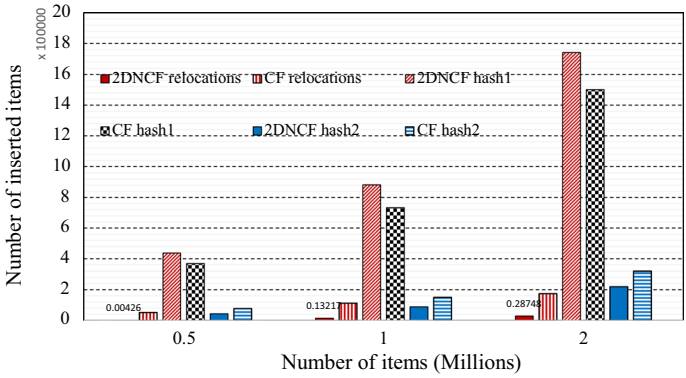


Fig. 9 The rate of hashing functions utilization, and relocation in 2DNCF, and Cuckoo filter

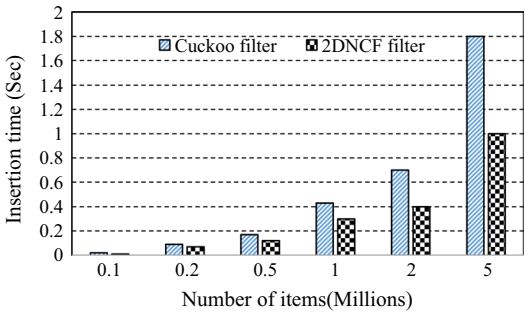


Fig. 10 The insertion time in both 2DNCF and Cuckoo filters for different number of items

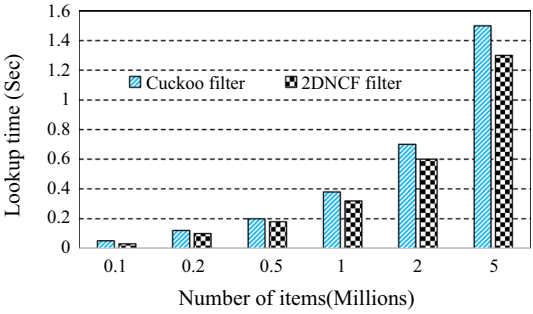


Fig. 11 The lookup time in both 2DNCF and Cuckoo filters for the different number of items

In this experiment, the fingerprint size is equal to 6 bits and each bucket has 4 entries. Figure 10 shows that the insertion time in the proposed 2DNCF filter compared to the Cuckoo filter is reduced by 40%.

Figure 11 depicts the lookup time for both 2DNCF and Cuckoo filters.

From Fig. 11, it can be observed that the search time in the 2DNCF filter is reduced because, the number of using of the second hashing function has decreased. This means that during the search, more items are found by calculating the first hashing function. On the other hand, more lookups can be found in the queried bucket or its neighbors. From Fig. 11 the amount of search time in the 2DNCF filter has been reduced by 12% compared to the Cuckoo filter.

Figure 12 depicts the deletion time for both 2DNCF, and Cuckoo filters.

Figure 12 shows that the 2DNCF decreases the deletion time by %10 in comparison to the Cuckoo filter. The reason is similar to the insertion and lookup operations. In other words, in 2DNCF, the second hashing function is used less than the Cuckoo filter. This is because that the 2DNCF filter for each item uses the neighbor bucket which increases the bucket length and consequently, the use of second hashing functions is decreased.

5.3 2DNCF-PIT Implementation

In this section, the PIT table, which is based on the 2DNCF is compared to DiPIT [15], hashtable [12], and FTDF-PIT [2]. To implement 2DNCF-PIT, Python

