Scalable Name Lookup in NDN Using Effective Name Component Encoding

Yi Wang, Keqiang He, Huichen Dai, Wei Meng, Junchen Jiang, Bin Liu
Tsinghua University,
Beijing, China, 100084
wy@ieee.org, {hkq09, dhc10, mw08, jyc07}@mails.tsinghua.edu.cn, liub@tsinghua.edu.cn

Yan Chen
Northwestern University
Evanston, IL USA, 60208
ychen@northwestern.edu

***Abstract*—Name-based route lookup is a key function for *Named Data Networking (NDN)*. The NDN names are hierarchical and have variable and unbounded lengths, which are much longer than IPv4/6 address, making fast name lookup a challenging issue. In this paper, we propose an effective Name Component Encoding (NCE) solution with the following two techniques: (1) A code allocation mechanism is developed to achieve memory-efficient encoding for name components; (2) We apply an improved State Transition Arrays to accelerate the longest name prefix matching and design a fast and incremental update mechanism which satisfies the special requirements of NDN forwarding process, namely to insert, modify, and delete name prefixes frequently. Furthermore, we analyze the memory consumption and time complexity of NCE. Experimental results on a name set containing 3,000,000 names demonstrate that compared with the character trie NCE reduces overall 30% memory. Besides, NCE performs a few millions lookups per second (on an Intel 2.8 GHz CPU), a speedup of over 7 times compared with the character trie. Our evaluation results also show that NCE can scale to accommodate the potential future growth of the name set size.**

***Index Terms*—Named Data Networking; Name Prefix Longest Matching; Name Component Encoding;**

I. Introduction

Named Data Networking (NDN) [1] is proposed recently as a clean-slate network architecture for future Internet, which no longer concentrates on "where" the information is located, but "what" the information (content) is needed. NDN uses names to identify every piece of contents instead of IP addresses for hardware devices attached to IP network.

NDN forwards packets by names, which implies a substantial re-engineering of forwarding and its lookup mechanism. We highlight the challenges as follows:

1) *Bounded processing time and high throughput.* NDN names, unlike fixed-length IP addresses, may have variable lengths without an externally imposed upper bound. This makes line-speed name lookup extremely challenging as arbitrarily long name will cost a lookup time linear to its length, rather than a fixed time using traditional tree-based or hash-based method.
2) *Longest name prefix matching.* NDN names, unlike today's classless IP addresses, have hierarchical structure and coarser granularity, consisting of a series of delimited *components*. NDN's longest prefix matching differs from that of IP in the way that NDN must match a prefix at the end of a component, rather than at any digit in IP. Therefore, traditional prefix matching algorithms will be far less efficient in NDN name lookup.
3) *High update rate.* NDN name lookup is accompanied with more frequent updates than Forwarding Information Base (FIB) in today's router, for the reason that, besides routing refreshing, NDN name table update is also conducted dynamically no matter when a new content is inserted or an old content is replaced. As this information is designed to be stored together with forwarding information in NDN router, the name lookup must support fast insertion and deletion with reasonably low overhead.

Traditionally, character trie is used to represent name prefixes. The character trie-based longest prefix matching algorithms often have O(nm) time complexity in the average case (n means the number of decomposed name components and m represents the number of children per node), which cannot satisfy the need of high speed lookup. Besides, the tree structured implementation is memory-inefficient. An alternative solution is hashing the name as a whole to an identifier for memory compression. However, this method cannot be applied to the longest prefix matching since it takes the whole name as a key. An improved hash-based approach decomposes the name to components and encodes each component to an identifier using hash functions directly. This method is memory-efficient and suitable for the longest prefix matching. Nevertheless, false positive caused by hash collision leads to potential lookup failure. In other words, some name prefixes will be hijacked by others which have the same identifier sequence. False positive will destroy the accuracy of routing and the integrity of the router function, so hash-based methods cannot be well adopted to name lookup in NDN.

Thus, in this paper, we propose a Name Component Encoding approach, which effectively reduces the memory cost and accelerates name lookup in NDN. Especially, we make the following contributions:

1) We propose an effective name component encoding mechanism to cut down the number of component codes and each component's code length without loss of the correctness of longest name prefix matching. Meanwhile, the component encoding mechanism separates

the encoding process with the longest prefix matching, makes it possible to use parallel processing technique to accelerate name lookup. In addition, this approach limits the name lookup time to the upper bound of the maximum time between the component encoding process and the longest encoded name prefix matching.

2) We develop the State Transition Arrays (STA) to implement the Component Character Trie and Encoded Name Prefix Trie. State Transition Arrays could reduce the memory cost while guaranteeing high lookup speed. Besides, we propose algorithms based on STA to support fast incremental construct, update, remove and modify. Further, both the memory cost and time complexity are analyzed theoretically.
3) Experiments on three datasets are carried out to validate the correctness of the proposed Name Component Encoding solution. We test Name Component Encoding in terms of memory compression, average lookup time, speedup as well as average packet delay and compare it with the traditional character trie method. The experimental results on 3,000,000 names set demonstrate that NCE could achieve memory compression ratio greater than 30%, and it can perform 1.3 million lookups per second on an Intel 2.8 GHz CPU which can process 2.1 million IP lookups per second. Besides, NCE can handle more than 20,000 updates per second in the worst case. Furthermore, average length of a name is shortened from 166 bits to 43 bits. The results also show that NCE could not only be used for small name sets, but also serve larger name sets.

The rest of this paper is organized as follows. The background of packet forwarding process in NDN is introduced in Section II. Section III describes longest name prefix lookup problems caused by NDN names. To solve the problem, Section IV proposes the Name Component Encoding solution. In Section V, we analyze the memory cost and time complexity theoretically. Section VI evaluates NCE in terms of memory compression, lookup time, speedup and average packet delay on three different datasets. Section VII is the related work and we conclude this paper in Section VIII.

II. Packet Forwarding in NDN

A. NDN Background

NDN, formerly known as CCN [2], is a novel network architecture proposed by [1] recently. Different from current network practice, it concentrates on the content itself ("what"), rather than "where" information is located. Despite of its novelty, NDN operations can be grounded in current practice, routing and forwarding of NDN network are semantically the same as that of IP network. What differs is that, every piece of content in NDN network is assigned a name, and NDN routes and forwards packets by these names, rather than IP addresses.

