Fast Name Lookup for Named Data Networking

Yi Wang†, Boyang Xu†, Dongzhe Tai†, Jianyuan Lu†©,
Ting Zhang†, Huichen Dai†, Beichuan Zhang‡, Bin Liu†
†Tsinghua National Laboratory for Information Science and Technology,
Departement of Computer Science and Technology, Tsinghua University
‡Computer Science Department, The University of Arizona

***Abstract*—Complex name constitution plus huge-sized name routing table makes wire speed name lookup a challenging task in Named Data Networking. To overcome this challenge, we propose two techniques to significantly speed up the lookup process. First, we look up name prefixes in an order based on the distribution of prefix length in the forwarding table, which can find the longest match much faster than the linear search of current prototype CCNx. The search order can be dynamically adjusted as the forwarding table changes. Second, we propose a new near-perfect hash table data structure that combines many small sparse perfect hash tables into a larger dense one while keeping the worst-case access time of $O(1)$ and supporting fast update. Also the hash table stores the signature of a key instead of the key itself, which further improves lookup speed and reduces memory use.**

keywords*—*Named Data Networking; Name Lookup; Perfect Hash Table; Linear Search; Random Search.

I. Introduction

Named Data Networking (NDN) is a new Internet architecture that routes and forwards packets based on names rather than addresses. While it brings important benefits to application development, network efficiency and resiliency, and data-centric security, NDN also poses some technical challenges. One of such challenges is fast, scalable, and efficient forwarding lookup. In NDN, the Forwarding Information Base (FIB) stores name prefixes and corresponding next-hops. When a packet needs to be forwarded, the packet's name will be used to do a longest prefix match (LPM) lookup against the FIB to find the next-hops. Though conceptually this process is very similar to IP's forwarding lookup, its high-performance realization is very challenging:

- *Fast name lookup*. Unlike IP addresses, which have fixed-length of 32 bits or 128 bits, NDN names have variable and unbounded length, and usually contain tens or hundreds of characters. These long, variable-length names will make conventional lookup algorithms time-consuming, especially considering the ever-increasing link speed, e.g., 40 Gbps (OC-768) is already deployed and 100 Gbps Ethernet is coming up. Thus wire-speed forwarding lookup is a significant challenge in NDN.
- *Efficient memory use*. NDN's FIB is expected to be much larger than IP's because (1) in each FIB entry the NDN name takes more space than an IP address, and (2) the number of name prefixes (i.e., aggregates of contents) is larger than the number of IP prefixes (i.e., aggregates of addresses). An NDN FIB may contain tens of millions of name prefixes or even more. Therefore careful design of FIB data structure is needed in order to use memory space efficiently.
- *High update rate*. In addition to changes in network topology and routing policy, an NDN router may also need to handle updates caused by content publishing and deletion, which makes FIB update much more frequent than that of today's Internet. Thus the design should support fast update operations, including insertion and deletion, while keeping name lookup fast.

The current prototype implementation, CCNx [1], achieves LPM as follows. The FIB is a hash table keyed by name prefixes. Given a hierarchical name with a number of components, CCNx retrieves all possible prefixes from the name, conducts exact-match search of the prefixes, from the longest to the shortest, in the FIB, and stops when the first match is found. The overall forwarding performance, however, is at least an order of magnitude below wire speed [2].

NameFilter [3] is a recent work to address this problem. It is a two-stage, Bloom filter-based scheme. In the first stage it determines the prefix length of a name, and in the second stage it looks up the prefix in a group of Bloom filters. By optimizing the hash value calculation of names, as well as a careful design of the data structure for storing multiple Bloom filters, NameFilter significantly reduces the memory access time and speeds up the overall lookup process. Its performance, however, depends on the distribution of prefix length and the number of ports in the router. The prefix distribution determines the number of Bloom filters in the first stage, and the number of ports determines the number of Bloom filters in the second stage. With unfavorable prefix distribution and large number of ports, NameFilter's speed may decrease.

After analyzing existing schemes, we identified two specific technical challenges in order to accelerate LPM name lookup: (1) how to quickly find the length of the longest prefix, and (2) how to speed up hash table operations. To address the first challenge, we precompute the distribution of prefix length of all prefixes in FIB, and use the distribution to guide the search of longest match. This is different from CCNx's linear search or NameFilter's Bloom filter approach, and it turns out to be

©Corresponding author: Bin Liu, liub@tsinghua.edu.cn. This work is supported by 863 project (2013AA013502), NSFC (61373143), Tsinghua University Initiative Scientific Research Program (20121080068).

a very effective technique. To address the second challenge, we designed a new hash table data structure to reduce access time as well as memory use. More specifically, we make the following major contributions in this paper:

1) We develop an adaptive greedy strategy to search for the longest matching prefix. The search is greedy and fast since it is guided by the knowledge of prefix distribution, and adaptive since it can dynamically adjust the search path in response to any changes in prefix distribution. This method not only speeds up name lookup but also is resilient to an attack of very long names. The overhead is the storage of some additional prefixes, but the percentage of these extra entries decreases as the FIB size increases.
2) We develop a novel near-perfect hash table data structure that combines many small sparse perfect hash tables into a larger dense one, which keeps the worst-case access time of $O(1)$. This new data structure stores the signature of a key (i.e., prefix) in each entry instead of the key itself, which further improves lookup speed and reduces memory use.
3) We conduct extensive experiments on different prefix tables and traces to evaluate our algorithm. Experiments on a commodity PC server with 10M name prefixes show that our name lookup engine achieves 41.74 MSPS (Millions Searches Per Second) by using only 247.20 MB memory, which is 12.56 times of speedup and 46.73% memory saving over CCNx. Our scheme also has a 24.5% speedup against NameFilter, with only 5.5% more memory, and it avoids NameFilter's problem of degrading performance as the number of ports increases. Furthermore, our scheme is scalable to large forwarding tables, and supports insertions of 0.75M per second and deletions of 3.08M per second while keeping high speed name lookup.

The rest of this paper is organized as follows. The name lookup process in CCNx and the greedy name lookup mechanism are presented in Section II. Then, we describe the designs of the basic and improved string-oriented perfect hash tables in Section III. In Section IV, we conduct extensive experiments to evaluate the performance of the greedy name lookup mechanism. After reviewing related work in Section V, we conclude the paper in Section VI.

II. Name Lookup Mechanisms

A. Name lookup process in CCNx

CCNx [1] is an open source project in early stage development exploring the next step in networking, based on a fundamental architectural change: replacing named hosts with named content as the primary abstraction. CCNx is based upon the CCN architecture and sponsored by the Palo Alto Research Center (PARC). There are some projects and organizations working on CCNx as well, including CONNECT project [4], CCN project [5] and NDN Project [6].

