

FCTree: A Space Efficient FIB Data Structure for NDN Routers

Ouassim Karrakchou, Nancy Samaan, Ahmed Karmouch
School of Electrical and Computer Engineering, University of Ottawa
Ottawa, Ontario, Canada
 {okarr102, nsamaan, akarmouc}@uottawa.ca

Abstract—Named Data Networking (NDN) is a future Internet architecture that replaces IP addresses with namespaces of contents that are searchable at the network layer. A challenging task for NDN routers is to manage forwarding-information bases (FIBs) that store next-hop routes to contents using their stored usually long names or name prefixes. In this paper, we propose FCTree, a compressed FIB data structure that significantly reduces the required storage space at the router and can efficiently meet the demands of having routes that are orders of magnitude larger than IP-based ones in conventional routing tables. FCTree employs a localized front-coding compression to buckets containing partitions of the routes. The top routes in these buckets are then organized in B-ary self-balancing trees. By adjusting the size of the buckets, the router can reach an optimal tradeoff between the latency of the longest prefix matching (LPM) operation and the FIB storage space. In addition, in contrast to existing hash and bloom-filter based solutions, the proposed FCTree structure can significantly reduce the latency required for range and wildcard searches (e.g., for latency sensitive streaming applications or network-layer search engines) where up to k routes are returned if they are prefixed by a requested name. Performance evaluation results demonstrate the significant space savings achieved by FCTree compared to traditional hash-based FIBs.

I. INTRODUCTION

The current Internet architecture is centered around the thin IP waist hourglass and its host-centric protocol suite. This architecture proved to be very efficient in hosting the first generation of client-server and peer-to-peer applications (e.g., e-mail exchanges and file sharing). Unfortunately, design trends of current and future applications (e.g., media streaming, social networking, augmented reality and scientific data sharing) are pushing towards a paradigm shift that focuses on information and content-centric communication. These applications allow the users to search and request contents without the need to reference their original producers or hosts. This location agnostic content delivery requirement is traditionally realized using application-layer solutions (e.g., domain name servers, content delivery networks and search engines). However, such a multi-step solution, where a particular host-name must be first identified, followed by an IP address resolution and finally IP-based routing, incurs an unnecessary additional network traffic. Moreover, it adds additional latencies that may not be tolerated by emerging ultra-low latency applications [1].

Named Data Networking (NDN) [2] represents a clean-slate future Internet architecture with the promise of providing an efficient host-independent content delivery service within the

network layer. To achieve this goal, NDN routers store content names to uniquely identify corresponding contents rather than IP addresses that identify hosts. NDNs are said to also democratize the Internet since any entity can create content names and, hence, publish and consume data. Clearly, in contrast to IP addresses, naming space scalability is achieved in NDNs where applications create their namespaces independently from the network. Efficient multicast with minimal bandwidth becomes inherent with the ability of routers to cache data and respond locally to multiple requests of the same data [3].

Similar to traditional routing tables, an NDN router's forwarding information base (FIB) stores content names or name prefixes and a corresponding list of next-hops that may have the contents or are on the route to the contents [8]. To serve a content request, received in an interest packet (Ipkt), the router finds the longest prefix match (LPM) of the content name on its FIB [9]. However, LPM is a complex operation in NDNs due to the hierarchical and variable length representation of content names and the expected size of the FIB. Indeed, the number of contents is expected to be several orders of magnitude larger than that of traditional IP addresses. Furthermore, it is realistic to assume that the number of contents grows faster than the number of hosts (e.g., a new video uploaded on *youtube* or an article that is published in an online newspaper issue generate a new content name). Additionally, NDN routers take advantage of content independence from hosts, and selectively store contents they route to their neighbours to speed up the response to future similar interests. There are two direct consequences of this functionality. First, FIB tables in the entire domain must be frequently updated to point to the location of the cached contents within the routers. Secondly, traditional IP route aggregation mechanisms become almost obsolete in NDNs since contents are constantly moving. These functionalities add stringent space and access-time requirements on any adopted FIB data structure that were not present in traditional IP routing tables.

Research efforts relating to FIB data structures rely broadly on a hybrid structure comprised of three well known solutions, namely, hash-tables, bloom-filters and tries (e.g., [8], [6], [7], [9]–[11]). The former achieves a constant time search for exact content name matches at the expense of lookup collisions for multiple names as the number of content names increases. Bloom-filters do not allow deletions and may produce false positives. Solutions based on hash-tables and Bloom-filters

TABLE I
A COMPARISON OF THE PROPOSED FIB DATA STRUCTURES

	Basic Hash Table [4]	Hash Table with Bloom Filters [5]	Bloom Filter only [6]	Character Trie [7]	Name Prefix Trie [7]
Advantages	Constant time lookup, insertion and deletion on average	Advantages of the Hash Table data structure	Advantages of the Hash Table data structure with a decreased memory consumption	Efficient time complexities of lookup, insertion and deletion. Longest prefix match is natively supported	Advantages of the Character Trie data structure with decreased memory utilization
Disadvantages	Degradation of time complexities due to collisions as table utilization increases. To restore the constant time complexities, a costly hash table resize needs to be performed	Disadvantages of the Hash Table data structure with a slower degradation of the time complexities as table utilization increases	Bloom filters have a false positive error rate which may cause incorrect Ipkt forwarding	Memory utilization is high due to the large number of pointers needed	The time complexity of the lookup is degraded due to the large average number of edges per node. This can be reduced using STAs, however the insertion complexity is still linear in the worst case

may not scale well to the expected large sizes of NDN FIBs. On the other hand, trie-based solutions can efficiently search for content names but, in general, require a large space to store the entire trie tree in memory. Otherwise, a large number of I/O operations is required.