Names used in a NDN network are dependent on applications and are opaque to the network. An NDN name is

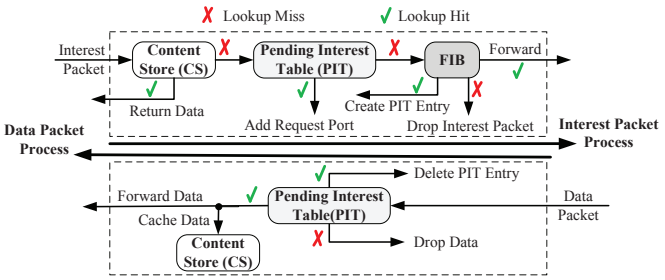

Fig. 1. Packet Forwarding Process in an NDN Router

hierarchically structured and composed of explicitly delimited *components*, while the delimiters, usually slash ('/') or dot ('.'), are not part of the name. For example, a map service provided by Google has the name */com/google/maps*, and *com*, *google* and *maps* are three components of the name. The hierarchical structure, like that of IP address, enables name aggregation and allows fast name lookup by longest prefix match, and aggregation in turn is essential to the scalability of routing and forwarding systems. Applications can use this hierarchical structure to represent the relationships between data, and evolve the naming scheme to best fit their demands independently from the network. In this paper, we utilize the *hierarchically* reversed domain names as NDN names.

B. Packet Forwarding in NDN

Communications in NDN are driven by the data requesters, i.e., the data consumers. A data requester sends out an *Interest* packet with the desired content name, routers forward the Interest packet by looking up its name in the *Forwarding Information Base (FIB)*, when the Interest packet reaches a node that has the required data, a *Data* packet is sent back to the requester. Fig. 1 illustrates the Interest and Data packets forwarding process in an NDN router in a high-level. Next we briefly describe the forwarding process and introduce a few basic concepts.

Once the Interest reaches a router, the router first searches the request name in the *Content Store (CS)*, which caches the data that has been forwarded by this router. If the desired Data exists in the CS, the router directly returns the requested Data. Otherwise, the name is checked against the *Pending Interest Table (PIT)*, where each entry contains the name of the Interest and a set of interfaces from which the matching Interests have been received. If the Interest name matches a PIT entry, it means one Interest for this data has been forwarded to upstream while the response Data does not arrive. After the failure of CS and PIT lookup, the Interest is forwarded by looking up its name in the FIB, and a new PIT entry is assigned the request name and interface.

When the Data packet arrives, the router finds the matching entry in the PIT firstly, and forwards the Data packet to all the interfaces listed in the matching PIT entry or drops the Data. Then the router removes the corresponding PIT entry, and caches the Data in the CS.

III. Name Prefix Trie for Name Lookup

Intuitively, three tables need three separate indexes, however we can integrate the indexes of PIT, CS and FIB to a single one and only one name lookup operation is actually performed when a packet comes, thus improving the forwarding efficiency. NDN names are composed of explicitly delimited components. Hence they can be represented by Name Prefix Trie (NPT). Therefore, we make use of the NPT to organize the integrated index. NPT is shown in Fig. 2, each edge of which stands for a name component, and a more illustrative example is shown in 3. The Name Prefix Trie is of component granularity, rather than character or bit granularity, since the longest name prefix lookup of NDN names can only match a complete component at once, i.e., no match happens in the middle of a component. It should be pointed out that the Name Prefix Trie is not necessarily a binary tree, which differs from that of the IP address prefix tree. Each edge of the NPT stands for a name component and each node stands for a lookup state. Name prefix lookups always begin at the root.

When an Interest Packet arrives, the longest prefix lookup for the NDN name of this packet starts, it firstly checks if NDN name's first component matches one of the edges originated from the root node, i.e., the level-1 edge. If so, the transfer condition holds and then the lookup state transfers from the root node to the pointed level-2 node. The subsequent lookup process proceeds iteratively. When the transfer condition fails to hold or the lookup state reaches one of the leaf nodes, the lookup process terminates and outputs the index that the last state corresponds to. For example, in Fig. 2, the longest prefix lookup for NDN name */com/parc/videos/USA/2011/B.mpg* starts from the root node. The first component *com* matches a level-1 edge of the NPT and the lookup state transfers from root to the corresponding child node. The second component *parc* matches a level-2 edge and it returns an index which points to one specific entry in FIB. When 6th component *B.mpg* does not match any level-6 edge, a new node needs to be inserted to the NPT, meanwhile a corresponding entry is added to PIT. However, if the request name is */com/parc/videos/USA/2011/A.mpg*, this name could be completely matched in NPT and the corresponding CS entry index would be found. When a Data Packet arrives, it will perform the longest prefix lookup for the given NDN name as stated above, too. Meanwhile it will modify NPT while both PIT and CS must be updated.

NPT can be constructed as Name Character Trie (NCT) which needs a large amount of memory to store the states and transitions. And one longest prefix matching needs $O(mn)$ in the average case, m is the average number of children per node and n is the average length of names.

B. Michel *et al.* [3] and Z. Zhou *et al.* [4] apply hash function to compress URL components, which has good compression performance. But false positive arises from the hash collision, which causes the confusion of distinguishing the real prefix with the false prefix. For example, suppose */com/parc* and */edu/cmu* are both hashed to the same identifier

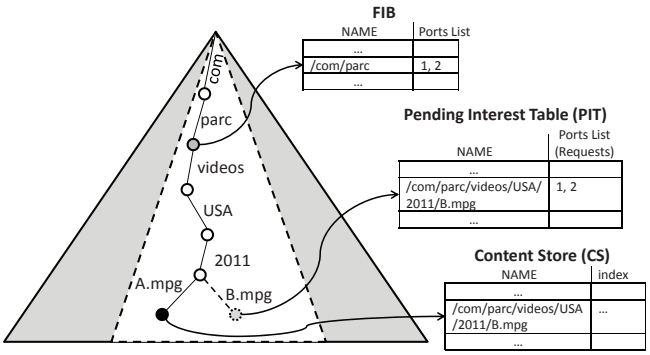

Fig. 2. Name Prefix Trie for CS, PIT and FIB indexes

5612, and */com/parc* is added to the hash table earlier than */edu/cmu*. When the lookup algorithm takes */edu/cmu* as input, it will return a pointer to */com/parc*, and */edu/cmu* will never be found. The false positive will destroy the exactness of routing and integrity of the router function, so hash based methods are not suitable for name prefix lookup in NDN.