The name lookup process in CCNx [7] is illustrated in Figure 1. In CCNx, the name lookup process first generates

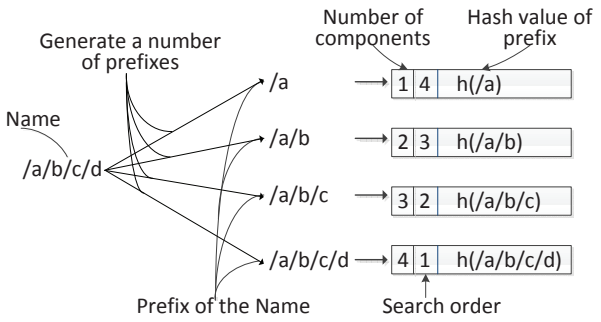

Fig. 1: The name lookup process in CCNx.

all the possible prefixes from the searched name. Then, these candidate prefixes are sorted in a descending order according to the component number of prefixes. Finally, the program looks up the candidate prefixes one by one until matching or failure. For example, in Figure 1, the name */a/b/c/d* generates 4 prefixes: */a/b/c/d*, */a/b/c*, */a/b* and */a*. Then the longest prefix */a/b/c/d* is first searched in the FIB. If there is an entry in the FIB matching */a/b/c/d*, the name lookup process terminates and returns the corresponding action in that entry. Otherwise, the next shorter prefix is looked up in FIB one by one and the procedure will stop until matching or after exhausting all the prefixes.

B. Greedy name lookup mechanism

Name lookup in NDN is a search process to find the longest prefix corresponding to a given key, i.e. the name. As described in Section II-A, the name lookup process in CCNx searches the longest prefix by enumerating all the possible prefixes of a name and looking up the candidate prefixes one by one in a descending order according to the component number of prefixes until matching or failure. Therefore, the name lookup process in CCNx usually takes several hash table searches before finding the right longest prefix. In our experimental setup on prefixes tables and traces (See Section IV-A), CCNx takes 4.81 searches to find the longest prefix in average.

In fact, we can speed up the name lookup mechanism by improving the search process. Table I demonstrates the prefixes distributions of 3M and 10M prefix table (Also see Section IV-A) based on the component number of prefixes, respectively. Here, 2,129,835 prefixes consist of 2 components and 391,377 prefixes consist of 3 components, which are 83.70% and 15.38% of 3M prefix table, respectively. Meanwhile, 2,125,845 prefixes with 2 components are the longest prefixes (leaf prefixes[1]), sharing 83.54% of the entire table.. Therefore, if we first search the prefix with 2 components of the name, the probability of returning the longest prefix is 83.54%. The distributions of 3M and 10M prefix table enlighten us to optimize the name lookup process by rearranging the search order of candidate prefixes according to the prefix distribution in a prefix table.

In order to explain the greedy name lookup mechanism clearly, we present the greedy name lookup process in the form of state transition diagram illustrated in Figure 2. There

TABLE I: The distribution of 3M and 10M prefix table.

	3M prefix table									10M prefix table								
Components #	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
Names #	224	2,129,835	391,377	21,664	1,598	76	17	2	1	1,248	7,150,441	2,106,964	264,956	27,183	1,268	288	12	2
Percentage (%)	0.01	83.70	15.38	0.85	0.06	0	0	0	0	0.01	74.85	22.06	2.77	0.28	0.01	0	0	0
Name Length (Byte)	3.41	16.47	18.88	21.95	28.66	35.05	40	36	55	6.15	16.97	18.91	22.84	24.30	33.59	30.59	36.58	40
Leaf Prefixes #	31	2,125,845	390,244	21,538	1,594	76	17	2	1	625	6,202,631	1,953,731	256,150	26,280	1,214	284	11	2

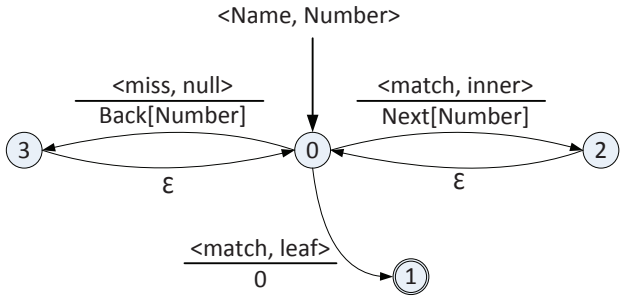

Fig. 2: The state transition diagram of name lookup process.

	1	2	3	4	5	6	7	8	9
Next:	0	3	4	5	6	7	8	9	0

	1	2	3	4	5	6	7	8	9
Back:	0	1	0	0	0	0	0	0	0

Fig. 3: The $Next$ array and $Back$ array of 3M prefix table.

are 4 states, state-0 is the start state, state-1 is the end state, state-2 and state-3 are the middle states. And the functions of the 4 states and the transitions between them are shown in the following:

1) State-0 generates the prefix with $Number$ components from the input $Name$, and searches the prefix in the hash table. If the prefix is found, state-0 transits to state-1 when the prefix is a leaf prefix, or state-0 transits to state-2 when the prefix is an inner prefix[2]. If the prefix is missed, state-0 transits to state-3.
2) State-1 stops the name lookup process and returns the result.
3) State-2 gets the next prefix's component number $Number$ from the $Next$ array, and transits to state-0.
4) State-3 gets the next prefix's component number $Number$ from the $Back$ array, and transits to state-0.

C. Algorithms for constructing $Next$ array and $Back$ array

Here, we build $Next$ array and $Back$ array according to the distribution of the prefix table in advance. $Next$ array indicates the component number of next searched prefix when the current prefix is found and the prefix is an inner prefix. Contrarily, $Back$ array indicates the next component number when the prefix with current component number is missed. Figure 3 illustrates the $Next$ array and $Back$ array of 3M prefix table. Since the prefixes with 2 components have the maximal number of leaf prefixes, the first input $Number$ in Figure 2 is 2. And $Next[2]$ is 3 as the number of leaf prefixes with 3 components is second only to the number of leaf prefixes with 2 components. Besides, $Next[1]$ is 0 as the number of leaf prefixes with 2 components is greater than the number of leaf prefixes with 1 component, which means the prefix with 2 components is always searched ahead of the prefixes with 1 component and the process should be stopped if matched. $Back$ array has the opposite characteristic. In $Back$ array, the longer prefixes with lower priority is set to 0, because the second only shorter prefixes have been searched. The building processes of $Next$ array and $Back$ array are described in Algorithm 1.

Algorithm 1 Build $Next$ and $Back$ array

```
1: procedure BuildArrays(LeafPrefixes[1..k])
2:     Num[1..k] ← Sort(LeafPrefixes[1..k]);
3:     Next[1..k] ← (−1, ..., −1); Back[1..k] ← (−1, ..., −1);
4:     for i ← 1 to k do
5:         Next[Num[i]] ← 0; Back[Num[i]] ← 0;
6:         for j ← i + 1 to k do
7:             is_available = true;
8:             if Num[i] < Num[j] then
9:                 for m ← Num[i] + 1 to Num[j] − 1 do
10:                    if Next[m] ≥ 0 then
11:                        is_available ← false;
12:                        break;
13:                if is_available = true then
14:                    Next[Num[i]] ← Num[j];
15:            else
16:                for m ← Num[j] + 1 to Num[i] − 1 do
17:                    if Next[m] ≥ 0 then
18:                        is_available ← false;
19:                        break;
20:                if is_available = true then
21:                    Back[Num[i]] ← Num[j];
```

The $Next$ array and $Back$ array can be dynamically adjusted to the optimal search path of name lookup according to the changes of the prefix table. Meanwhile, the $Next$ array and $Back$ array have the same length, which equals the maximal prefix's length in the FIB. The limit lengths of $Next$ array and $Back$ array can prevent the attack on the scheme of providing names with one hundred components, or even more.