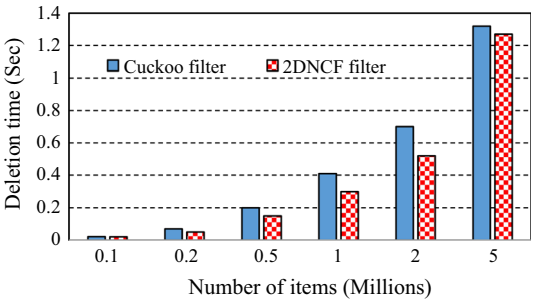


Fig. 12 The deletion time in both 2DNCF and Cuckoo filters for the different number of items

programming language is used in a Linux 18.04 LTS system. The scenario has used a router with 8 interfaces (8 bits are needed to list the interfaces in the PIT tables). In implementing DiPIT [15], 4-bit counting Bloom filters with a shared Bloom filter are used. The PIT implemented based on the hash table does not have a fixed size. Therefore, according to [3], on average, 128 bits are considered for each name in the hash table, as well as 8 bits for the list of interfaces. To implement the FTDF-PIT, one hashing function is used. These names are placed in interest packets using the packet generator in the NDNSim emulator 2.6, and these interest packets are sent over the network.

5.3.1 2DNCF-PIT Insertion Time

Figure 13 depicts the insertion time of 2DNCF-PIT in comparison to other approaches.

Data with 500,000, 1,000,000, 2,000,000, and 5,000,000 names are inserted in the PIT tables. As shown in Fig. 13, the proposed PIT table insertion time is lower than the other approaches. In this experiment, 4 hashing functions were used for DiPIT. The PIT, based on the hash table, has the worst possible time to insert items among the presented approaches. The proposed 2DNCF-PIT table has less insertion time than the others because it uses less the second hashing function and the relocation operations.

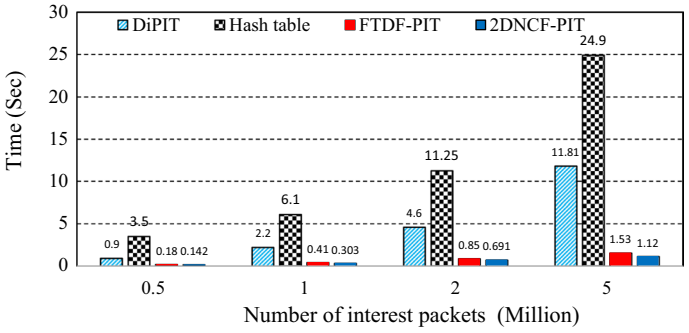


Fig. 13 Insertion time of 2DNCF-PIT in comparison to other approaches

5.3.2 2DNCF-PIT Lookup Time

In this experiment, the proposed tables are first filled with 5,000,000 names. To lookup for data, 500,000, 1,000,000, 2,000,000, and 5,000,000 names are searched in each table. In this experiment, as in the previous experiment, 4 hashing functions are used for DiPIT, and a 32-bit hashing function is used to implement the hash table PIT. Figure 14 depicts the lookup time of 2DNCF-PIT in comparison to other approaches.

As shown in Fig. 14, 2DNCF-PIT improves the lookup time in comparison to the other approaches.

5.3.3 2DNCF-PIT Memory Usage Evaluation

In this experiment, the amount of memory consumed in each of the proposed PIT tables is evaluated. The proposed 2DNCF-PIT table for each name uses 6 bits of fingerprint, 8 bits for the list of interfaces. The amount of memory used for each name in the 2DNCF-PIT table is based on the relation $\frac{f}{\text{load factor}}$ [2] plus 8 bits to store the list of interfaces. The amount of memory consumed for DiPIT is equal to the amount of memory consumed by eight 4-bit counting Bloom filters and one shared Bloom filter. In hash table-based PIT, 128 bits for each name, and 8 bits as a list of interfaces are also used. For FTDF-PIT, each entry uses eight bits. Figure 15 depicts the amount of memory used by the proposed 2DNCF-PIT in comparison to other approaches.

As shown in Fig. 15, the proposed 2DNCF-PIT decreases memory usage in comparison to other approaches.

5.4 NDN Router Implementation Approaches

In this subsection the different NDN router implementation approaches are discussed.