Based on the observation that an NDN name set has limited number of components, there is an opportunity to use encoding based method to compress the memory and improve the lookup performance. Applying encoding based method to NPT, we should solve the following problems.

1) High-speed longest name prefix matching. Accelerating the name lookup in NDN is the major objective of name component encoding mechanism.
2) Fast component code lookup. When a packet arrives, the corresponding codes of the name's components must be looked up before starting the longest prefix matching. The router's throughput and packets' delay are affected by the speed of component code lookup process.
3) Low memory cost. An effective encoding based method should reduce the total memory cost, which includes two basic parts, the memory used to store names' codes lookup table and the storage used to implement NPT.
4) Good update performance. As we have described above, NPT is updated frequently. Poor update performance will become the bottleneck of the longest name prefix matching.

IV. Name Component Encoding (NCE)

In this section, we propose the Name Component Encoding (NCE) solution to solve the problems stated above. In NCE, a memory efficient Code Allocation Mechanism is designed to shorten the bytes which represent a code by reducing the total number of codes. Then we ameliorate the State Transition Arrays (STA) techniques for trie structure to compress memory size and accelerate longest prefix lookup. At last, we present the algorithms of managing the STA to satisfy the frequent name update in NDN.

In order to describe the Code Allocation Mechanism clearly, three definitions are given first.

Definition 1: A component (edge) C_i **belongs to** a state (node) S_j when C_i leaves S_j to another state in a trie.

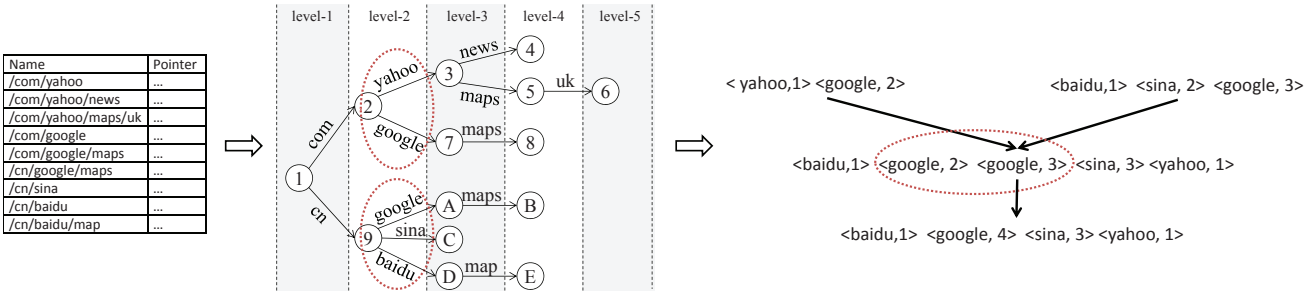

Fig. 3. Illustration of Code Allocation Mechanism

Definition 2: **Original Collision Set** is a set of components, and all the components belong to a given state S_j. Each component in the set should be encoded with different codes to avoid collision.

Definition 3: **Merged Collision Set** is a component set. Let CS_i, CS_j be an Original Collision Set or a Merged Collision Set. $CS_m = CS_i \bigoplus CS_j$ means CS_i, CS_j are merged to CS_m by re-encoding the collision components.

In the rest part of this section, we introduce the four major parts of NCE: 1) allocate a code to each name component and transfer NPT to ENPT; 2) construct STA to represent ENPT; 3) map a single name component to its corresponding code; 4) manage STA to support incremental insertion, removal and modification of name prefixes.

A. Code Allocation Mechanism

As mentioned in Section III, names are represented by NPT. For example, as shown in Fig. 3, the given 9 names can be organized as an NPT with 14 nodes. Different components (edges) leaving a given node should be encoded differently and these components comprise an Original Collision Set according to *Definition* 2. In Fig. 3, *yahoo* and *google* which both belong to node 2 can be encoded as <yahoo,1>[1] and <google,2> respectively. The component set {yahoo,google} is an Original Collision Set of state 2.

If the component codes are produced at the granularity of Original Collision Set, that is, different Original Collision Sets are encoded independently, the produced code depends on both the component itself and its corresponding node. For example, the level-2 components starting from node 2 and node 9 are {yahoo, google} and {google, sina, baidu}, respectively. Suppose they are encoded as {<yahoo, 1>, <google, 2>} and {<baidu, 1>, <sina, 2>, <google, 3>}. We can find that the same component *google* is encoded differently in the two Original Collision Set (*google* is encoded as 2 in the first Original Collision Set and 3 in the second Original Collision Set). So given a component without node information, we cannot predict the corresponding code. Thus we need alterative solutions.

One straightforward method is to assign unique codes to all the components in NPT, which constitute a Merged Collision Set. However, there will be a large amount of codes and its code is of great length.

[1] *yahoo* is encoded as 1.

Based on the fact that components of domains are separated by special delimiters, we can get which level a given component belongs to. The component code lookup process could be carried out at each level. The Original Collision Sets at the same level are merged to a Merged Collision Set. If a specific component is assigned different codes in at least two Original Collision Sets, we re-assign the component's code as the maximal code number of these Original Collision Sets plus 1 (i.e., if the maximal code number of the two Original Collision Sets is N, the component will be encoded as $N+1$). Fig. 3 illustrates the procedure of merging Original Collision Sets to larger Merged Collision Set.

Please note that different components in a Merged Collision Set may have the same code. For example, in Fig. 3, *baidu* and *yahoo* are both encoded as 1 after the merging procedure. This property will shorten the number of produced codes and each code's length. Besides, Theorem 1 proves that the Code Allocation Mechanism keeps the correctness of name lookup.

Theorem 1: The Code Allocation Mechanism keeps the correctness of the longest name prefix lookup.