In order to explain the building processes of $Next$ array and $Back$ array more clearly, Figure 4 illustrates an example. Suppose the Num array {3,1,6,2,4,5} is sorted in a descending order according to the number of leaf prefixes which share the same number of components. The

[1]A leaf prefix is the longest prefix for which the name lookup process searches. In this scene, the search process stops and returns the result when finding a leaf prefix.

[2]A name prefix is an inner prefix, if and only if it is the prefix of another name prefix.

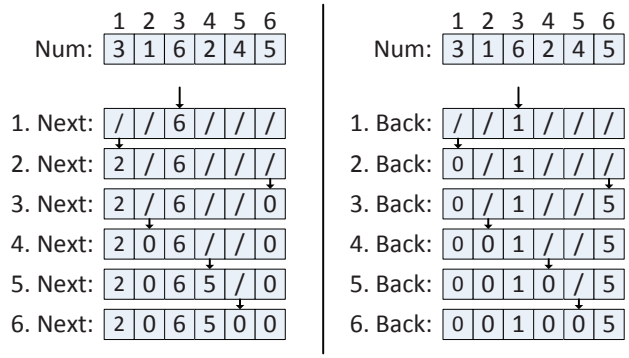

Fig. 4: The building process of $Next$ array and $Back$ array.

building process of $Next$ array has 6 iterations: First, the building process starts from $Num[1]$=3, and finds that the $Num[3]$=6 is greater than $Num[1]$. At the same time, $Next[4]$ and $Next[5]$ are not used, so $Num[3]$ is available and we get $Next[Num[1]]$=$Next[3]$=6; Then the process moves to $Num[2]$=1, and finds that $Num[4]$=2 is available, so we get $Next[Num[2]]$=$Next[1]$=2; The rest 4 iterations are similar to the first and the second iterations which have been shown in Algorithm 1 from line 6 to line 17. The building process of $Back$ array also has 6 iterations and each iteration has been described in Algorithm 1 from line 18 to 28. Finally, the process generates the $Next$ array and $Back$ array to guide the name lookup operation.

D. FIB reconstruction

Greedy name lookup process employs $Next$ array and $Back$ array to optimize the search path. However, the original CCNx FIB structure does not support $Next$ array and $Back$ array. For instance, supposing that prefix "/a/b/c" is the longest prefix of name "/a/b/c/d", the input $number$ is 2, and the $Next$ array and $Back$ array are $\{0,3,4,0\}$ and $\{0,1,0,0\}$, respectively. If the prefix "/a/b" is not found in the hash table, the process will move to the prefix "/a" and finally returns a failure, which would cause a mistake. In order to support greedy name lookup mechanism, therefore, we need to add the prefix "/a/b" and "/a" to the FIB as inner prefixes when the prefix "/a/b/c" is inserted into the prefix table.

The experimental results (illustrated in Section IV-B3) demonstrate that a reconstructed FIB on 10M prefix table needs to store 14.4% additional prefixes, and the percentage of extra entries decreases while the FIB size increases.

III. String-oriented Perfect/Near-Perfect Hash Table Design

CCNx applies hash table to construct the FIB, hence a good design of hash table would effectively improve the name lookup performance. Chaining data structure is used to solve the hash collisions, therefore the time complexity of hash table search in CCNx is $O(1+\alpha/2)$ [8]. On the other hand, the search time complexity of Perfect Hash Table (PHT) is $O(1)$ [9], [10]. For boosting the hash table search speed, we first present a basic string-oriented perfect hash table in Subsection III-A, which achieves near minimal perfect hash table and reduces the search time complexity from $O(1+\alpha/2)$ to $O(1)$. Then, in Subsection III-B, we continue to improve the basic string-oriented prefect hash table for further speeding up hash table search and reducing memory consumption. The incremental update mechanism for both the basic and the improved string-oriented near-perfect hash table is introduced in Subsection III-C. Finally, Subsection III-D and Subsection III-E analyze the perfect hash table search speed and the false positive introduced by replacing the string key with its signature, respectively.

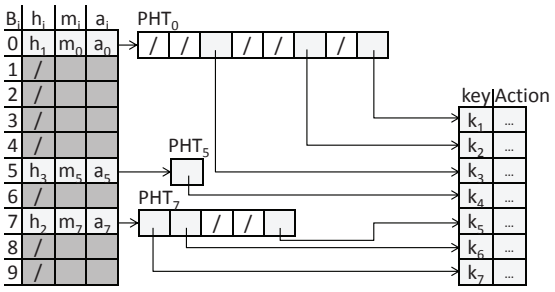

Fig. 5: Basic string-oriented perfect hash table.

A. Basic string-oriented perfect hash table

First of all, we give the definitions of perfect hash function and perfect hash table. In all situations, a "universe" U of possible keys is given, and a set $S \subseteq U$ of size $n = |S|$ of relevant keys is given as input. The range is $[m] = 0, 1, ..., m-1$. The definition of perfect hash function, minimal perfect hash function, perfect hash table and minimal perfect hash table are described in Definition 1, 2, 3 and 4, respectively.

Definition 1: Perfect Hash Function (PHF): Suppose that S is a subset of size n of the universe U. A function $h(\cdot)$: $S \in [m]$ is called a perfect hash function for $S \subseteq U$, when restricted to S, it is injective (one-to-one) [11], [12].

Definition 2: Minimal Perfect Hash Function (MPHF): Let $|S| = n$. A perfect hash function $h(S)$ is minimal if $h(S)$ equals $0, 1, ..., n-1$ $(n = m)$ [11], [12].

Definition 3: Perfect Hash Table (PHT): If all keys are known ahead of time, a perfect hash function can be used to create a perfect hash table with no collisions [11], [12].

Definition 4: Minimal Perfect Hash Table (MPHT): If all keys are known ahead of time, a minimal perfect hash function can be used to create a minimal perfect hash table with no collisions [11], [12].

