Length Indexed Bloom Filter Based Forwarding In Content Centeric Networking

Vishal Manghnani
Rochester Institute of Technology
vjm1952@rit.edu

Abstract—Named Data Network (NDN) is a modern networking architecture that makes use of name prefixes instead of host IPs for communication purpose. Traditional NDN require variable length name-based lookup with unlimited length and large scale forwarding tables. Also, scaling is an issue where longest prefix match is a bottleneck at every node because of hierarchical naming with possibly infinite length, which could make search slower. To eliminate these challenges, we employ bloom filters that complements hash table search at each router. The aim is to come up with an optimized search strategy requiring longest prefix match starting from the optimum length and proceeding in the manner that will reduce the redundant searches. To achieve this, we propose LIBF (Length Indexed Bloom Filter) in the form of a binary tree structure where the tree is constructed based on length of name prefix in bloom filter at each node resulting in lower average false positive rates and reduction in lookup time. We simulate the process using NDNSim module in NS-3 framework using our approach and compare our version of Longest Prefix Match (LPM) with traditional NDN hash table search and single bloom filter search and show that our approach is faster.

I. INTRODUCTION

Named data network (NDN) is an architecture that uses name prefixes in place of host IPs for communication [1]. NDN naming is hierarchical and unlimited in length where each name is divided into components separated by delimiter “/”, similar to the URL. This kind of name structure causes a slowdown in performing longest prefix match as names are of variable length and unbounded [2], [3]. NDN uses FIB (Forwarding Information Base) which is similar to forwarding table in IP network. The length of name prefixes can be much larger than fixed length IP address. This results in huge forwarding table size, which is in the order of billions. Adding more content in future could result in the fast growth of forwarding table size. Therefore scaling NDN to support the huge size of forwarding table can be an issue [2], [3]. Longest prefix match is very time consuming and causes a bottleneck in NDN packet forwarding. An alternative solution is to employ a complementary data structure bloom filter on-chip [4], which stores forwarding table entries in bit vector and support space efficient longest prefix match. However, bloom filter comes with certain limitations like False Positive Rate (FPR) and no support for deletion of an element [5]. Using bloom filter can solve deletion of elements but it results in an increase in false positive rates as it uses counter [5]. Due to a large number of collisions in bloom filter insertion, false positive rates increase. Changing the number of hash functions or the size of the bloom filters can vary FPR. But hash computation in bloom filter is time consuming hence increasing the number of hash functions can result in more time [5].

Due to this tradeoff between space and time we need to come up with a better approach that improves the longest prefix match performance along with optimization of the space-time compared to traditional NDN hash table used. We propose Length Indexed Bloom Filter (LIBF) which aims at improving longest prefix match by reducing the FPR and increasing the lookup speed at the cost of a slight increase in memory compared to single Bloom Filter approach. LIBF arranges the set of counting bloom filter in a balanced binary tree fashion. Longest prefix match starts from a particular length and continues until the algorithm reaches either the shortest or longest search length. We simulate LIBF using NDNSim based on NS-3 and which is written in C++. We replace the current approach with our own approach and obtain the results. We see that our approach using LIBF has a significant reduction in false positive rate and also there is an increase in speed of LPM at the cost of an increase in memory as compared to the single bloom filter approach.

The rest of the paper is organized as follows. We first explain the background information required pertaining to our approach in Section II. Then we introduce Length indexed based bloom filter design in section III. Section IV provides details of our simulation methods to test our algorithm. Section V provides detailed explanation of our results. Section VI concludes the discussion along with the future work followed by acknowledgments in section VII.

II. BACKGROUND

A. Hash table based lookup

Currently, a hash table with the key as name prefix and value as interface number is used to store the forwarding table and search them during forwarding. Name prefix is in the form of a string. Hash collisions are resolved using chaining and each hash value is mapped to a particular bucket [1], [3]. If there are any hash collisions then it is appended to the end of linked list in the mapped bucket. Each name prefix is mapped to the next interface or port where the packet will be transferred if longest prefix matched has the corresponding interface. Figure 1 depicts the packet flow from consumer to the publisher. At each router forwarding table, FIB is looked up for next route.
Now let us assume we have to search the name prefix with the K components delimited by /. The lookup starts with K components and if it is found then the name prefix is returned otherwise the lookup is done for K-1 components and this continues until any length prefix is found [1], [3], [6], [7]. For example, hash table contains prefixes /a/, a/b, a/b/c and request for the name prefix a/b/c/d/e. It will first search a/b/c/d/e and search will fail then it will search for a/b/c/d and henceforth when the a/b/c/ will be matched and the correct port mapping will be returned.