Proof: Suppose two names $A = C_{a1}..C_{ai}..C_{am}$ and $B = C_{b1}..C_{bi}..C_{bm'}$ are encoded to two code sequences $E_{a1}..E_{ai}..E_{am}$ and $E_{b1}..E_{bi}..E_{bm'}$. Two components C_{ai} and C_{bi} form the same level i ($C_{ai} \neq C_{bi}$) have the same code $E_{ai} = E_{bi}$. Let $S_{a1}..S_{ai}..S_{am+1}$ and $S_{b1}..S_{bi}..S_{bm'+1}$ be the lookup path of A and B. If and only if $S_{ai} = S_{bi}$, the same code E_i of C_{ai} and C_{bi} causes collision. Because the lookup path starts from the root of a trie, we can deduce that $E_j = E_{aj} = E_{bj}$ where $j < i$. Since there is only one Original Collision Set in the first level, according to the definition of Original Collision Set and $E_{a1} = E_{a2}$, we can get $C_{a1} = C_{b1}$ and $S_2 = S_{a2} = S_{b2}$. C_{a2} and C_{b2} are transited from the same state S_2 with the same label E_2, so C_{a2} and C_{b2} belong to the same Original Collision Set, i.e., $C_{a2} = C_{b2}$. Recursive derivation to the level-i, we can get that C_{ai} and C_{bi} belong to the same Original Collision Set. Since $E_{ai} = E_{bi}$, we get $C_{ai} = C_{bi}$, which is in contradiction with the previous assumption $C_{ai} \neq C_{bi}$. ■

We assign the produced codes to the corresponding components (edges) in NPT and the new trie is called Encoded Name Prefix Trie (ENPT). For example, the trie illustrated in Fig. 4 is the corresponding ENPT of the NPT shown in Fig. 3. In the next subsection, we will introduce the State Transition Arrays mechanism for ENPT.

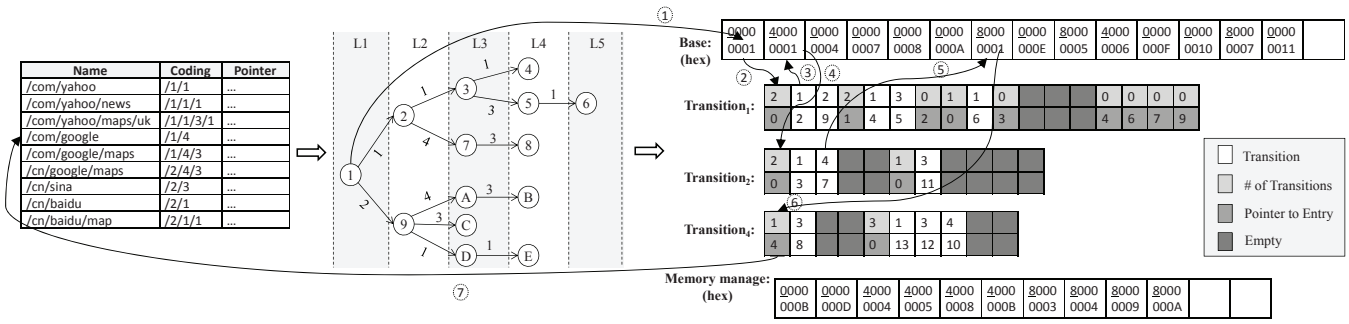

Fig. 4. Illustration of State Transition Arrays for Encoded Name Prefix Trie

B. State Transition Arrays for Encoded Name Prefix Trie (ENPT-STA)

As shown in Fig. 4, State Transition Arrays are used to implement ENPT. There are 3 types of arrays: Base Array, Transition Array, and Manage Array. Transition Array includes three arrays which have different entry size. For convenience, suppose all the arrays discussed in this paper are indexed from 1 and we refer the i-th entry of array A as A:i.

The entry of Base Array is 4 bytes. Base:i represents state i in ENPT. The first two bits of the Base entry mean in which Transition Array the associated state's information is stored. 00 means the information is stored in Transition_1, 01 means Transition_2 and 10 means Transition_4. The left bits represent the offset value of the targeted array. Due to the fact that components are encoded to variable-length codes, the entry size of Transition_1, Transition_2 and Transtion_4 is 5 bytes, 6 bytes and 8 bytes, respectively. A node's transitions are stored in three Transition Arrays according to the maximal code of the transitions. When the maximal code is less than 2^8, all transitions of the node can be stored in Transition_1. If the maximal code is equal or greater than 2^8 and less than 2^{16}, the node's transitions are store in Transition_2. Otherwise, all transitions of the node are store in Transition_4. For example, in Fig. 4, Base:7 is 0x80000001, that means the information about state 7 of the ENPT is stored in Transition_4 and Transition_4:1 records the information about state 7 since the left bits of Base:7 equals to 1.

Transition Array has two types of entries (denoted as *indicator* and *transition*). *Indicator* records the state's transition number k (represented by the first number of the entry) and the entry pointer (represented by the second number). If the state points no entry in FIB, PIT or CS, the pointer is assign to 0. The following k entries' type is *transition*. The first number of *transition* represents the component code produced by the Code Allocation Mechanism and the second number represents next state. The *transitions* are sorted according to the first number to support binary search. For instance, Transition_4:1 is an *indicator*. The first number of this entry means there is 1 transition leaving state 7. The second number of this entry points to the 4th entry of FIB.

The entry in Manage Array indicates the free entries in the Transition Arrays. The first two bits of a Manage entry have the same meaning with Base Array. The odd-numbered entry of Manage indicates the start position of a segment of free entries, and the next even-numbered entry of Manage indicates the end position of this segment. In Fig. 4, Manage:5 and Manager:6 mean that the entries from Transition_1:8 to Transition_1:11 are free.

We take an example to illustrate how ENPT-STA works. In Fig. 4, suppose the given name is */com/google* with code sequence */1/4*. We will explain the lookup procedure step by step.

1) Step 1, the lookup procedure starts from Base:1 which corresponds to state 1 (root) of the ENPT.
2) Step 2, Base:1=0x00000001 means state 1 (root) information is store in Transition_1:1.
3) Step 3, Transition_1:1 is an *indicator*. The first number of this entry means there are two transitions from state 1 and the following 2 entries are *transitions*. After binary searching in the follow two entries, code 1 is matched with the first number in Transition_1:2. Then it turns to Base:2 since the second number of Transition_1:2 is 2.
4) The lookup procedure proceeds iteratively. Finally code sequence */1/4* is completely matched, and an entry index which points to the 4th entry of FIB is returned.

C. State Transition Arrays for Component Character Trie (CCT-STA)

A given name is firstly decomposed to several components. Before the longest name prefix lookup, we need to look up each component's code first. The component entry includes the corresponding code and a state list to which this component belongs. The component set is constructed as a Component Character Trie (CCT), which also can be implemented by STA. The STA for CCT are similar to those of STA for ENPT. Fig. 5 shows an example of State Transition Arrays for CCT.