Similar to other perfect hash table algorithms [10], [11], [12], the construction process of the Basic string-oriented Perfect Hash Table (BPHT) contains 3 steps:

1) ***Partition***. The set S containing n keys is divided into m small buckets according to their hash values. In Figure 5, 7 keys in S are assigned to 3 buckets B_0, B_5 and B_7 based on their hash values.
2) ***Displace***. A hash function $h_i(\cdot)$ and a small hash table with m_i entries are chosen for each bucket B_i. Every key k_{ij} in bucket B_i does not conflict with other keys in

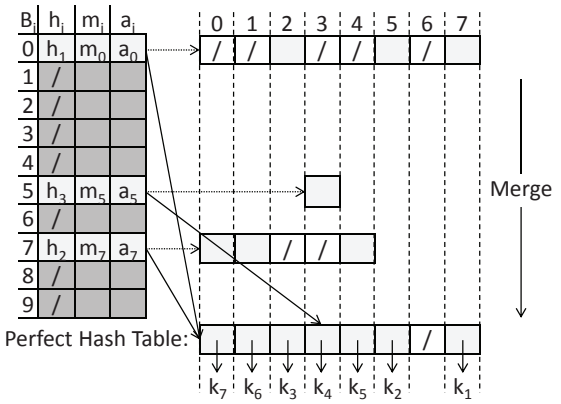

Fig. 6: Merge and compress lots of small sparse perfect hash tables into one dense perfect hash table.

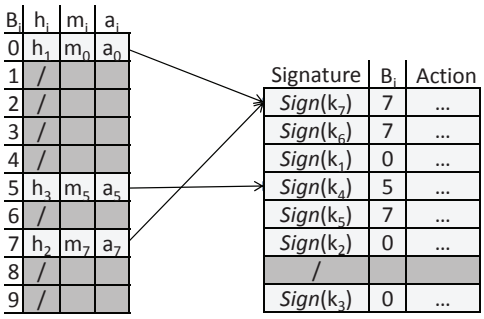

Fig. 7: Improved string-oriented perfect hash table architecture.

the same bucket B_i. In other words, there are $k_{ip} \in B_i$, $k_{iq} \in B_i$ and $k_{ip} \neq k_{iq}$, which can derive the result that $(h_i(k_{ip}) \mod m_i) \neq (h_i(k_{iq}) \mod m_i)$. Finally, a small perfect hash table for the keys in bucket B_i is built. In Figure 5, the keys in buckets B_0, B_5 and B_7 are inserted into three small perfect hash tables PHT_0, PHT_5 and PHT_7, respectively.

3) ***Merge and compress***. After step 2, there are a lot of small sparse perfect hash tables PHT_i. Typically, m_i is set to $|B_i|^2 = n_i^2$, which means the load factor of perfect hash table PHT_i is $\frac{n_i}{m_i} = \frac{1}{n_i}$. So the small perfect hash tables of B_i are sparse and have large potential space for memory saving. Meanwhile, the small sparse perfect hash tables should be merged and compressed to reduce memory consumption. Tarjan and Yao [13] proposed a merge strategy to compress the storage of two-dimension state transition tables. Inspired by this merging strategy, we combine many small sparse perfect hash tables into a larger dense one while keeping the performance of $O(1)$ worst case access time. After merging, a_i is used to store the basic offset of PHT_i in the total prefect hash table, m_i is still used to represent the length of PHT_i, and the entry address of k_{ip} can be calculated by $a_i + (h_i(k_{ip}) \mod m_i)$. For instance, PHT_0, PHT_5 and PHT_7 are merged into one larger perfect hash table in Figure 6. $PHT_0[2]$, $PHT_0[5]$ and $PHT_0[7]$ are occupied and the remaining 5 entries are available in PHT_0, while in PHT_7, $PHT_7[0]$, $PHT_7[1]$ and $PHT_7[4]$ are occupied, so PHT_0 and PHT_7 are merged where a_0 and a_7 are both set to 0. Because $PHT_5[0]$ in PHT_5 is in conflict with PHT_7, a_5 is set to 3 to avoid conflict.

The load factor α in the BPHT described above can achieve 0.99, or even larger [11], [13], [14], so BPHT is a near minimal perfect hash table.

B. Improved string-oriented near-perfect hash table

The BPHT described in Subsection III-A achieves near minimal perfect hash table, and reduces the time complexity of hash table search from $O(1+\alpha/2)$ to $O(1)$ which effectively improves the name lookup speed in CCNx. However, BPHT still has room for improvement.

1) *Memory*. For evading collisions, a hash table entry still needs to store the corresponding string key which costs more than 50% of the FIB memory in our experimental setup.

2) *Speed*. A key takes two steps to get the matched entry: calculating the bucket and searching in the perfect hash table based on the offset of the bucket. After finding the entry, we need to compare the input key with the key stored in the entry to decide whether they are matched. In this way, the overhead of both the hash calculation and the string comparison costs more than 70% of the expense spent by a hash table search.

3) *Update*. Inserting a new key into FIB will cause collisions between the new key and the existing one, so the bucket needs to choose a new hash function or build a new small hash table to keep the consistency of perfect hash table. In the worst case, the bucket should be moved to a new address of the hash table and all keys in the bucket should be re-inserted.

To reduce memory cost and improve search performance, the perfect hash table stores the signature of a key in the entry, instead of storing the key itself as shown in Figure 7. The improved string-oriented perfect hash table is much shorter than BPHT. Since a key's signature is much shorter in length than itself, the size of each entry is dramatically shrunk. At the same time, compared to the variable-length of a key, a signature is fixed-length, hence the hash table search process simply compares the signatures instead of the keys and this will speed up the search process significantly.

Furthermore, we reconstruct the hash function $h(\cdot)$ to calculate the hash value and signature of a key in one time string scanning. In other words, we redesign the hash function $< B_i > \leftarrow h(key)$ as $< B_i, Sign(key) > \leftarrow h(key)$. And the final address of key k_{ip} in the hash table is calculated by $a_i + (Sign(k_{ip}) \mod m_i)$. As a result, the hash table search process scans the key one time in average, saving 66.7% operations compared to the original 3 times key scanning. Given the update performance of the perfect hash table is related to the search process, the search performance improvement by leveraging signature also benefits the fast incremental update.

C. Incremental update

As presented in Figure 6, a lot of small sparse perfect hash tables are merged into one large dense hash table for saving memory. However, the bucket information of an entry

is lost during the merging process, which causes the perfect hash table no longer supports incremental update. To keep the incremental update ability, a new field is added to each entry of the perfect hash table, which is used to record the bucket information. Because the keys of bucket B_i are stored in the entries from a_i to $a_i + m_i$, we can scan these m_i entries and compare the bucket information B_i to filter out the keys. Therefore, the BPHT recovers the ability of incremental update. However, the improved string-oriented perfect hash table still fails to support incremental update, all signatures should be recalculated from the original keys when the signatures of the keys have collisions. For instance, the keys of bucket B_0 are stored in the perfect hash table from $a_0 = 0$ to $a_0 + m_0 = 7$ in Figure 7. And the keys belonging to bucket B_0 can be distinguished from the value of B_i in the entry. Here, 3 keys of bucket B_0 are stored in locations 2, 5, 7. When a new key k_8 with $Sign(k_0) = Sign(k_8)$ is inserted into bucket B_0, the signatures of keys in bucket B_0 should be recalculated to resolve the conflict. Unluckily, there is no original key stored in the entries. Therefore, the improved string-oriented perfect hash table cannot support incremental update.