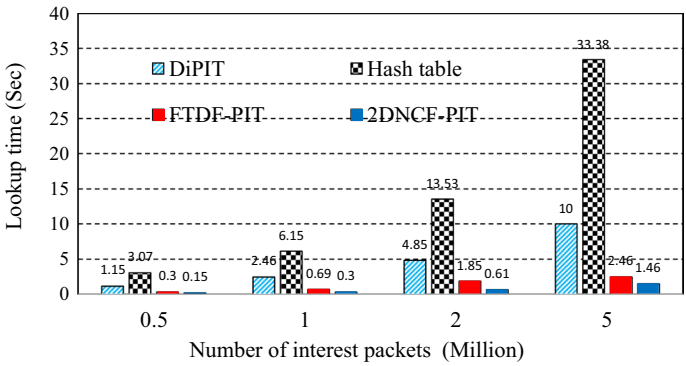


Fig. 14 Lookup time of 2DNCF-PIT in Comparison to other approaches

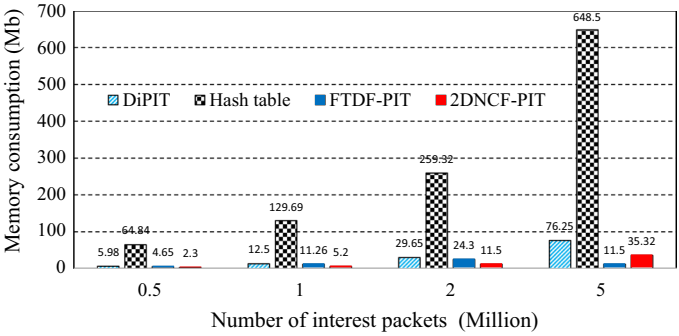


Fig. 15 Memory usage of 2DNCF-PIT in comparison to other approaches

The NDN router implementation can be viewed from different aspects.

1. Implementation of the 2DNCF accelerator in hardware as a part of hardware NDN router.
2. Implementation of NDN router and 2DNCF as a virtualized network function in network function virtualization.
3. Implementation of NDN router on commodity hardware.

In [23], an FPGA-based hardware accelerator to lookup packets with 56–60 MPPs has been designed. This accelerator, utilized an on-chip Bloom filter and an off-chip linear-chained hash table in the PIT design. In this design, the hardware integrates the lookup and the associated update operation into one command. The main core of this architecture is a supervisor engine which includes a network processor unit, an FPGA, a PIT, an nidT, a memory unit of PIT, and a fabric arbiter.

In [24, 25], two software approaches to design NDN routers are presented. The first paper, presents three scenarios to implement a named data network function (NDNF) as: End-system NDNF, edge-router NDNF, and any-router NDNF, which the first one the end-system explicitly invokes the network function, the second one, the edge router of a network invokes the network function, the third one, any router in the local network can invoke network functions, respectively. This approach enables network functions for any type of traffic, even traffic that does not traverse the edge router. The second paper, implements a prototype of the building blocks of an NDN software router and empirically measures the consumption of CPU cycles and dynamic random-access memory (DRAM) channels. Afterward, proposes some proposals to design the high-speed software routers.

In [26], a software NDN-router with high-speed forwarding engine that capable of forwarding rate 100 Gbs on commodity hardware. They utilize some optimization techniques, efficient data structures and parallelization approaches on a multi-core CPU to achieve this forwarding rate. They use Data Plane Development Kit (DPDK) to implement the proposed NDN software router. The DPDK has been extended to support some features in NDN networks, such as the introduction of the PIT token for efficient Interest-Data prefix matching, the introduction of indirect CS entries for

efficient CS prefix matching, resulting in more effective in-network caching, and the secure support for NDNs forwarding hints. The experimental results show that the NDN–DPDK can sustain 1.8 Mpps, or a corresponding 108 Gbps when 8 KB Data packets.

It should be noted that both the Cuckoo filter and Bloom filter are membership query data structures. Based on the applications, there are different solutions to implement Bloom filter in hardware. In [27–29] were presented several hardware implementations for Bloom filters. In [29], the energy, delay, and area characteristics of two implementations for Counting Bloom Filters (CBFs) using full custom layouts in a commercial 0.13-μm fabrication technology are studied. The proposed S-CBF, uses an SRAM array of counts and a shared up/down counter. They also proposed a low power Counting Bloom filter (L-CBF) which is an energy- and delay-efficient implementation. L-CBF utilizes an array of up/down linear feedback shift registers (LFSRs) and local zero detectors. In [28], the Cuckoo filter is extended to integrate a Bloom filter that is used to improve the performance of insertions. This variant has been called CFBF. The CFBF targets hardware implementations where the Bloom filter can be checked with negligible cost and where the memory width can also be adjusted to the bucket size. In[27], a hardware implementation of parallel Bloom filters for deep packet inspection is presented. The focus of this paper is to present the architecture of hardware implementation of Bloom filters and analyze its performance. The analysis for the Field Programmable Gate Arrays (FPGAs) shows that OC-48 line speed scanning for more than 10,000 strings can be achieved with this technique. Based on the mentioned works in the NDN router and Bloom filter implementations, the 2DNCF filter and NDN router based on the 2DNCF can be implemented at different levels. The 2DNCF can implement PIT as an NDN accelerator engine, or independently can be implemented and used as a high-speed membership checking engine. Due to the advances in network function virtualization, and the future of the Internet, the 2DNCF-PIT and the NDN router can be implemented as a named data network function (NDNF) and executed in the network function virtualization environment (NFV). In the NFV, all networking functions as well as named data networking functions can be defined as virtualized network functions and utilized the Cloud-based system.