There are two types of arrays in STA. One is the Base Array and the other is Transition Array. Transition Array includes two types of entries : *indicator* and *transition*, which have the same meaning with ENPT-STA. The component code lookup process in CCT-STA is similar to the lookup in ENPT-STA. For example, in Fig. 5, if *cn* is the input component, the lookup starts from Base:1, i.e., root of the Component Character Trie. The first number of Transition:1 is 1, this means only 1 transition leaving the root. The first number of Transition:2 matches character *c* and it turns to Base:2 since the second

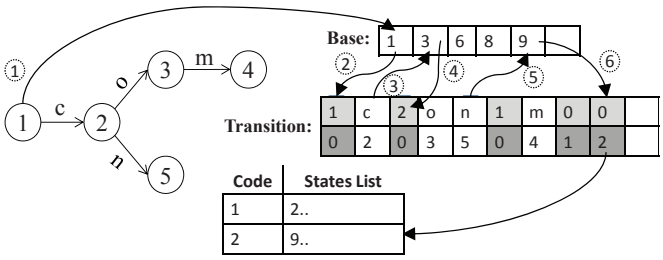

Fig. 5. Example of State Transition Arrays for Component Character Trie

number is 2. It performs the above procedure and finally the matching entry of *cn* is returned.

D. Management of the State Transition Arrays

The name insertion process is formally described in Algorithm 1. $Name$ represents the input name that will be decomposed into k components, which are denoted as C_1, C_2, $\cdots$, C_k. CCT_i is the level-i CCT, and S represents the current state in an ENPT-STA T. Given CCT_i and the i-th component C_i, lookup operation is performed to get C_i's corresponding code and the state list $List_s$ to which C_i belongs. If the code does exists and $S \in List_s$, it turns to the next state in ENPT-STA. Otherwise, add C_i to CCT_i and create a new state as well as the corresponding transition to ENPT-STA.

The $Decompose(name)$ function splits the input $name$ to components by recognizing the delimiters. The procedure of looking up the component's code is implemented by $LookupCode(CCT_i, C_i)$, which returns a code E_i and a state list $List_s$. It is probable that S has a component C_j encoded to E_i when $C_i \neq C_j$. So we must confirm whether $S \in List_s$ to guarantee the correctness. If $S \in List_s$, it suggests C_i, C_j belong to the same state S, and we can get $C_i = C_j$ according to the definition of Original Collision Set. The $Add_c(CCT_i, S, C_i)$ function is used to insert a new component C_i (belongs to S) to the level-i CCT, and the $Add_S(T, S, E_i)$ function creates a new state (node) S' to ENPT-STA T which transfers from S with label E_i, and returns S' as current node S.

For example, in Fig. 4, */cn/google* is inserted to the ENPT-STA as a new name. It is decomposed to *cn* and *google* firstly. Then we get the *cn*'s code $E_1 = 2$ and a state list $\{1\}$. Because the list $\{1\}$ contains $S = 1$, we transfer to the next state $S = 9$ and look up next component *google*. The $LookupCode()$ function returns the code $E_2 = 2$ and list $\{2\}$. However, $S = 9$ is not in list $\{2\}$, that we should add *google* to CCT_2. At the same time, we need to re-assign a code to *google*, since CCT_2 already contains *google*'s code 2. Suppose the maximal code in state 2 and state 3 are 2 and 3, respectively. Then we assign $E_2 = 4 = 3 + 1$ to *google*. At last, a new state A is created and a transition from 9 to A with label $E_2 = 4$ is inserted to ENPT-STA.

We can build ENPT incrementally by inserting names one by one as described in Algorithm 2. The delete process can be implemented by setting the second number of corresponding *transition* entry to 0. And the updated process could be implemented by modifying the Transition Array directly.

Algorithm 1 Insert a Name to ENPT-STA (INENPT)

```
1: procedure INENPT(name, T, CCT_1, ..., CCT_K)
2:     S ← 1
3:     (C_1, C_2, ..., C_k) ← Decompose(name)
4:     for i ← 1 to k do  ▷ k is the number of components
5:         (E_i, List_s) ← LookupCode(CCT_i, C_i)
6:         if E_i ≠ NULL and S ∈ List_s then
7:             S ← Transition(T, S, E_i)
8:         else
9:             CCT_i ← Add_C(CCT_i, S, C_i)
10:            (T, S) ← Add_T(T, S, E_i)
11:        end if
12:    end for
13: end procedure
```

Algorithm 2 Bulidng ENPT-STA

```
1: procedure BENPT(name_1, name_2, ..., name_N)
2:     (CCT_1, CCT_2, ..., CCT_K) ← NULL, T ← NULL
3:     for i ← 1 to N do      ▷ N is the number of Names
4:         INENPT(name_i, T, CCT_1, ..., CCT_K)
5:     end for
6:     return T, CCT_1, ..., CCT_k
7: end procedure
```

V. Analysis

For convenience, we summarize the main notations used in this section in TABLE I.

TABLE I
Table of Notations

$Nodes(A)$	calculates the number of nodes in A, which can be a trie or an array
$Edges(A)$	calculates the number of edges in A, which can be a trie or an array
n	the average length of a name
m	the average number of children per node in ENPT
k	the average number of components in a name
n_c	the average number of characters in a component
m_c	the average number of children per node in CCT
m_t	the average number of children per node in NCT
m_s	the average number of states which a component belongs to
P	the number of parallel encoding modules
α	the base memory size of a node
β	the base memory size of an edge

A. Space Complexity Analysis

In a trie T, $Nodes(T) = Edges(T) + 1$. The memory size of a trie can be calculated according to the following Equation,

$$\begin{aligned} Memory &= Nodes(T) * \alpha + Edges(T) * \beta \\ &= Nodes(T) * (\alpha + \beta) - \beta \end{aligned} \quad (1)$$

For the traditional Name Character Trie, every node at least needs a pointer to the edges list, a pointer to the matching entry (CS, PIT, FIB entry), and a list of edges which includes a key (character), and a pointer to the next trie node and a pointer to its brother edge. Every pointer needs 4 bytes, a character needs 1 byte, and the total memory can be calculated by Equation 1.