However, similar to IP routers, NDN routers also have control plane and data plane. And the routing table, used to generate the forwarding table, is stored and maintained in control plane. Therefore, we apply BPHT to construct the routing table and leveraging the improved string-oriented perfect hash table to construct the forwarding table. As a result, the original keys of a bucket can be found in the routing table. Finally, with the help of the routing table, the forwarding table completely supports incremental update.

D. Search performance analysis for the improved string-oriented near-perfect hash table

The set U of string keys is infinite, while the signature is fixed-length and finite. Hence, it is possible that two or more keys in the same bucket B_i have the same signature. In this case, bucket B_i needs to find a new hash function $h_i(\cdot)$ which guarantees all the keys in B_i have different signatures. As a result, the perfect hash table search process takes more time to recalculate the signature based on the hash function $h_i(\cdot)$ when there are collisions with default signatures in bucket B_i. Fortunately, the probability of recalculating the signature is very low, which can be calculated by the following formulas.

Suppose that the outputs of hash function $h(\cdot)$ are perfectly random. Then, the number of keys in bucket B_i obeys Poisson Distribution when the number of keys in set S is large. Therefore, the probability of a bucket with k keys can be calculated by Formula 1.

$$P(k) = \frac{e^{-\frac{n}{m}} * (\frac{n}{m})^k}{k!} \tag{1}$$

Here, n is the total number in set S, and m is the number of buckets. A bucket contains k keys, the probability of that k keys have different k signatures can be calculated by Formula 2.

$$P_s(k) = P_s(k|k-1) * P_s(k-1)$$

$$P_s(k) = \prod_{i=2}^{k} P_s(i|i-1) * P_s(1) = \prod_{i=2}^{k} P_s(i|i-1) \tag{2}$$

Since, $P_s(k|k-1) = 1 - \frac{k-1}{N}$. Here, N is the number of different signatures. We can derive the Formula 3 to compute the probability of collisions among the signatures of the k keys in the same bucket.

$$P_c(k) = 1 - \prod_{i=1}^{k}(1 - \frac{i-1}{N}) \tag{3}$$

Derived from Formula 1 and Formula 3, the average probability of collision can be calculated by Formula 4.

$$\overline{P_c} = \sum_{k=1}^{n}(P_c(k) * P(k)) = \sum_{k=1}^{n}(1 - \prod_{i=1}^{k}(1 - \frac{i-1}{N})) * P(k)$$

$$\overline{P_c} = \sum_{k=1}^{n}(1 - \prod_{i=1}^{k}(1 - \frac{i-1}{N})) * \frac{e^{-\frac{n}{m}} * (\frac{n}{m})^k}{k!} \tag{4}$$

In our experimental setup, a signature has 4 bytes which means $N = 2^{32}$, and $m = n/4$. So the average probability of recalculating a key's signature is about $1.8 * 10^{-9}$, which means the search in improved string-oriented perfect hash table scans $1 + 1.8 * 10^{-9}$ times of the key in average. Compared with the search in BPHT, the improved string-oriented perfect hash table speeds up 2 times.

E. The false positive

The improved string-oriented perfect hash table will not cause false negative, since all the keys in the same buckets have a unique signature to be identified. But it will cause false positive in the following scenario: a searched key k_e does not belong to the keys set S, but has $< B_i, Sign(k_e) > \leftarrow h(k_e)$. By coincidence, $Sign(k_e)$ is equal to a signature $Sign(k_{ip})$ of key k_{ip} in the bucket B_i where $k_e \neq k_{ip}$. Finally, the search process returns the entry of key k_{ip} to key k_e, which causes false positive. The false positive probability of this scenario can be calculated by Formula 5.

$$FP(k) = \frac{k}{N} \tag{5}$$

According to Formula 1, the average false positive probability can be derived to Formula 6.

$$\overline{FP} = \sum_{k=1}^{n} FP(k) * P(k) = \sum_{k=1}^{n} \frac{e^{-\frac{n}{m}} * (\frac{n}{m})^k}{k!} * \frac{k}{N}$$

$$\overline{FP} < \sum_{k=1}^{\infty} \frac{e^{-\frac{n}{m}} * (\frac{n}{m})^k}{k!} * \frac{k}{N} = \frac{n}{m} * \frac{1}{N} \tag{6}$$

Here, $\overline{FP} = 4 * \frac{1}{2^{32}} = \frac{1}{2^{30}} < 10^{-9}$ in our experimental setup[3].

[3]In NDN, Pending Interest Table can eliminate the Interest packet loop caused by false positive [3], [6]. Thus, generally, the extremely small false positive issue here can be ignored in practical applications.

IV. Experimental Evaluation

In this section, we evaluate the performance of greedy name lookup mechanism and compare it with the other name lookup methods in terms of lookup throughput, memory space, update speed as well as scalability.

We implement and compare 5 methods to expose the advantages of greedy name lookup mechanism. The performance of the method used by CCNx [1] (hash table load factor α=0.5) is the baseline, and CCNx is improved by employing the basic string-oriented perfect hash table (named CCNx-PHT). The greedy name lookup mechanism introduced in Section II further speeds up the name lookup (named Greedy-PHT). Furthermore, Greedy-SPHT combines greedy name lookup mechanism with the improved string-oriented near-perfect hash table developed in Section III-B, which significantly speeds up the name lookup. Even compared with NameFilter [3], Greedy-SPHT demonstrates better name lookup performance.

A. Experimental setup

1) Prefix Tables and Name Traces: Both prefix tables and name traces used in our experiments are downloaded from *www.namelookup.org* [15]. The two prefix tables, "3M prefix table" and "10M prefix table", contain 2,544,794 entries and 9,552,363 entries, respectively. Each prefix table entry is composed of an NDN-style name and a next-hop port number. The distribution of the number of components and the average length of prefixes are shown in Table I. The name traces simulate the destination names carried in NDN packets. There are two types of name traces, simulating average lookup workload and heavy lookup workload, respectively. Each name trace contains 200M names.

2) Computational platform: The name lookup engine is implemented and run on a commodity PC with two 6-core CPUs. Relevant hardware configuration is listed in Table II. The PC runs Linux Operating System in the version 2.6.41.9-1.fc15.x86_64. The entire greedy name lookup program consists of about 2,500 lines of code, developed by C++ programming language. The part of multi-core parallel processing is developed using OpenMP API [16] in version 2.5.

TABLE II: Hardware configuration.

Item	Specification
CPU	Intel Xeon E5645×2 (6 cores, 2 threads, 1.6GHz)
RAM	DDR3 ECC 48GB (1333MHz)
Motherboard	ASUS Z8PE-D12X (INTEL S5520)

B. Experimental results

In Section IV-B1, we compare the name lookup throughput of the 5 methods in multi-core environment with different prefix tables and traces. Then the memory consumptions are evaluated in Section IV-B2. In Section IV-B3 and Section IV-B4, we further evaluate the scalability and update performance.

TABLE III: The name lookup throughput of different methods with one processing thread.