To the best of the authors' knowledge, little efforts have been directed towards two critical issues; the first is the unavoidable use of data compression in order to meet the storage scalability requirements for the exponentially increasing FIB entries. This is even more important if these tables are exchanged by routers as part of their routing protocols [8], [12]. The second issue relates to supporting additional functionalities such as range and wildcard searches in order to serve emerging applications. For example, an NDN router may serve real-time traffic (e.g., media streaming applications) where a single wildcard content interest request triggers the router to return multiple contents representing the streamed traffic [13]. Similarly, NDN routers may also support network-level content browsing functionalities in order to reduce the latency of current web-based search engines [14], [15].

In this paper, we propose a new data structure, front-coded tree (FCTree), that addresses the aforementioned issues. FCTree is a novel FIB data structure that can efficiently reduce the time needed for lookup, insertion and deletion of name-based routes while significantly reducing the space required for FIB storage. It takes advantage of the high probability of having multiple route names with redundantly repeated prefixes. It then eliminates this redundancy using front coding [16]. The space savings come at the expense of a slight increase in the lookup time for the LPM of a name. To achieve a tradeoff between the space efficiency and lookup-time, FIB routes are partitioned into blocks called buckets. Front coding (FC), a lossless compression technique, is then applied locally at each bucket which contains, on average β routes. The uncompressed bucket headers are then sorted in a B-tree where lookups take $\mathcal{O}(\log_B \frac{n}{\beta})$ where n is the number

of routes in the FIB. The parameters B and β provide a means to control the tradeoff between the FIB storage space and the speed of LPM, wildcard and range searches.

The rest of this paper is organized as follows; Section II describes and analyzes some of the key FIB implementations that have been proposed in the literature. In Section III, we introduce and analyze FCTree. Section IV presents our initial theoretical and experimental evaluation of FCTree using real datasets. Finally, Section V concludes this paper and outlines planned future work.

II. RELATED WORK

There have been several approaches described in the literature to reduce the overall state stored in NDN routers. Some of these solutions focus on the reduction of the FIB size while others explore different techniques to decrease the forwarding state managed in the network.

A. FIB size reduction

Several data structure designs have been proposed in the literature to implement the NDN routers' FIB. These data structures can be broadly classified as hash-tables, Bloom-filter based lists and tries. In this section, we analyze some of the most prominent solutions in each category and highlight their advantages and limitations. A summary of the key advantages and limitations of existing approaches is also presented in Tbl.I.

Hash-tables (HTs) constitute an efficient means to store information indexed by a string. HTs have thus been used extensively in the literature as the basis for the NDN FIB data structure [4], [8]. The basic HT consists of an array containing a fixed number m of cells termed buckets. A route is inserted in the HT by first applying a hash function to the content prefix to generate a hash value which is a number in $[0, m-1]$. This number is used to select the bucket that will be used to store the list of next hops of the content prefix. When two content prefixes have the same hash value (i.e., there is a hash

collision), chaining is used to store both the next hop lists in the same bucket.

Basic HTs have several benefits, especially in term of time complexity. Applying the hash function to a content prefix is indeed a constant time operation. As a result, when no collisions occur, the time complexities of the insertion, deletion and lookup operations remain constant. However, as the number of routes stored in the HT increases, the table size m staying constant, the probability of collision grows and chaining is used more often. As the loading factor (i.e. ratio between the number of routes n and slots m) increases, the time complexity of these operations becomes linear.

The Name Tree implementation used in Named Data Networking Forwarding Daemon (NFD) [4] addresses this problem by continuously resizing the HT whenever the number of routes stored in the FIB reaches a certain predefined threshold (by default 50% of m). Clearly, lookup time remains acceptable as long as the table utilization is small and the hash function is adjusted according to the number of entries. This comes at the expense of occasionally executing a costly table resize operation that involves recalculating the hashes of all the content prefixes and moving them to their new buckets. On the other hand, α Route [8] overcomes this limitation by partitioning the routes namespace into fixed partitions, where each partition is hashed and used for LPM operations separately. On the contrary, BFAST [5] reduces collisions probabilities by using several different hash functions to allocate content names to different partitions called buckets. Within each bucket, a counting bloom-filter is used as a second layer mapping. A route is inserted by applying a number of hash functions to the content prefix and then increment the k counters indexed by these hashes. The route is inserted in the bucket corresponding to the smallest counter. The routes stored in the other buckets are moved if needed to enforce the property that a content prefix is always stored in the bucket with the smallest counter among the k buckets indicated by the hashes. While performing better than single hashing, BFAST performance still degrades significantly as the size of the FIB grows.