Here $\alpha = 8$ and $\beta = 9$, so the memory cost of a character trie is $17 * Nodes(NCT) - 9$.

Let State Transition Arrays construct the NCT, one entry in Base Array and one entry in the Transition Array are needed to represent a node. And an edge needs one entry in Transition array. Here, one Base Array entry needs 4 bytes and one Transition Array entry occupies 5 bytes. Therefore, we have $\alpha = 9$ and $\beta = 5$, and the total memory is $14 * Nodes(NCT) - 5$.

We use one Base Array and three Transition Arrays to organize the ENPT. If the state's transitions are stored in Transition$_1$, we get $\alpha = 9, \beta = 5$. Similarly, $\alpha = 10, \beta = 6$ for Transition$_2$ and $\alpha = 12, \beta = 8$ for Transition$_4$. We use Equation 1 to calculate the total memory. Thus, the memory cost of ENPT is $14 * Nodes(\text{Transition}_1) + 16 * Nodes(\text{Transition}_2) + 20 * Nodes(\text{Transition}_4) - 19$, where $Nodes(\text{Transition}_1) + Nodes(\text{Transition}_2) + Nodes(\text{Transition}_4) = Nodes(ENPT)$.

In summary, compared with NCT, NCE utilizes the following three parts to reduce storage overhead.

1) NCE uses State Transition Arrays to construct the NCT, and the memory cost can be reduced at least save $1 - \frac{14*Num(nodes)-5}{17*Num(nodes)-9} \approx 17.64\%$.
2) Code Allocation Mechanism reduces the number of components by merging the Original Collision Set at the same level.
3) NCE stores the transitions in different sizes of Transition Arrays. Compared with the method that uses Transition$_4$ only, it can reduce the memory overhead further.

B. Time Complexity Analysis

In NCE, the longest name prefix matching contains two steps, (1) finds the components' corresponding codes in CCT-STA and (2) looks up codes in ENPT-STA. The basic lookup process of a component in CCT-STA has $O(n_c \log(m_c))$ complexity in the average case, since binary search can be proceeded to find the matching key in the node's transitions which have been sorted. Similarly, a longest prefix matching in ENPT-STA needs $O(k \log(m))$. So, a name lookup has $O(kn_c \log(m_c) + k \log(m))$ complexity when the lookup is proceeded serially. If there are P parallel code lookup modules, the complexity can be reduced to $O(\frac{kn_c \log(m_c)}{P} + k \log(m)) = O(\frac{n \log(m_c)}{P} + k \log(m))$ (Since components are decomposed from a name, we get $n = kn_c$).

In a character trie, the average lookup performance is O(nm_t). Compared with character trie, NCE can gains $\frac{n \log(m_c) + Pk \log(m)}{Pnm_t}$ speedup for longest name prefix lookup.

In the worst case, all transitions of a state in Transition Array should be moved to new entries when a new transition is inserted, and states which contain the inserted component need to update the component's code. Therefore, the insertion procedure of CCT-STA has $O(n_c m_c + m_s)$ complexity in the worst case. And the worst insertion performance of ENPT-STA is $O(kn_c m_c + km + km_s) = O(nm_c + km + km_s)$. But in the average case, the complexity of insertion procedure is $O(n \log(m_c) + k \log(m) + k \log(m_s))$. As described above, deletion and modification operation has the same performance of lookup, $O(n \log m_c + k \log(m))$.

As described in Algorithm 2, the building procedure of ENPT-STA invokes insertion operation (Algorithm 1) to complete the work. So, a set has N names needs $O(N(nm_c + km + km_s))$ to build ENPT-STA in the worst case, or $O(N(n \log(m_c) + k \log(m) + k \log(m_s)))$ in the average case.

VI. Experimental Results

In this section, we evaluate the performance of NCE and compare it with the NCT in terms of memory compression, building time, lookup speedup, update time, and average packet delay, etc.

TABLE II
NUMBER OF DOMAINS WITH DIFFERENT COMPONENTS' NUMBER

Number of Components	1	2	3	4	5	6	7
DMOZ	0	10	170,709	2,325,677	454,953	48,702	4,193
Blacklist	0	0	1,334	1,449,493	272,693	322,689	3,772
ALEXA	0	0	177	870,108	117,468	10,646	1,328

Number of Components	8	9	10	11	12	13	14
DMOZ	132	10	2	0	0	0	0
Blacklist	458	311	51	12	2	24	47
ALEXA	147	86	33	4	3	0	0

A. Experimental Setup

NCT and NCE mechanism are implemented in C language, and the core programs include about 100 and 800 lines of code, respectively. The memory cost and time performance are measured on a PC with an Intel Core 2 Duo CPU of 2.8 GHz and DDR2 SDRAM of 8 GB.

Then we utilize the domain name information from D-MOZ [5], Blacklist [6] and ALEXA [7] to construct three datasets as our experiments' input. We extract 3,004,388 different domains from DMOZ's 4,328,389 different URL to construct DMOZ dataset. Using all the domain and URL collections in June, 2011 from Blacklist , we construct the Blacklist dataset which contains 2,050,886 domains. Similarly, we utilize the top 1,000,000 sites' domains form ALEXA as the ALEXA dataset. Besides, an IP lookup table from RRIP [8] which contains 354,641 IP prefixes is used to compare the lookup performance between name lookup and IP lookup. TABLE II shows the number of domains with different components' number obtained from these three datasets.

B. Effects of Code Allocation Mechanism

The Code Allocation Mechanism can effectively reduce the number of codes and shorten the length of each code. Fig. 6 shows the number of components and codes with different name collections sizes, and the compression ratio of Code Allocation Mechanism. Compared with the method that directly assigning one code to one component, our allocation method can save more than 50% codes when the size of name set is larger than 2,000K. The average encoded name length on three datasets is shown is TABLE III. We can see the encoded name length in NCE is much shorter than the average name length in NCT. For example, the average name length is reduced by 75.15% after encoding on DMOZ.