![Fig. 1. NDN forwarding](image1)

There are many challenges in using the hash table for traditional NDN forwarding. The lookup starts with K components and keeps on reducing the number of components until the match is found. To match K components, worst case it takes is K lookups. There is some extra time taken D for resolving hash collisions. Hence the overall worst case time complexity of O(KD) where K is the number of components and D is the time taken for resolving hash collisions [2], [3]. Each entry in hash table takes up space for name prefix string, interface number, a pointer to K-1 entry for traversing List. Also, the memory increases when there are too many hash collisions as there is wastage of memory. So overall memory consumed is much higher [2], [3]. Unlike fixed length IP address, NDN has variable K length name prefixes which can be infinitely long. Hence the number of memory access or hash probes is quite high and in the case of no match there are about K memory accesses which could go up to infinity in poorly designed data structure [2], [3]. Hash values are calculated using hash functions. But since the NDN name prefixes are variable length unlike fixed length IP addresses, each name prefix takes different amount of time to calculate hash values and during longest prefix match, we have to traverse each component and calculate a hash value, for all K components which add to the processing time [2], [3].

B. Standard bloom filter based lookup

Bloom filter is a compact probabilistic data structure that determines efficiently if the element is part of the set or not. Bloom filter can be added as a complementary data structure to FIB hash table for increasing lookup speed and improving the performance of Longest prefix match. Once the longest prefix match is found then FIB hash table could be looked up to get the interface where the packet needs to be forwarded.

The lookup starts with the lowest component and keeps on incrementing the number of components until a match is found [4], [8], [9]. In the worst case, it takes K lookups to match k components. Figure 2 shows how the LPM is performed in bloom filter. Hence the overall worst case time complexity is O(K). Since Bloom filter uses a bit vector to store the name prefixes, hash values for name prefixes are calculated and those index bits are set. Hence there is no extra memory usage for variable name prefix [5]. The memory remains amortized constant as the memory for each bloom filter is initially set to expected number of entries [10]. Also, the bloom filter is a probabilistic data structure [10], it has a false positive probability table. Changing the size of filter and number of hash functions can vary the false positive rate.

![Fig. 2. Longest prefix match in bloom filter](image2)

III. LENGTH INDEXED BLOOM FILTER TREE

We now introduce a improved data structure, Length Indexed Bloom Filter tree and as the name suggests it contains the nodes arranged in the form of a binary tree and each node contains the bloom filter that stores the name prefixes of a particular number of components as shown in Figure 3. Each node has two branches right and left. Right node has the bloom filter that stores the name prefixes of length greater than the length of name prefixes stored in bloom filter in current node and left node has less than the length of the current node.

Bloom filters are arranged based on the number of name components and name prefix with the equal number of components are stored in same bloom filter. Therefore if there are K components in a name prefix, then LIBF will have at least K bloom filters. To do longest prefix match, binary search is done on these bloom filter tree starting from the root node. At each bloom filter node if there is matching prefix then the search proceeds to the right subtree to look for the longer prefix match. But if there is no match found in the current bloom filter node then the lookup is done in left subtree for a shorter prefix match. The lookup is performed until the search ends in shortest prefix or longest prefix. Total time taken by
lookup is $\log(k)$ as each prefix search traverse the tree up to $K$ components. As we can see from the Figure 4 that during longest prefix match, the search starts from length 4 and then proceeds to length 6 until it stops at length 7.

For example, Let us suppose that we want to search a/b/c/d/e/f/g then the search starts from the root node a/b/c/d. Since it is present in bloom filter of length 4, the search proceeds to the right subtree and searches node of length 6. It successfully finds the prefix of length 6, hence it moves to the node of length 7. The search terminates at this node as the longest prefix of length 7 is found in the LIBF tree.

There is a possibility of error when it comes to search starting from node 4 does not contain the sub prefix a/b/c/d for matching a/b/c/d/e/f/g. In that case, the search would traverse through the node 2 and then 1 and it would not find the correct longest prefix match. Hence, in that case, we need to create additional marker entries to ensure that the shorter prefix match is available so the search successfully traverses in the right direction. Marker entries are added during the insertion of the forwarding table entries in the LIBF to make sure that there are no missing entries that can halt the longest prefix match process. For example, when the a/b/c is inserted, we have to traverse to node 3. But that traversal passes through node 2 hence we need to add marker entries in node 2 as a/b that will assist in LPM process to proceed to node 3 from node 4. Hence, the use of marker entries is necessary to do longest prefix match in LIBF.