NameFilter [6] aims at decreasing the memory utilization of the hash table by using Bloom filters (BF) [17]. A BF is a bit field of m bits associated to k hash functions returning hashes in $[0, m-1]$. BFs are used to represent a set of strings. A string is stored in a BF by applying the k hash functions to it, and setting to 1 the k bits indicated by the k hashes. The set membership operation is executed by checking if the k bits pointed out by the k hashes of the queried string are set to one. In NameFilter, every port of the router has an associated BF that will store the set of content prefixes that can be forwarded through the port. These BFs are merged in a single structure to decrease the number of memory accesses during lookup. NameFilter succeeds in significantly reducing the memory size of the FIB. However, BF is an approximate membership function with a false error rate that increases as the number of content prefixes stored in them grows [18]. Finally, it is worth noting that Katsaros et al. provide a detailed

analysis of the limitations of BFs in NDNs [19].

Several designs of NDN FIBs that are widely used in the literature are also based on Tries. The Character Trie (CT) is a tree where each node corresponds to a content prefix and stores the associated next hop list. The root of the CT corresponds to the empty string. Each edge is labeled with a character that extends its source node to form a new prefix. CTs are very efficient at performing the longest prefix match operation (its complexity is $O(l)$ with l the size of the longest prefix). However, they suffer from a high memory utilization due to the large number of pointers needed (every character added to a prefix increases the number of pointers). For example, Name Prefix Trie (NPT) [7] overcomes this limitation by reducing the number of nodes in the CT but at the expense of the lookup time complexity. Wang et al. improved NPT by encoding the name components to reduce the memory size of the tree and implementing the trie using State Transition Arrays (STAs) [7]. STAs allow using binary search during longest prefix match which makes the time complexity of the lookup operation logarithmic on the average number of child edges per node. However, the insertion operation remains linear in the number of edges. Furthermore, while improved, the memory utilization issue remains unsolved due to the significant number of pointers needed, especially when content prefixes have a large number of components. In general an additional limitation to Trie-based FIBs is the required large number of I/Os to traverse downward paths [20].

Finally, Quan et al. suggest to address this size issue by using a hybrid data structure called Name Lookup engine with Adaptive Prefix Bloom filter (NLAPB) [10]. In NLAPB, content prefixes are split in two parts: a B-Prefix and a T-Suffix. The B-Prefixes are managed by a counting bloom-filter with an associated HT which stores a pointer to the root of a trie that will handle the corresponding T-Suffixes. This data structure shares the advantages of the Bloom filters and trie data structures while handling some of their drawbacks (false positive rate for bloom-filters and memory utilization for the trie). This is achieved thanks to the reduced number of B-prefixes (several contents may share the same B-prefix) and the reduced length of the T-suffix. However the drawbacks of these data structures are still present especially the false positive rate that causes the network resources to be wasted.

To the best of the authors' knowledge little work has been done to investigate the efficiency of compressed data structures, other than tries, in order to meet the size scalability requirements of NDN FIBs. Furthermore, with the exception of the work of Ascigil et al. that associates tags to content names [15], none of the existing approaches provide additional router functionalities such as wildcard and range based name look ups. These two limitations are the main focus of FCTree.

B. Forwarding state reduction

Several complementary efforts have also been proposed to reduce the memory utilization in NDN routers. For example, the work of Tsilopoulos et al. [21] achieves significant state reduction in routers by having Ipkts tracked every d hops

instead of every hop. Similarly, Wang et al. [22] suggest to classify Ipkts in different classes based on popularity and track only the high popularity class which effectively reduces the state stored in NDN routers. Yuan et Crowley [23] develop efficient schemes to reduce the PIT size of each router using novel implementation techniques. On the other hand, Wu et al. [24] reduce the routing table size by introducing a service oriented ICN that addresses service names instead of contents and add a network layer that can forward directly using host addresses when a service is resolved to a specific location. These approaches are complementary in nature to the approach followed in this paper which focuses on the FIB implementation, they all however seek to address the scalability issues faced during Internet-scale deployments of NDN.

III. PROPOSED FRONT-CODED TREE (FCTREE)

We consider a router's FIB that contains a set $\mathcal{S} = \{s_1, \dots, s_n\}$ of n content names at a given time instance. Each content name prefix s_i is associated with a list l_i of one or more corresponding next-hop interfaces. The name prefix s_i is a variable length string generated from an alphabet σ . In NDN, s_i is composed of name segments separated by "/" to represent a hierarchical namespace (e.g., */ca/ottawa/* and */ca/ottawa/engineering/*). However, our data structure is generic enough to include both hierarchical and flat presentations of namespaces.

With no compression, such a FIB structure will need $\sum_{i=1}^n \|s_i\|$ bytes of memory, where $\|s_i\|$ is the number of characters in s_i , to store the content names. FIBs are used by the routers to perform an LPM of a content name s that is requested by an Ipkt. A hash function is the most commonly used method for exact route lookup. It provides a probabilistic answer in $\mathcal{O}(1)$ to the location of an exact match to s [4]. However, for an LPM to s in $\mathcal{S}$, a sequence of hash-based lookups are needed, each removes one segment at a time from the end of s until an exact match is found. Our main objective is to develop a data structure that can efficiently reduce the storage space for $\mathcal{S}$ while not sacrificing the LPM lookup time for s in $\mathcal{S}$. Furthermore, we would like to allow an additional functionality where an Ipkt may request k routes prefixed by a specific name with a wildcard search such as s^* , (i.e., a wildcard s^* interest should return a list of up to k routes having s as their prefix).

In the following sections, we provide a brief description of the front coding (FC) compression technique [16]. We then show how we employ FC to store route entries in a straightforward manner before we finally describe the proposed FCTree data structure.