Prefix Table	Trace	Lookup Speed (MSPS)				
		CCNx	CCNx-PHT	Greedy-PHT	NameFilter	Greedy-SPHT
3M	Average	0.205	0.692	1.369	2.034	3.961
3M	Heavy	0.133	0.559	0.821	1.904	3.135
10M	Average	0.188	0.445	1.108	1.896	2.915
10M	Heavy	0.093	0.235	0.834	1.835	2.912

1) Throughput: First of all, we compare the name lookup speed on the 3M and 10M prefix table of the 5 methods with the average and heavy workload traces. Table III demonstrates the name lookup throughput of the 5 methods running in a single thread. CCNx achieves 0.205 MSPS and 0.133 MSPS on 3M prefix table under average workload and heavy workload, while CCNx-PHT realizes 0.692 MSPS and 0.559 MSPS, respectively. Greedy-PHT effectively improves the name lookup performance and speeds up to 1.369 MSPS and 0.821 MSPS, which is about 6.7 times speedup of CCNx and 1.98 times of CCNx-PHT. With negligible false positive, Greedy-SPHT even achieves 3.961 MSPS and 3.135 MSPS, which is almost 19.3×, 5.7×, 2.9× and 1.6× speedup of CCNx, CCNx-PHT, Greedy-PHT and NameFilter. On the 10M prefix table, we get the same performance improvements of Greedy-PHT and Greedy-SPHT.

On the other hand, the name lookup speed of Greedy-PHT jitters more severely than other methods under different traces, because Greedy-PHT speeds up the name lookup via optimizing the search path of name lookup which is strongly correlated with the searched names. Fortunately, Greedy-SPHT significantly reduces the jitter of name lookup speed by storing a key's signature in the entry instead of the key itself.

Figure 8 and 9 illustrate the name lookup throughput of the 5 methods with different number of parallel threads from 1 to 48. The name lookup speed increases monotonically along with the number of parallel threads grows while the thread number is less than 24. And the name lookup throughput reaches the peak with 24 parallel threads in our platform. Once the name lookup engine employs more than 24 threads, the lookup performance degrades and fluctuates. Because there are two Intel E5645 CPUs with 24 hardware threads in total, more than 24 parallel threads running in one program causes the competition of hardware threads with frequent program context-switch. The fastest name lookup speeds running with 24 parallel threads are presented in Table IV. On the 10M prefix table, Greedy-SPHT running with 24 parallel threads achieves 41.735 MSPS and 40.717 MSPS in average workload and heavy workload, respectively, which means 14.3 times speedup of Greedy-SPHT running with only one thread. Meanwhile, Figure 8 and Figure 9 also demonstrate that the name lookup throughput curve slope of Greedy-SPHT is higher than other methods, indicating that multiple threads support Greedy-SPHT better.

2) Memory: The memory consumption of the different methods is illustrated in Table V. CCNx needs 119.20 MB and 464.07 MB on 3M and 10M prefix table, respectively. Compared with CCNx, CCNx-PHT needs extra 12% memory

TABLE IV: The name lookup throughput of different methods with 24 parallel processing threads.

Prefix Table	Trace	Lookup Speed (MSPS)				
		CCNx	CCNx-PHT	Greedy-PHT	NameFilter	Greedy-SPHT
3M	Average	3.444	11.296	29.476	36.983	57.138
3M	Heavy	2.538	9.233	13.513	35.641	48.654
10M	Average	3.323	9.566	22.291	33.525	41.735
10M	Heavy	1.618	7.509	17.505	32.601	40.717

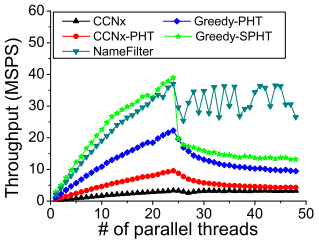

Fig. 8: The name lookup throughput of the methods with different parallel threads (10M prefix table, Average workload).

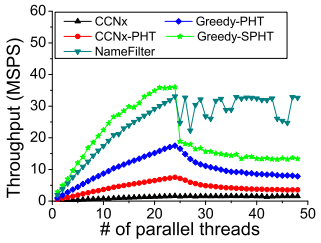

Fig. 9: The name lookup throughput of the methods with different parallel threads (10M prefix table, Heavy workload).

to construct FIB with basic string-oriented perfect hash table. For supporting greedy name lookup mechanism, Greedy-PHT stores additional information in FIB, so it consumes a little more memory than CCNx-PHT. By storing the signature of a key instead of storing the key itself, Greedy-SPHT significantly reduces the memory consumption to 72.95 MB and 247.20 MB on 3M prefix table and 10M prefix table, respectively. Greedy-SPHT saves about 46.8%, 47.5% and 53.7% memory of CCNx, CCNx-PHT and Greedy-PHT, respectively.

3) Scalability: From the name lookup throughput illustrated in Table IV and the memory consumption presented in Table V, we have already recognized that Greedy-SPHT has better lookup speed than other methods. However, we are still interested in foreseeing its performance trend as prefix table size grows. Toward this end, we partition each prefix table into ten equal-sized subsets, and progressively generate ten prefix tables for each of them; the k-th generated prefix table consists of the first k equal-size subsets. Experiments are then conducted on these 20 generated prefix tables derived from the 3M prefix table and 10M prefix table. Experimental results on lookup throughput and memory space requirement are presented from Figure 10 to 16, respectively.

As illustrated in Figure 10 and Figure 12, the name lookup throughput of the 4 methods trend to stabilize on average workload along with the prefix table size growing. And on heavy workload, the name lookup performance of CCNx and CCNx-PHT is still stable, but the name lookup speed of Greedy-PHT and Greedy-SPHT decreases with the table size increasing when the prefix table size has less than 5 million entries. When the table size is larger than 5 million, Greedy-PHT's and Greedy-SPHT's name lookup throughput become stable owing to that the greedy strategy dynamically optimizes the search path to keep the name lookup speed.

The comparison of the memory space requirements about the above 4 methods on different prefix table sizes is shown

TABLE V: The memory consumption of different methods.

Prefix Table	Memory Cosumption (MByte)				
	CCNx	CCNx-PHT	Greedy-PHT	NameFilter	Greed-SPHT
3M	119.20	131.19	147.75	64.73	72.95
10M	464.07	471.27	534.83	234.27	247.20

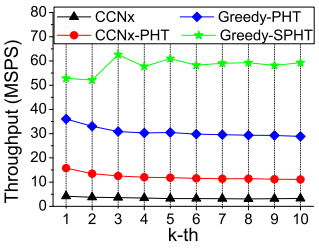

Fig. 10: The name lookup throughput of the 4 methods on different prefix table sizes (3M prefix table, Average workload).

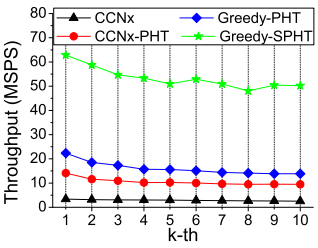

Fig. 11: The name lookup throughput of the 4 methods on different prefix table sizes (3M prefix table, Heavy workload).

in Figure 14. The memory consumption grows with linear scalability, which is consistent with our intuition. Meanwhile, the slope of Greedy-SPHT's curve is smaller than other methods, which means Greedy-SPHT is more suitable for large-scale prefix table.