6 Conclusion

Named data networking is a new paradigm based on the content to content communication that influences the future Internet. The NDN consists of three main data structures including: CS, forwarding information base, and the PIT. One of these tables is the PITs which have an important role in delivering the packets. On the other hand, the membership query data structures e.g., Bloom, Quotient, and Cuckoo filters have been used in recent years in different network applications to improve the performance. This paper presented a new variant of the Cuckoo filter called two-dimensional neighbor-based Cuckoo filter (2DNCF), which utilizes the physical neighbors of the selected bucket of Cuckoo filter. The 2DNCF utilizes the neighbor buckets to improve the insert, lookup, and delete operations in the filter.

Subsequently, the 2DNCF variant is used to design the PIT table in NDN, which is called 2DNCF-PIT. The results show that the 2DNCF outperforms the Cuckoo filter and 2DNCF-PIT outperforms some important existing PIT table approaches. As the overall result, the 2DNCF can be used in both FIB and PIT tables simultaneously using the same hashing functions and similar configurations.

References

1. Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N.H., Braynard, R.L.: Networking named content. In: CoNEXT '09: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies, pp. 1–12 (2009)
2. Shubbar, R., Ahmadi, M.: A filter-based design of pending interest table in named data networking. J. Netw. Syst. Manag. **27**(4), 998–1019 (2019)
3. Azgin, A., Ravindran, R., Wang, G.: PIT/less: stateless forwarding in content centric networks. In: 2016 IEEE Global Communications Conference (GLOBECOM), pp. 1–7. IEEE, Washington, DC (2016)
4. Mansour, D., Osman, H., Tschudin, C.: Load balancing in the presence of services in named-data networking. J. Netw. Syst. Manag. **28**, 298–339 (2020). https://doi.org/10.1007/s10922-019-09507-x
5. Saxena, D., Raychoudhury, V.: Radient: scalable, memory efficient name lookup algorithm for named data networking. J. Netw. Comput. Appl. **63**, 1–13 (2016). https://doi.org/10.1016/j.jnca.2015.12.009
6. Dai, H., Liu, B., Chen, Y., Wang, Y.: On pending interest table in named data networking. In: 2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 211–222. IEEE, Austin (2012)
7. So, W., Narayanan, A., Oran, D.: Named data networking on a router: fast and dos-resistant forwarding with hash tables. In: Architectures for Networking and Communications Systems, pp. 215–225. IEEE, San Jose (2013)
8. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: practically better than bloom. CoNEXT **14**, 75–88 (2014)
9. Mun, J.H., Lim, H.: New approach for efficient IP address lookup using a bloom filter in trie-based algorithms. IEEE Trans. Comput. **65**(5), 1558–1565 (2016)
10. Kwon, M., Reviriego, P., Pontarelli, S.: A length-aware cuckoo filter for faster IP lookup. In: 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 1071–1072. IEEE, San Francisco (2016)
11. Li, Z., Liu, K., Zhao, Y., Ma, Y.: Mapit: an enhanced pending interest table for ndn with mapping bloom filter. IEEE Commun. Lett. **18**(11), 1915–1918 (2014)
12. So, W., Narayanan, A., Oran, D., Wang, Y.: Toward fast NDN software forwarding lookup engine based on hash tables. In: 2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 85–86 (2012)
13. Alubady, R., Hassan, S., Habbal, A.: Pending interest table control management in named data network. J. Netw. Comput. Appl. **111**, 99–116 (2018)
14. Tan, Y., Li, Q., Jiang, Y., Xia, S.: Rapit: Rtt-aware pending interest table for content centric networking. In: 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC), pp. 1–8. IEEE, Nanjing (2015)
15. You, W., Mathieu, B., Truong, P., Peltier, J.F., Simon, G.: Dipit: a distributed bloom-filter based PIT table for CCN nodes. In: 2012 21st International Conference on Computer Communications and Networks (ICCCN), pp. 1–7. IEEE, Munich (2012)
16. Sivaraman, V., Guha, D., Sikdar, B.: Optimal pending interest table size for icn with mobile producers. IEEE/ACM Trans. Netw. **28**(4), 1615–1628 (2020)
17. Buragohain, M., Nandi, S.: Quality of service provisioning in named data networking via PIT entry reservation and PIT replacement policy. Comput Commun. **155**, 166–183 (2020)
18. Yuan, H., Crowley, P.: Scalable pending interest table design: from principles to practice. In: IEEE INFOCOM 2014—IEEE Conference on Computer Communications, pp. 2049–2057. IEEE, Toronto (2014)