A. Front Coding Based FIB

To minimize the memory footprint of any string dictionary, front coding (FC), an efficient lossless compression technique [16] can be applied. FC takes advantage of the shared prefixes between consecutive strings as follows. The first string is

stored completely and then each subsequent string is stored as a pair of values $(\|p_{i-1}\|, s_i - p_{i-1})$ where p_{i-1} is the common prefix between s_i and s_{i-1} , i.e., the substring that is common between the two names starting from their first characters. Hence, $s_i - p_{i-1}$ is the suffix added to p_{i-1} to construct s_i . As a result, the complete list of strings in our FIB can be stored in a compressed form as follows,

$$FC(\mathcal{S}) = (0, s_1), (\|p_1\|, s_2 - p_1), \dots, (\|p_{n-1}\|, s_n - p_{n-1})$$

The memory efficiency of the routers' FIB using FC can be further improved by rearranging the name prefixes based on their lexicographical order as they are inserted. In this manner, we increase the probability of shared prefixes among consecutive strings and hence increase their compression ratios. The space needed to store the strings in $FC(\mathcal{S})$ is $\sum_{i=1}^n \|s_i - p_{i-1}\|$

with the empty string p_0 . If the route names contain no common prefixes then the size of $FC(\mathcal{S})$ reduces to that of $\mathcal{S}$ with no compression. A variable length non-negative integer encoding scheme such as VByte [25] is usually employed in order to compress the integer representing the length of the common prefix. This compression can also be applied to the pointers to the list l_i , $i = 1, \dots, n$ of next-hop ports.

The main limitation with the above representation is the LPM, insertion and deletion time complexities in $FC(\mathcal{S})$ which take $\mathcal{O}(n)$. For example, using the above representation, in order to lookup s_i , one needs to sequentially decompress the first $i - 1$ FIB entries by concatenating the name prefix to its predecessor. Clearly, this is not an acceptable behaviour for an Internet-scale deployment of NDN.

B. Bucket-based Front-Code FIB (BFC)

In order to reduce the lookup time when using FC, one can partition the list of lexicographically ordered routes into a number of blocks referred to as buckets. The first content name in each bucket is always stored in full and is referred to as the bucket header, the subsequent content names in a bucket are differentially encoded as in FC. The buckets are then ordered in a list according to the lexicographic order of their headers as shown in Fig.1.

The BFC structure can be constructed incrementally by limiting the number of routes in each bucket to a maximum threshold value β . The first route in the FIB is inserted in a new bucket. The content prefix of the second route is compared to the header of this first bucket (i.e., the name prefix of the first route). If the second content prefix is lexicographically larger than the first name prefix, it is added to the first bucket using simple FC. Otherwise, a new bucket is created in the list just before the first one in order to store the second route. Consequently, each new data entry is added to the bucket that precedes another one whose header is lexicographically larger than the route name. We proceed similarly for the next routes where routes are inserted in their appropriate bucket unless this bucket contains β entries. When a route is inserted in a full bucket, we move the last route of this bucket to a newly created

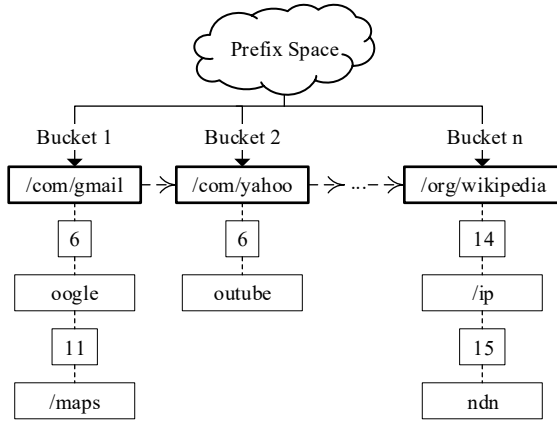


Fig. 1. BFC used to represent a FIB with n buckets. The header of the first bucket stores the content name $s_1 = "/com/gmail"$ while the second entry in the bucket stores $s_2 = "/com/google"$. Since s_2 shares the prefix $"/com/g"$ with size 6 with s_1 , it is stored as (6, "oogle"). Similarly, $s_3 = "/com/google/maps"$ is stored as (11, "/maps").

bucket. The resulting data structure with buckets $b_1, \dots, b_{\lceil \frac{n}{\beta} \rceil}$ can be represented as follows:

$$FC_{BFC}(\mathcal{S}) = \begin{aligned} & \langle \langle b_1 : (0, s_1), \dots, (\|p_{j-1}\|, s_j - p_{j-1}), \rangle, \\ & \langle \langle b_2 : (0, s_{j+1}), (\|p_{j+1}\|, s_{j+2} - p_{j+1}), \dots \rangle, \\ & \dots \\ & \langle \langle b_{\lceil \frac{n}{\beta} \rceil} : (0, s_k), \dots, (\|p_{n-1}\|, s_n - p_{n-1}) \rangle \rangle \end{aligned}$$

In order to delete or look up a route, we first perform a linear search on the bucket headers to determine the bucket where the content prefix may be located. A linear search is then performed inside the bucket to retrieve the content name. When a route entry is to be deleted, it is removed from its bucket. The bucket is then deleted if it contains no more routes. For a FIB of size n , finding a route takes an average of $\frac{n}{2\beta}$ bucket header comparisons and $\frac{\beta}{2}$ bucket decoding operations. BFC might provide an acceptable performance for routers with commodity multi-core CPUs that can perform simultaneous lookups, one at each bucket [26].