TABLE III
COMPARISON OF MEMORY USAGE

Dataset	Total Domains	Total Components	Avrage Name Length (bits)	NCT Size (MBytes)	NCT-STA (MBytes)	NCE Size(MBytes)				Compression Ratio (NCT vs NCE)	Compression Ratio (NCT-STA vs NCE)
						CCT-STA	ENPT-STA	Total Size	Encoded Name Length (bits)		
DMOZ	3,000,000	12,392,934	165.11	403.05	331.93	199.81	72.45	272.27	41.03	32.45%	17.97%
ALEXA	1,000,000	4,143,835	158.17	132.492	109.11	63.69	15.95	79.64	43.50	39.89%	27.01%
Blacklist	2,050,886	9,136,076	171.55	272.737	224.61	113.21	55.30	168.51	45.21	38.22%	24.98%

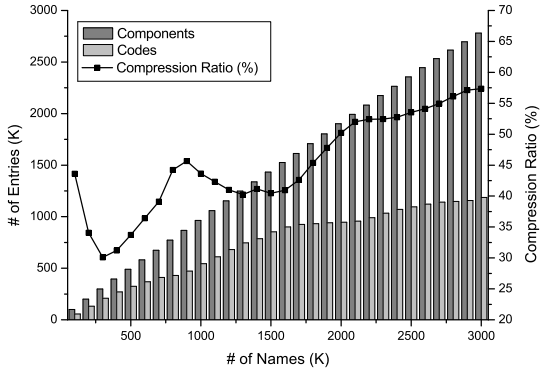

Fig. 6. The number of different components and codes, and the compression ratio of Code Allocate Mechanism on DMOZ dataset

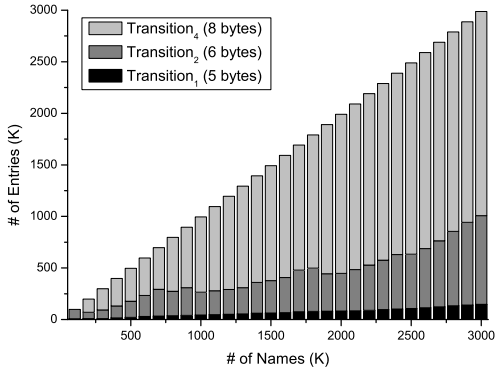

Fig. 7. Number of Entries for Transition_1, Transition_2 and Transition_4 on DMOZ dataset

C. Memory Compression

The memory cost in NCE includes two parts, (1) State Transition Arrays for ENPT and (2) State Transition Arrays for CCT. TABLE III presents the memory compression results of NCE. It shows ENPT only needs 18% memory size compared with the NPT , which is constructed as NCT. In other words, the memory cost of a FIB table can be compressed 82%. We can observe from TABLE III that compared with NCT, the memory cost of NCE is reduced by 30%. Even if NCT is implemented by State Transition Arrays, NCE still cuts down the memory usage by 20%.

As stated above, we use three types of Transition Arrays to implement ENPT. Fig. 7 illustrates the number of entries for these three kinds of Transition Arrays used on DMOZ dataset. When there are 3,000K domains, 66.3% of the total entries are Transition_4, 28.79% of the total entries are Transtion_2 and the left 4.91% are Transition_1. Compared with the method that implements the Transition Array with Transtioin_4 only, the solution which uses three sized arrays to construct Transition Arrays can reduce the memory cost by 33.7%.

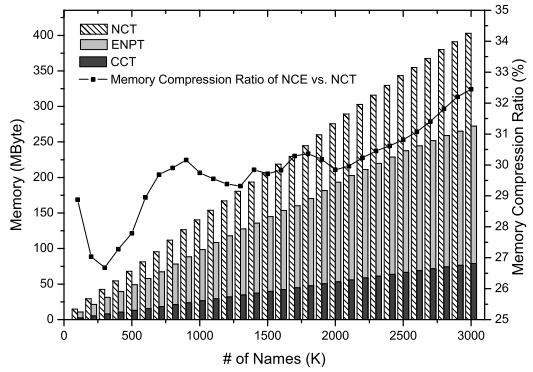

Fig. 8. The Memory Cost of NCE and NCT on DMOZ dataset

Fig. 8 depicts the memory cost of NCE, which includes the space of CCT-STA and ENPT-STA on DMOZ dataset. We can observe that with the increase of the number of names, NCE's memory compression ratio gradually grows too. Fig. 8 reveals that when the number of domains increases, NCE's components number and states number grows more slowly than those of NCT. Consequently, NCE's memory cost increases slower than that of NCT, which demonstrates that NCE is memory-efficient on both small domain set and quite large domain set.

D. Lookup Time and Speedup

In this subsection, we investigate the lookup performance of NCE. In order to get the average name lookup time, we input 100K random names each time and get the total execution time. Then names' average lookup time can be obtained. As illustrated in Fig. 9 and TABLE IV, when there are three parallel code lookup modules, NCE's average lookup time is about 1,800∼3,250 CPU cycles, which equals to 643ns∼1161ns since the CPU frequency is 2.8 GHz. And it needs 1,332.57 CPU cycles to look up an IP prefix in an IP table which is constructed by trie (2.1 million lookups per second), in the average case. So, NCE is an effective approach for accelerating the longest name prefix lookup.

Then we investigate the relationship between NCE's average lookup time and the number of parallel CCT lookup modules. We extract 1,000K names form DMOZ, Blacklist and Alexa respectively. Repeat the experiments and get the corresponding average lookup time. The experimental results are shown in Fig. 10.

TABLE IV
COMPARISON OF NCT AND NCE'S PROCESSING PERFORMANCE

Dataset	Total Domains	NCT			NCE(P=3)			Speedup
		Building Time(s)	Average Packet Lookup Time (CPU Cycle)	Packet Delay(us)	Building Time(s)	Average Packet Lookup Time (CPU Cycle)	Packet Delay(us)	
DMOZ	3,000,000	43.69	23,112.78	8.25	34.91	2,975.26	1.90	7.77
ALEXA	1,000,000	18.62	12,154.76	4.34	12.32	2,881.78	1.23	4.22
Blacklist	2,050,886	31.58	9,699.37	4.15	23.53	2,335.51	1.65	4.15

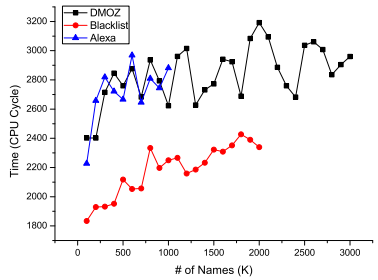

Fig. 9. NCE's Average Lookup Time (When the Number of Parallel CCT lookup modules is 3)