As shown in Figure 5 after longest prefix match in LIBF is done, FIB hash table is looked up for next hop or face where the packet needs to be sent. If name prefix is not found in the hash table resulting from false matching in LIBF, then search continues in LIBF again starting from length greater than the previous length found. This, however, increments the number of hash probes until the correct longest prefix is found by LIBF. Due to the false positive rate at each bloom filter which adds to the overall false positive rate and increases the number of hash probes. Moreover, when a new entry in LIBF is inserted, the marker entries have to be updated to support the new entries with similar prefixes. This puts an extra load on each bloom filter as redundant entries are added, hence before insertion; each marker entry is checked if already present then reentry is skipped. False positive rate of each bloom filter affects the longest prefix match performance. False positive of each bloom filter is different based on the number of entries and number of hash functions used. In that case, the FIB hash table will be accessed to find that element is not present and hence the number of hash probes would rise.

The main advantage of LIBF is that search can start from any length. If we start the search from the length which has the highest frequency in current router then this will enable search in LPM to avoid redundant searches and save lookup time. Therefore we calculate the frequency distribution of the name prefixes at each router and set the start length with the length having highest frequency, thus avoiding the redundant searches. Furthermore, there is a significant improvement in
lookup speed that happens due to binary search in LPM as it traverses only \( \log(K) \) components compared to traversal of \( K \) components in the hash table and single bloom filter. So in a case, where the maximum number of components is 32, our LIBF will have to do maximum 5 lookups while the single bloom filter or hash table will take a maximum of 32 lookups. This is a huge improvement in terms of time saving and an efficient use of search space. Finally, forwarding table name entries are distributed among multiple bloom filters based on the length so the load at each bloom filter is reduced. This reduces the false positive rate at each bloom filter and thus helps in reducing the overall false positive rate.

### Algorithm 1 Longest prefix match in LIBF

**procedure** \textsc{LongestPrefixMatch}(name, startlength)

\[
\text{length} \leftarrow \text{startlength} \\
\text{node} \leftarrow \text{getNode(startlength)} \\
\text{while true do} \\
\quad \text{if node} \in \text{visited then} \\
\qquad \text{return length} \\
\quad \text{visited(node)} \leftarrow \text{true} \\
\quad \text{if substring(name, length) \in node(bloomfilter) then} \\
\qquad \text{length} \leftarrow \text{node(length)} \\
\qquad \text{if node(right) is null then} \\
\qquad \quad \text{successor} \leftarrow \text{getSuccessor(node)} \\
\qquad \quad \text{if successor is null or visited then} \\
\qquad \qquad \text{return length} \\
\qquad \quad \text{else} \\
\qquad \qquad \text{node} \leftarrow \text{successor} \\
\qquad \text{else} \\
\qquad \quad \text{node} \leftarrow \text{node(right)} \\
\qquad \text{else} \\
\qquad \quad \text{if node(left) is null then} \\
\quad \quad \text{predecessor} \leftarrow \text{getPredecessor(node)} \\
\qquad \quad \text{if predecessor is null or visited then} \\
\qquad \qquad \text{return length} \\
\qquad \quad \text{else} \\
\qquad \qquad \text{node} \leftarrow \text{predecessor} \\
\qquad \text{else} \\
\qquad \quad \text{node} \leftarrow \text{node(left)}
\]

### IV. Simulation

For Simulation we need a network that can be compared to the real world NDN. We use NDNSim based on NS-3 written in C++ language and it is designed to simulate close to real world NDN applications [11]. It provides a common user-friendly framework that implements major NDN primitives that are used in various NDN applications. NDNSim can be used to implement and test various algorithms that support NDN protocols [11]. It supports multiple data structure used in NDN applications such as FIB (Forwarding Information Base), PIT (Pending Interest table) and CS (Content store). It supports multiple forwarding strategies and management protocols. NDNSim also provides several helper classes and traffic simulators that can be used in designing and testing algorithms with different scenarios [11]. Bloom filter and LIBF tree are written in C++ so it can be then easily integrated with NDNSim. It will be used to test our new algorithms with Bloom filter.

We have to design our data structure so that it is easily plugged and removable to test various algorithms. Also, we need to take care that it integrates with the current architecture of NDNSim and does not affect any other functionality of NDN. We carefully designed loosely coupled classes as shown in UML diagram in Figure 6. The data structure designed in our project is LIBF. As we know the FIB data structure, it complements the FIB hash table so the new addition should not affect the functionality of hash table FIB. Name prefix should be inserted into our new data structure LIBF along with the insertion in hash table at the time of initialization.