Another means to reduce the linear time complexity of BFC is to employ an order preserving perfect hash function [27] to the bucket headers. This function provides an $\mathcal{O}(1)$ access to the appropriate search bucket and uses $\mathcal{O}(\frac{n}{\beta})$ additional space. This implementation reduces the overall access-time complexity of BFC to $\mathcal{O}(1 + \beta)$. However, we did not adopt this structure, since it does not efficiently support wildcard and range lookups, and because it requires an a-priori knowledge of the content name space.

C. Front coded tree based FIB (FCTree)

The BFC data structure improves the lookup time compared to the plain FC structure while providing means to control

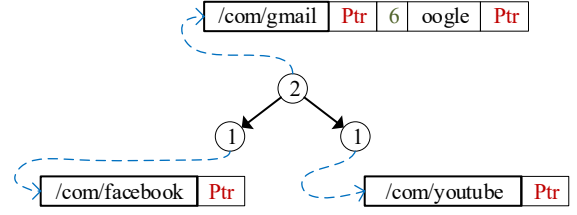


Fig. 2. FCTree used to store routes for $/com/facebook$, $/com/gmail$, $/com/google$, $/com/youtube$ with a maximum bucket size of $\beta = 2$ and Ptr stores the next-hop list.

the space compression using the β parameter. Our proposed FCTree data structure provides a further improvement to the LPM time compared to BFC. The efficiency of FCTree stems from its ability to maintain the reduced space of BFC while further reducing the linear-time search needed on the bucket headers.

In FCTree, we structure the buckets using a self-balancing binary search tree. Fig.2 shows an example of our implemented red-black tree (RBT) for our experimental results [28]. Each bucket is represented by a node in the RBT whose key is the bucket header. Let T be the FCTree representing the FIB routes, then for any node v in T , the left (respectively, right) subtree of v contains only nodes whose keys are lexicographically smaller than v (respectively, larger than v).

To guarantee that the tree is always balanced, the RBT enforces three constraints by performing tree rotations during the insertion and deletion of nodes. Each node in T is marked as either red or black, such that the root of T is black and the parent of any red node in T is also black. Finally, for any node n , every direct path from n to any of its leaf children must contain the same number of black nodes. These properties guarantee that the longest path from the root to a leaf is at most twice the size of the shortest path from the root to a leaf. This ensures that the height of the RBT is $\mathcal{O}(\log_2 \frac{n}{\beta})$ [28].

In general, we can employ a B -arity self-balancing tree, where B is the maximum number of children per node in T and $\frac{n}{\beta}$ is the number of buckets in the tree. Hence, the lookup, insertion and deletion operations are all performed in $\mathcal{O}(\beta + \log_B \frac{n}{\beta})$ time.

D. Wildcard searches in FCTree

In this section, we are interested in adding an additional functionality to the router where the FIB can be queried to return k routes to a content name with a wildcard search s^* (e.g., $/a/b/mymovie^*$), where the k returned routes, if any, must have s as their prefix (e.g., $/a/b/mymoviepart1$ and $/a/b/mymoviepart2$). In addition, a range matching request given two name prefixes s_i and s_j where s_i is lexicographically smaller than s_j should be allowed. We envision that these functionalities will be essential in future NDN routers. They can serve different types of applications (e.g., real-time streaming [13], network-level search engines [14] and Internet of things applications [15]). They can also resolve problems

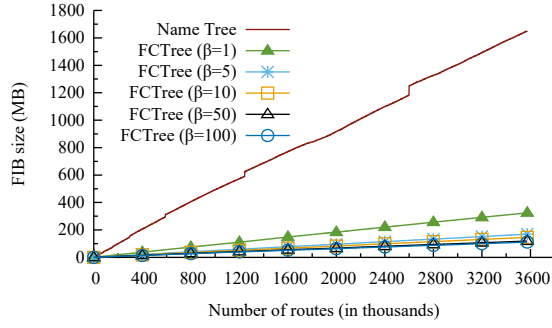


Fig. 3. Used storage space for different FIB sizes.

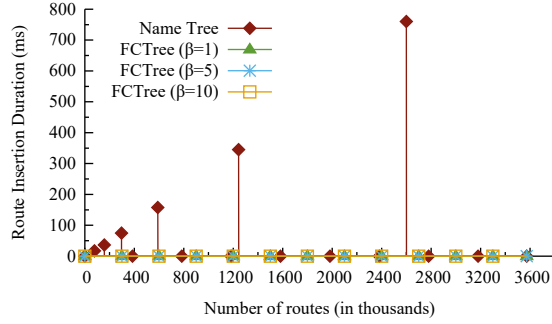


Fig. 4. Insertion time for different FIB sizes.