19. Li, Z., Xu, Y., Zhang, B., Yan, L., Liu, K.: Packet forwarding in named data networking requirements and survey of solutions. IEEE Commun. Surv. Tutor. **21**(2), 1950–1987 (2019). https://doi.org/10.1109/COMST.2018.2880444
20. Almeida, P.S., Baquero, C., Preguia, N., Hutchison, D.: Scalable bloom filters. Inf. Process. Lett. **101**(6), 255–261 (2007)
21. Knuth, D.E.: The art of computer programming. In: Fundamental Algorithms, 3rd edn. Addison Wesley, Boston (1997)
22. Varvello, M., Perino, D., Linguaglossa, L.: On the design and implementation of a wire-speed pending interest table. In: 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 369–374. IEEE, Turin (2013). https://doi.org/10.1109/INFCOMW.2013.6970719
23. Yu, W., Pao, D.: Hardware accelerator to speed up packet processing in ndn router. Comput. Commun. **91**(C), 109–119 (2016). https://doi.org/10.1016/j.comcom.2016.06.004
24. Fang, P., Wolf, T.: Enabling virtual network functions in named data networking. In: IEEE INFOCOM 2021—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 1–6. IEEE, Vancouver (2021)
25. Taniguchi, K., Takemasa, J., Koizumi, Y., Hasegawa, T.: A method for designing high-speed software NDN routers. ACM-ICN '16, pp. 203–204. 10.1145/2984356.2985234 (2016)
26. Shi, J., Pesavento, D., Benmohamed, L.: NDN–DPDK: NDN forwarding at 100 gbps on commodity hardware. In: Proceedings of the 7th ACM Conference on Information-Centric Networking, ICN '20, pp. 30–40 (2020)
27. Dharmapurikar, S., Krishnamurthy, P., Sproull, T., Lockwood, J.: Deep packet inspection using parallel bloom filters. IEEE Micro **24**(1), 52–61 (2004). https://doi.org/10.1109/MM.2004.1268997
28. Reviriego, P., Martnez, J., Pontarelli, S.: CFBF: reducing the insertion time of cuckoo filters with an integrated bloom filter. IEEE Commun. Lett. **23**(10), 1857–1861 (2019). https://doi.org/10.1109/LCOMM.2019.2930508
29. Safi, E., Moshovos, A., Veneris, A.: L-CBF: a low-power, fast counting bloom filter architecture. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp. 628–638. IEEE, Hoboken (2008). https://doi.org/10.1109/TVLSI.2008.2000244

Aramn Mahmoudi has received his BSc from Kurdistan University in 2015, and his MSc from Razi University in 2019, respectively. His research interests are: Named data networking, and software defined networking.

Mahmood Ahmadi received his Ph.D. in May 2010 from Delft University of Technology. His research interests include network processing, probabilistic filters, software-defined networking, named data networking, and high-performance computing. He is working as an associate professor at Computer Engineering and Information Technology Department in Razi University, Kermanshah, Iran. He has published more than 80 conference and journal papers.

Authors and Affiliations

Arman Mahmoudi[1] · Mahmood Ahmadi[1]

Arman Mahmoudi
Arman.Mahmoudi@outlook.com

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