To evaluate the performance of longest prefix match we have to run NDNSim application and provide the data and topology. We used GEANT network topology with 22 nodes representing routers, and links between is defined based on the European standards. NDN name prefixes are similar to file system names or URL names delimited by / and divided into name components. For our application, we generated \( 10^5 \) name prefixes using find command from our Unix file system. These name prefixes are then loaded using ApplicationHelper in NDNSim. It loads all the name prefixes inside FIB hash table. GlobalRoutingController in NDNSim [11] then calculates routes in the network. This route calculation loads our data structure bloom filter or LIBF based on runtime configurations. Once the simulation starts, each node runs longest prefix match and metrics measured are memory usage, time taken, false positive rate and the number of hash probes are measured. We then run the analysis of the metric file and generate the final results.

### V. Results

To evaluate the algorithm we have created \( 10^5 \) name prefixes using the Unix file system "find" command, since names in the prefixes are similar to file system. For e.g. names from file system are like "/usr/share/lib" which contains 3 name
components separated by delimiter "/". For result evaluation, we compare 2 different approaches hash table with bloom filter and hash table with LIBF. Metrics used in the experiment are Memory consumption, false positive rate and the number of hash probes.

A. Memory consumption

Memory consumption is the amount of space used by each of the data structure. We compare memory consumption of bloom filter vs LIBF. From the Table I we can see that LIBF takes almost double amount of memory compared to the bloom filter. Bloom filter stores all the entries in single bit vector which puts lot of load on single filter therefore in LIBF we split the data based on its length and put it into several bloom filter. For our experiment we precalculated the number of entries expected and assigned each bloom filter the size equal to twice number of entries expected. This will keep the false positive rate of bloom filter to minimum [10].

![Fig. 7. Memory consumption](image)

<table>
<thead>
<tr>
<th>Input size</th>
<th>Bloom Filter</th>
<th>LIBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000</td>
<td>79.077</td>
<td>148.275</td>
</tr>
<tr>
<td>31000</td>
<td>222.853</td>
<td>417.855</td>
</tr>
<tr>
<td>51000</td>
<td>366.629</td>
<td>687.435</td>
</tr>
<tr>
<td>71000</td>
<td>510.405</td>
<td>957.015</td>
</tr>
<tr>
<td>91000</td>
<td>654.181</td>
<td>1226.6</td>
</tr>
</tbody>
</table>

Table I: Memory consumption

![Table II: False positive rate](image)

<table>
<thead>
<tr>
<th>Number of hash functions = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input size</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>11000</td>
</tr>
<tr>
<td>31000</td>
</tr>
<tr>
<td>51000</td>
</tr>
<tr>
<td>71000</td>
</tr>
<tr>
<td>91000</td>
</tr>
</tbody>
</table>

Table II: False positive rate

B. False positive rates

We compare the false positive rate of single bloom filter with LIBF. We calculate the false positive rates (fpr) for each data structure with varying the number of functions as we want to see the effect of number of hash functions on the fpr of each data structure. From figure 8 we can see that even after increasing number of hash functions, fpr of single bloom filter is higher compared to LIBF. That shows the LIBF reduces the false positive rate by distributing the load among multiple bloom filter.

![Fig. 8. False positive rate](image)

C. Number of hash probes

Hash probes is number of times hash table is accessed during longest prefix match process. We compare the number of hash probes for traditional NDN hash table with the LIBF approach. The number of hash probes affects LIBF false positive rate. we can clearly see from the Figure 9 that LIBF reduces the number of hash probes. In the ideal condition if the false positive rate is 0 then the number of hash probes would be constantly 1. Reducing the number of hash probes results in increase of lookup speed and improves the performance of longest prefix match.

![Fig. 9. Number of hash probes](image)

VI. CONCLUSION

In this paper, we proposed a design that enables faster longest prefix match in NDN forwarding at the expense of higher memory consumption. We compared our design with the hash table and single bloom filter. Our approach reduces the time taken for longest prefix match compared to single bloom filter and hash table. We modify the FIB to include our data structure that provides good average lookup time. We tested our design using NDNSim. In future, we plan to run our design on advanced hardware technology such as GPU to run LPM in parallel which could enable better performance.
VII. ACKNOWLEDGMENT

I would like to show my gratitude to Dr. Minseok Kwon for his expert guidance and feedback that helped me to make improvements in this research.

REFERENCES