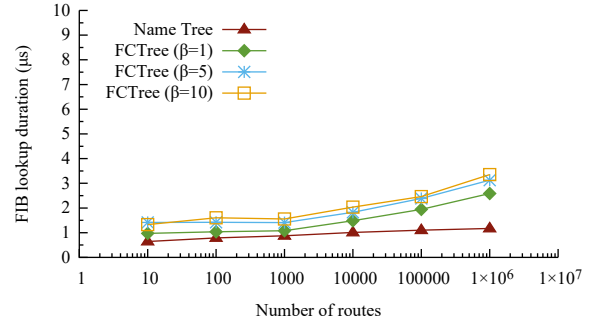
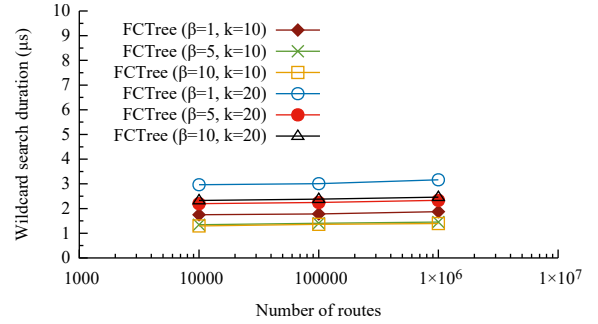


Fig. 5. LPM time for different FIB sizes.

Fig. 6. FCTree performance for wildcard searches with k routes returned.

related to content partitions and visioning [29] in addition to increasing the efficiency of existing routing protocols [30]. These functionalities cannot be easily implemented directly in HT and BF based FIBs since their mapping functions do not necessarily preserve the routes lexicographical ordering. On the other hand, these two functionalities can be implemented in FCTree as follows.

For wildcard searches, the appropriate bucket for s is first located and a sequence of decoding operations are performed. The process stops when the k routes are found or whenever a new route is found that is not prefixed by s . Consecutive buckets may be visited to find the needed k routes. The range operation proceeds in the same manner, however, the search starts with the LPM for s_1 and all the following routes are returned as long as they are lexicographically smaller than s_j .

Since our main focus is on the FIB structure, we leave the details of how these retrieved routes should be used by the routing protocols and forwarding strategies. One solution to consider is as follows; if these requests are part of a routing protocol execution, the results are returned as the content component of a data packet that is sent back to the querying router. However, if these queries are generated by an application, a simple approach is to forward the interest packet sent by the application towards each of the returned routes.

IV. PERFORMANCE EVALUATION

In this section, we evaluate the performance of FCTree and compare it to the Name Tree data structure used in NFD [4]. Name Tree employs a hash table that stores nodes which are

also part of an associated tree structure. Each node contains a content name and its associated FIB and Pending Interest Table (PIT) entries. The root of the tree structure is the node corresponding to the '/' name. The tree structure links together the content prefixes in a hierarchical manner (e.g., $/a/b/c$ is stored in the tree as $'/' \rightarrow '/a' \rightarrow '/a/b' \rightarrow '/a/b/c'$). During the forwarding of an Ipkt, the node that contains the PIT entry is first located by performing a lookup operation in the hash table and created if it doesn't exist, then the FIB entry of the longest content prefix is found by going up the tree structure. To provide a fair comparison to our proposed FCTree, we eliminated all additional operations (e.g., PIT insertions and lookups) and separated the FIB structure in the Name Tree from its PIT. We will also compare theoretically the performance of FCTree to the Character Trie (CT) FIB.

We implemented FCTree as an alternative FIB implementation for NFD, and we used ndnSIM [31] to execute our test scenarios. We implemented FCTree with an RBT (we thus have $B = 2$). We utilized the domain name information from DMOZ¹ to construct a realistic content prefix dataset that we used in all our experiments. All the experiments have been conducted on a server running Ubuntu 16.04 LTS with an Intel Xeon E31220 (3.1 GHz) and 32 GB of RAM.

A. FIB storage size

In the first experiment we measure the storage size of the FIB as we increase the number of routes. The storage

¹"DMOZ," <https://dmoztools.net/>, accessed on Jan, 18th, 2018.

TABLE II
SPACE AND TIME COMPARISON OF THE PROPOSED SOLUTIONS AND THE CHARACTER TRIE

	FC	BFC	FCTree	Character Trie
Space	$S_{FC} = M + n \times \log(\text{avg}(\ p_i\))$	$S_{BFC} = S_{FC} + (\text{avg}(\ p_i\) + c) \times \frac{n}{\beta}$	$S_{FCTree} = S_{BFC} + B \times c \times \frac{n}{\beta}$	$S_{CT} = M + M \times c$
Time	$\mathcal{O}(n)$	$\mathcal{O}(\frac{n}{\beta} + \beta)$	$\mathcal{O}(\log_B(\frac{n}{\beta}) + \beta)$	$\mathcal{O}(1)$

size measured is the full memory utilization of the FIB data structure (i.e. including the pointers' sizes). We start with an empty FIB and insert additional routes until all the DMOZ data set is stored. Fig.3 plots the size of the two FIB structures against the number of stored routes for the Name Tree and for FCTree using five different maximum bucket sizes β . It is worth noting also that Name Tree resizes its hash table whenever the number of stored routes reaches 50% of the current table size. This behavior explains the constant increase when the number of routes reaches 600, 1200 and 2400 thousand routes.

As shown in the figure, FCTree consumes a much smaller storage space and the difference between the size of FCTree and that of Name tree increases significantly as the number of routes increases. In fact, at approximately 3.6 million routes, FCTree with $\beta = 10$ is stored in approximately 100 MB while Name Tree uses 1.65 GB. This significant difference can be attributed to two factors; the first is the compression used by FCTree. The second is due to the additional tree structure and pointers maintained by Name Tree in order to speed up the FIB LPM operation.