As described in Section II-D, FIBs are reconstructed to facilitate the greedy name lookup mechanism. Figure 15 illustrates the extra number of entries (prefixes) along with the FIB size grows. Compared with the FIBs in CCNx, the reconstructed FIBs need store additional prefixes. Fortunately, the percentage of extra entries decreases as the FIB size increases.

The lookup throughput of NameFilter and Greedy-SPHT with different number of ports is presented in Figure 16. NameFilter's lookup speed decreases gradually with the increase of the number of ports in a router, while greedy name lookup mechanism is immune to the number of ports.

Therefore, from Figure 10 to Figure 16, we can conclude that Greedy-SPHT has good scalability on different prefix table sizes and is competent for large-scale NDN prefix table.

4) Update: As described in Section III-C, with the help of routing table in control plane, our improved string-oriented perfect hash table can support incremental insertion, modification and deletion. Figure 18 and Figure 19 present the insertion and deletion performances of the 4 methods. Greedy-SPHT supports about 0.75 million insertions and 3.08 million deletions per second, respectively. Meanwhile, the insertion/deletion performance of Greedy-SPHT is independent of the prefix table size.

Furthermore, Figure 17 demonstrates the name lookup performances of Greedy-PHT and Greedy-SPHT accompanied with different speed of updates (50% insertions and 50% deletions) under average workload and heavy workload, respectively. Given the operation of update only occupies one thread at a time, the impact of incremental update on the overall name lookup performance of Greedy-PHT and Greedy-SPHT can be negligible. As a result, greedy name lookup mechanism supports fast incremental update while keeping high performance of name lookup.

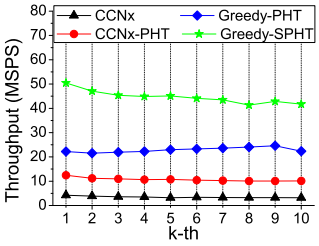

Fig. 12: The name lookup throughput of the methods on different prefix table sizes (10M prefix table, Average workload).

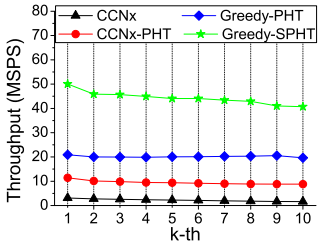

Fig. 13: The name lookup throughput of the methods on different prefix table sizes (10M prefix table, Heavy workload).

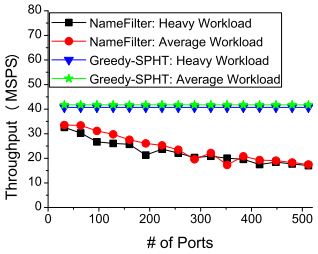

Fig. 16: The name lookup throughput of the methods on prefix tables with different number of ports (10M prefix table).

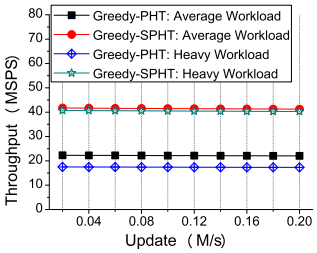

Fig. 17: The name lookup performance Greedy-PHT and Greedy-SPHT accompanied with updates.

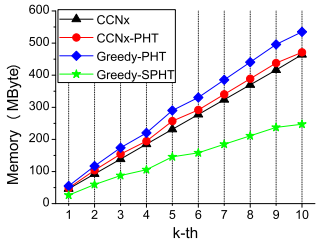

Fig. 14: The memory consumption of the methods on different prefix table sizes (10M prefix table).

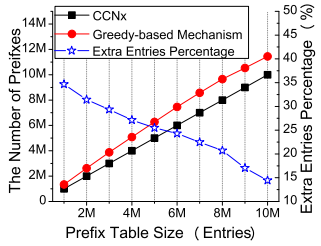

Fig. 15: The extra number of entries in greedy name lookup mechanism along with the FIB size grows (10M prefix table).

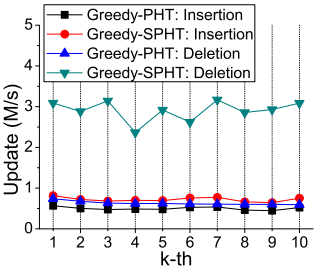

Fig. 18: The insertion and deletion performance of Greedy-PHT and Greedy-SPHT on 3M prefix table.

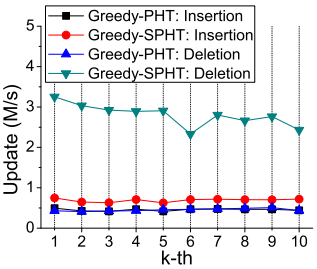

Fig. 19: The insertion and deletion performance of Greedy-PHT and Greedy-SPHT on 10M prefix table.

V. Related Work

Many recent research efforts and papers [5], [6], [17], [18], [19], [20], [21] have noted that we should move the current Internet architecture from point-to-point communication between hosts to content-centric or information-centric paradigm. Name-based forwarding as a fundamental technique in NDN, has been recently studied in the literature [22], [2], [23], [24], revealing the feasibility of routing based on hierarchical names instead of IP addresses from the viewpoint of working principle. However, longest prefix matching for hierarchical names can potentially slow down the lookup process.

Similar to IP-based longest prefix matching methods, name-based longest prefix matching mechanisms can be classified into two types: hardware-based methods and software-based methods. TCAM is well-known for its fast speed. A TCAM-based name lookup mechanism in NDN proposal [6] is presented. Unfortunately, from the viewpoint of cost and power saving, it can hardly make a practical option, given its low memory density, high price and excessive power consumption.

Character trie-based and hash-based methods are two kinds of name lookup approaches in NDN. In paper [23], the prefix table is constructed as a component trie, which transits from parent node to child node at the granularity of component. But a transition still needs to match the input component through an entire character trie, so the name lookup process takes too much time to search the longest prefix from root to the leaf node. Name Component Encoding method proposed in paper [25] speeds up the name lookup via encoding components to codes before searching in FIB. However, the mappings between components and codes are stored in a character trie, hence the encoding process will potentially slow down the name lookup remarkably. Yi Wang et al. [26] propose a GPU-based approach to implement wire speed name lookup. For better adapting to GPU's MDSI (Multiple Data Single Instruction) running mode, a trie-based multiple aligned transition array (MATA) data structure is proposed to construct prefix table. By exploiting GPU's massive parallel processing power, the GPU-based name lookup engine can achieve more than 60 MSPS (million searches per second) with a strict constraint of no more than 100μs per-packet lookup latency. This solution may be good to build software NDN routers, however, similar to TCAM-based methods, the GPU's huge power consumption will keep it from being used in the line-cards of a backbone hardware router.