It is worth noting that the size of the original file that contained the DMOZ name-based route strings is approximately 92.5 MB. Clearly, FCTree with $\beta = 10$ buckets has the same storage space. This comparison shows the significant space savings by FCTree, given that it also stores all the next-hop lists in addition to the tree structure memory usage.

B. Insertion speed

In this experiment, we measure route insertion speeds for both of the FIB data structures. Similar to the first experiment, we start with an empty FIB and gradually insert additional routes while measuring the time taken to perform each route insertion operation. The experiment is repeated and average values for several runs are obtained. Fig. 4 shows that both structures have an insertion speed of the same order of magnitude (less than 10 μ s) at all time points except when Name Tree performs an expensive hash table resize operation to increase the size of the table.

C. LPM lookup cost

In this experiment, we create a small network consisting of a consumer, a producer, and an NDN router connecting them using ndnSIM [31]. The router's FIB is filled with a variable number of routes. The consumer then sends IpKts for a random selection of contents distributed according to a Zipf-Mandelbrot distribution (parameters $q = 0.7$ and $s = 0.7$ as set by the simulator) [32]. We then measure the time taken by the LPM operation during the router's IpKt forwarding

process. The experiment is repeated for 10000 IpKts to obtain the average value for each point in Fig.5.

We can see that while FCTrees are slightly slower than Name Tree, their performance is still of the same order of magnitude despite the fact that FCTree has a logarithmic time LPM operation while Name Tree has a constant time LPM operation.

D. Wildcard searches

In this experiment, we populate FCTree with the DMOZ routes and then perform wildcard searches that returns $k = 10$ and $k = 20$ routes. We then measure the time taken to perform these searches. The experiment is repeated and averaged over 1000 runs. As shown in Fig.6, in general, all searches remain between 1 to 3 μ s. The worst performance is when $\beta = 1$ and the router is asked to return up to 20 routes since the search time is dominated by the binary tree traversal. On the other hand, when $\beta = 10$ and the wildcard request was set to return 10 routes, the lookup operation was very efficient. For all the plots in Fig.6, we can see that the increase in the lookup time remains logarithmic with respect to the number of buckets in the tree i.e., the ratio between the number of routes and the bucket size $\frac{n}{\beta}$.

E. Theoretical comparison with the CT FIB

Table II provides the space and time complexities of our proposed solutions and the CT FIB ($M = \sum_{i=1}^n \|s_i - p_{i-1}\|$ is the size of stored common suffix lengths, $\text{avg}(\|p_i\|)$ is the average prefix length and c is the size in bytes of a pointer to a memory address).

In terms of space utilization, FC consumes M bytes to store the front coded strings and $n \times \log(\text{avg}(\|p_i\|))$ bytes to store the integers representing the size of the common prefixes. As these integers are VByte encoded, we can assume that for realistic content prefixes $\log(\text{avg}(\|p_i\|))$ is small and close to 1. As a result, S_{FC} can be accurately approximated by $M + n$. On the other hand, the CT FIB achieves the same compression ratio as FC on the strings as prefixes are shared and stored only once. CT needs then M bytes for the string storage and $M \times c$ bytes for the pointers storage (one pointer per character stored in an optimal implementation). However, M is several orders of magnitude higher than n ($M = n \times \text{avg}(\|s\|)$) and thus $S_{CT} \gg S_{FC}$. BFC and FCTree introduce the β parameter that gives the ability to adjust the compression ratio: $\beta = 1$ holds no compression while letting β tend to n reduces FCTree to FC.

Our experimental results in Fig.3 show that the compression gain increases rapidly for small values of β but then slows down as β is increased. This proves that on a realistic data

set, $avg(\|p_i\|) \times \frac{n}{\beta}$ can be neglected for values of β as small as 5.

Our solutions are gradually improved from linear time to logarithmic time, which, nevertheless, is less efficient than the constant time complexity of the CT. However, CT achieves this performance at the expense of a large space overhead and I/O requirements.

V. CONCLUSION AND FUTURE WORK

In this paper, we designed and analyzed FCTree, a novel forwarding information base (FIB) data structure for NDN routers. The proposed FCTree is a space-efficient data structure that relies on local compressions of lexicographically ordered name-based routes. These routes are stored as partitions, or buckets in a self-balancing searchable tree structure. The significant space savings, when compared to traditional hash and bloom filter based solutions, come at the expense of a small increase in the lookup time. More precisely, FCTree has a longest prefix matching (LPM) time complexity that is logarithmic in the number of partitions, or buckets, used in the tree but efficiently meets the scalability requirement of an Internet-scale deployment of NDN FIBs. We have also shown that new functionalities such as wildcard and range searches for content names can be efficiently supported. As a future work, we plan to further improve FCTree through operation parallelization. We will also investigate the performance of FCTree using significantly larger datasets and in distributed domains.

REFERENCES

- [1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Emerging Networking Experiments and Technologies*. ACM, 2009, pp. 1–12.
- [2] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A survey of information-centric networking research," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 2, pp. 1024–1049, 2014.
- [3] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.
- [4] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, and S. DiBenedetto, "NFD developer's guide," *Technical report, NDN-0021, NDN*, 2014.
- [5] H. Dai, J. Lu, Y. Wang, T. Pan, and B. Liu, "BFAST: High-Speed and Memory-Efficient Approach for NDN Forwarding Engine," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1235–1248, Apr. 2017.
- [6] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, "Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters," in *INFOCOM*. IEEE, 2013, pp. 95–99.
- [7] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen, "Scalable name lookup in NDN using effective name component encoding," in *Distributed Computing Systems (ICDCS)*. IEEE, 2012, pp. 688–697.
- [8] R. Ahmed, F. Bari, S. R. Chowdhury, G. Rabbani, R. Boutaba, and B. Mathieu, "Alpha Route: Routing on Names," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 3070–3083, 2016.
- [9] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: Fast and DoS-resistant forwarding with hash tables," in *Architectures for Networking and Communications Systems*. IEEE, 2013, pp. 215–225.
- [10] W. Quan, C. Xu, J. Guan, H. Zhang, and L. A. Grieco, "Scalable name lookup with adaptive prefix bloom filter for named data networking," *IEEE Communications Letters*, vol. 18, no. 1, pp. 102–105, 2014.
- [11] M. Liu, T. Song, Y. Yang, and B. Zhang, "A unified data structure of name lookup for ndn data plane," in *Information-Centric Networking*. ACM, 2017, pp. 188–189.
- [12] C. Muñoz, L. Wang, E. Solana, and J. Crowcroft, "I (FIB) F: Iterated bloom filters for routing in named data networks," in *Networked Systems (NetSys)*. IEEE, 2017, pp. 1–8.
- [13] K. Matsuzono, H. Asaeda, and T. Turletti, "Low latency low loss streaming using in-network coding and caching," in *INFOCOM*. IEEE, 2017, pp. 1–9.
- [14] X. Qiao, G. Nan, Y. Peng, L. Guo, J. Chen, Y. Sun, and J. Chen, "NDNBrower: An extended web browser for named data networking," *Journal of Network and Computer Applications*, vol. 50, pp. 134–147, Apr. 2015.
- [15] M. Ascigil, S. Reñé, G. Xylomenos, I. Psaras, and G. Pavlou, "A keyword-based ICN-IoT platform," in *Information-Centric Networking*. ACM, 2017, pp. 22–28.
- [16] I. H. Witten, I. H. Witten, A. Moffat, T. C. Bell, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.
- [17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
- [18] K. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a bloom filter," *Information Processing Letters*, vol. 110, no. 21, pp. 944–949, 2010.
- [19] K. V. Katsaros, W. K. Chai, and G. Pavlou, "Bloom filter based inter-domain name resolution: A feasibility study," in *Information-Centric Networking*. ACM, 2015, pp. 39–48.
- [20] C. Dillabaugh, M. He, and A. Maheshwari, "Succinct and I/O efficient data structures for traversal in trees," *Algorithmica*, vol. 63, no. 1-2, pp. 201–223, 2012.
- [21] C. Tsilopoulos, G. Xylomenos, and Y. Thomas, "Reducing forwarding state in content-centric networks with semi-stateless forwarding," in *INFOCOM*. IEEE, 2014, pp. 2067–2075.
- [22] X. Wang, W. Wang, C. Zeng, R. Dai, S. Wang, and S. Xu, "Reducing the size of pending interest table for content-centric networks with hybrid forwarding," in *International Conference on Communications (ICC)*. IEEE, May 2016, pp. 1–6.
- [23] H. Yuan and P. Crowley, "Scalable pending interest table design: From principles to practice," in *INFOCOM*. IEEE, 2014, pp. 2049–2057.
- [24] Q. Wu, Z. Li, J. Zhou, H. Jiang, Z. Hu, Y. Liu, and G. Xie, "SOFIA: Toward service-oriented information centric networking," *IEEE Network*, vol. 28, no. 3, pp. 12–18, 2014.
- [25] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *The Computer Journal*, vol. 42, no. 3, pp. 193–201, 1999.
- [26] K. Suksomboon, N. Matsumoto, S. Okamoto, M. Hayashi, and Y. Ji, "Configuring a Software Router by the Erlang-k -Based Packet Latency Prediction," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 422–437, Mar. 2018.
- [27] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, "A family of perfect hashing methods," *The Computer Journal*, vol. 39, no. 6, pp. 547–554, 1996.
- [28] S. Hanke, "The Performance of Concurrent Red-Black Tree Algorithms," in *Algorithm Engineering*. Springer, Jul. 1999, pp. 286–300.
- [29] R. S. Antunes, M. B. Lehmann, R. B. Mansilha, L. P. Gaspar, and M. P. Barcellos, "NDNrel: A mechanism based on relations among objects to improve the performance of NDN," *Journal of Network and Computer Applications*, vol. 87, pp. 73–86, 2017.
- [30] E. Hemmati and J. J. Garcia-Luna-Aceves, "A Comparison of Name-Based Content Routing Protocols," in *Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2015, pp. 537–542.
- [31] S. Mastorakis, A. Afanasyev, I. Moiseenko, and L. Zhang, "ndnSIM 2: An updated NDN simulator for NS-3," *Technical Report, NDN-0028, Revision 2, NDN*, 2016.
- [32] S. Mastorakis, A. Afanasyev, and L. Zhang, "On the evolution of ndnSIM: An open-source simulator for NDN experimentation," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 19–33, 2017.