In order to quickly determine which prefix to follow when lookup a name, hash techniques have been intensively studied. NameFilter [3] is a two-stage Bloom filter-based scheme for name lookup, in which the first stage determines the length of a name prefix, and the second stage looks up the prefix in a narrowed group of Bloom filters based on the results from the first stage. By optimizing the hash value calculation of name strings, as well as elaborating the data structure for storing multiple Bloom filters, NameFilter significantly reduces the memory access times compared with that of non-optimized Bloom filters. However, the first stage, searching all the candidate prefixes of a name in the Bloom filters to find the longest one, wastes a lot of computing resources. The probing times in the first stage Bloom filters can be reduced via improving the search strategy. In CCNx, all prefixes in the FIB are stored in a hash table. The name lookup process first generates all the possible prefixes from the searched name

and sorts the candidate prefixes from long to short. Then, the program looks up the candidate prefixes one by one until matching or failure. However, the name lookup performance is still too low to be satisfied [2] because of the slow hash table search and poor longest prefix search strategy.

Therefore, in this paper, we propose a fast name lookup scheme which includes the greedy strategy for optimizing name lookup search path and the improved string-oriented near-perfect hash table. For improving the name lookup performance, we reconstruct the data structure of FIB and leveraging greedy strategy to optimize the search path of longest prefix matching. Furthermore, inspired by perfect hash table designs in previous works [9], [10], [27], [28], [29], we design a string-oriented near-perfect hash table to reduce memory consumption and boost name lookup speed by storing a key's signature in the entry instead of storing the key itself. Our name lookup mechanism addresses all these performance issues including high lookup speed, memory efficient, as well as scalability and fast incremental update.

VI. Conclusion

In this paper, we propose a fast name lookup scheme for Named Data Networking via reconstructing the data structure of FIB to support greedy search strategy and improving the basic string-oriented perfect hash table. Though the reconstructed FIBs need store additional prefixes, the percentage of extra entries decreases as the FIB size increases. A greedy strategy based on the distribution of prefixes effectively boosts the name lookup speed via reducing the times of perfect hash table search. Then, the improved string-oriented near-perfect hash table further reduces the memory consumption and speeds up name lookup by storing a key's signature instead of the key itself. Compared with NameFilter [3], our name lookup scheme not only achieves faster lookup speed with negligible extra memory cost, but also is immune to system factors of the prefix distribution and the number of ports in a router which will influence the name lookup performance of NameFilter. Extensive experiments on large prefix tables also demonstrate that our name lookup scheme can greatly reduce memory cost and boost name lookup speed while supporting high incremental update and improving the scalability.

References

[1] CCNx project. http://www.ccnx.org/.

[2] Haowei Yuan, Tian Song, and P. Crowley. Scalable NDN Forwarding: Concepts, Issues and Principles. In *International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9, 2012.

[3] Yi Wang, Tian Pan, Zhian Mi, Huichen Dai, Xiaoyu Guo, Ting Zhang, Bin Liu, and Qunfeng Dong. NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters. In *Infocom mini-conference'13*, 2013.

[4] CONNECT project. http://anr-connect.org/.

[5] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking Named Content. In *CoNEXT'09*, pages 1–12, 2009.

[6] Lixia Zhang, Deborah Estrin, Van Jacobson, and Baichuan Zhang. Named data networking (ndn) project. http://www.named-data.net/. 2010.

[7] Van L. Jacobson and James D. Thornton. Method and apparatus for forwarding packets with hierarchically structured variable-length identifiers using an exact-match lookup engine. Number EP2214355. August 2012.

[8] Donald E. Knuth. *Art of Computer Programming, volume 1/Fundamental Algorithms; volume 3/Sorting and Searching*. Addison-Wesley, 1973.

[9] M. Fredman and J. Komlos. On the Size of Separating Systems and Families of Perfect Hash Functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984.

[10] Djamal Belazzougui, Fabiano Botelho, and Martin Dietzfelbinger. Hash, Displace, and Compress. In *Algorithms - ESA 2009*, volume 5757, pages 682–693. 2009.

[11] Torben Hagerup and Torsten Tholey. Efficient Minimal Perfect Hashing in Nearly Minimal Space. In *STACS 2001*, volume 2010 of *Lecture Notes in Computer Science*, pages 317–326. 2001.

[12] Yi Lu, B. Prabhakar, and F. Bonomi. Perfect Hashing for Network Applications. In *Information Theory, 2006 IEEE International Symposium on*, pages 2774–2778, july 2006.

[13] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, November 1979.

[14] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a Sparse Table with 0(1) Worst Case Access Time. *J. ACM*, 31(3):538–544, June 1984.

[15] Name Lookup Project. http://www.namelookup.org/.

[16] The OpenMP API specification for parallel programming. http://openmp.org/wp/.

[17] D.R. Cheriton and M. Gritter. TRIAD: A New Next-Generation Internet Architecture. http://www-dsg.stanford.edu/triad/, July 2000.

[18] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A Data-Oriented (and beyond) Network Architecture. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM'07, pages 181–192, 2007.

[19] Luis M. Correia, Henrik Abramowicz, Martin Johnsson, and Klaus Wnstel. *Architecture and Design for the Future Internet: 4WARD Project*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[20] Nikos Fotiou, Pekka Nikander, Dirk Trossen, and George C. Polyzos. Developing Information Networking Further: From PSIRP to PURSUIT. volume 66, pages 1–13. 2012.

[21] Diego Perino and Matteo Varvello. A reality check for content centric networking. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, ICN'11, pages 44–49, 2011.

[22] Haesung Hwang, Shingo Ata, and Masayuki Murata. A Feasibility Evaluation on Name-Based Routing. In *IP Operations and Management*, volume 5843 of *Lecture Notes in Computer Science*, pages 130–142. 2009.

[23] Yi Wang, Huichen Dai, Junchen Jiang, Keqiang He, Wei Meng, and Bin Liu. Parallel Name Lookup for Named Data Networking. In *IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1 –5, dec. 2011.

[24] Cheng Yi, Alexander Afanasyev, Lan Wang, Beichuan Zhang, and Lixia Zhang. Adaptive forwarding in named data networking. *SIGCOMM Comput. Commun. Rev.*, 42(3):62–67, June 2012.

[25] Yi Wang, Keqiang He, Huichen Dai, Wei Meng, Junchen Jiang, Bin Liu, and Yan Chen. Scalable Name Lookup in NDN Using Effective Name Component Encoding. In *IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 688–697, june 2012.

[26] Yi Wang, Yuan Zu, Ting Zhang, Kunyang Peng, Qunfeng Dong, Bin Liu, Wei Meng, Huichen Dai, Xin Tian, Zhonghu Xu, and Di Yang. Wire Speed Name lookup: A GPU-based Approach. In *NSDI'13*, 2013.

[27] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1st edition, 1984.

[28] Jaikumar Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, 41(4):203 – 207, 1992.

[29] Fabiano Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and Space-Efficient Minimal Perfect Hash Functions. In *Algorithms and Data Structures*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. 2007.