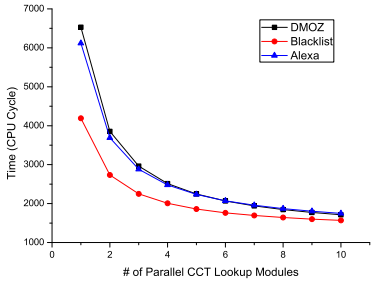

Fig. 10. The relationship between NCE's average lookuptime and the number of parallel CCT lookup modules

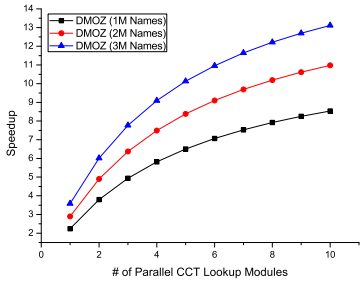

Fig. 11. The relationship between NCE's speedup and the number of parallel CCT lookup modules

Next we study the relationship between NCE's speedup and the number of parallel encoding modules. 3,000K names in DMOZ are constructed into three sets, which have 1,000K, 2,000K, and 3,000K names, respectively. Fig. 11 illustrates the results, which shows that the speedup performance of NCE gets better when the number of names increases. It also shows that NCE's speedup grows gradually accompanied with the increase of the number of CCT lookup modules.

E. Average Packet Delay

We extract 1,000K different names and calculate NCE's average packet delay using different number of parallel C-CT lookup modules. The experimental results are shown in Fig. 12.

In our experiments, there are several parallel CCT lookup modules and one ENPT lookup module. Using parallel CCT lookup modules, the CCT lookup delay could be reduced. However, as is shown in Fig. 12, when the number of parallel CCT lookup modules is greater than 6, NCE's average packet delay almost stay at the same level. Because the packet delay is determined by the maximal component code lookup time of a name when the number of parallel CCT modules is greater than the number of a name's components. When the number of parallel CCT lookup modules is greater than 6, almost all the components of a given name could be looked up concurrently since 99% of the existing domains have no more than 6 components (See Table II).

F. Update Performance

Fig. 13 shows the update time in the worst case. The time is calculated by inserting 100K new names to the exiting NCE with different initial NCE scales. It spends 142,143.46 CPU cycles (50.77 us) on inserting a new name to NCE which already has 2,900K names. In other words, NCE can handle more than 20,000 updates per second in the worst case.

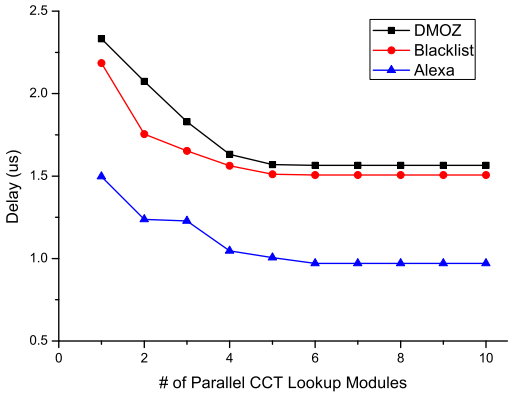

Fig. 12. The relationship between NCE's packet delay and the number of parallel CCT lookup modules

VII. RELATED WORK

In NDN Proposal [1], L. Zhang *et al*. propose a fast scalable name lookup mechanism that uses Ternary Content Address-able Memory (TCAM) as the basic hardware components. But this method directly loads names to TCAM which causes great waste of the valuable TCAM memory and leads to excessive power consumption.

Z. Genova *et al.* [9] and X. Li *et al.* [10] hash URLs to the fixed-length signatures, and look up the signature in the hash table, which has good URL lookup performance. However, these methods consider an URL as an indivisible entity, thus cannot provide longest prefix matching capability. In order to overcome the problem caused by longest prefix matching, Y. Yu *et al.* [11] propose a mechanism to map a whole name to a code, but it needs additional protocol to exchange codes tables between routers. Similarly, A. Singla *et al.* [12] and S. Jain *et al.* [13] apply the flat name (ID) to replace IP address in the future networks, their work focused on the routing

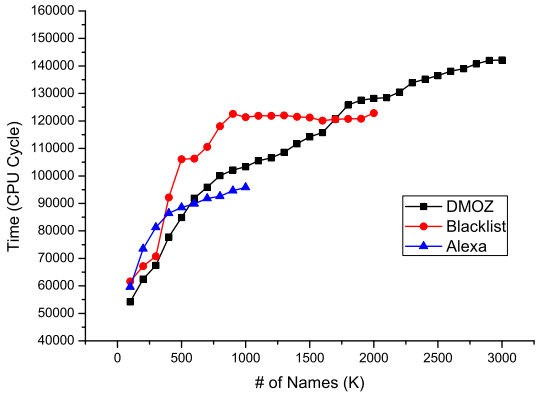

Fig. 13. The worst case update performance of three name collections

mechanism, and the performance of flat name forwarding is still unsatisfactory.

Some other methods try to decompose an URL into components and build a component tree to aggregate URLs [14]. The URL component tree is similar to our Name Prefix Trie, which can only offer basic prefix lookup and cannot satisfy the name lookup performance requirement in NDN. B. Michel *et al.* [3] design a compression algorithm named Hash Chain, which is based on a hierarchical URL decomposition, to aggregate URLs sharing common prefixes. Hash Chain uses an incremental hashing function to compress common prefixes length and minimize collisions between prefixes. Z. Zhou *et al.* [4] use CRC32 hash function to compress the URL components, and combine a multiple string matching algorithm for URL lookup engine. Hashing functions not only compress the memory of URL sets, but also accelerate the search speed by computing the searching keys. However, these hash-based algorithms have a fatal drawback, hash collision (false positive), making it cannot be applied to name lookup in NDN. False positive will cause the Interest packet cannot be forwarded to the right destination. Any possibility of hash collision will undermine the integrity of the basic functions of the NDN router.

VIII. Further Work and Conclusions

A. Further Work

The entry length in the Ternary Content Addressable Memory (TCAM) can be effectively cut down by the encoding method as discussed in Section IV. For example, in our DMOZ dataset, a TCAM entry needs about 165.11 bits to store a name. Using Code Allocation Mechanism, the length of an entry is shortened to 41.03 bits. It can save 75.15% precious TCAM memory by using codes to replace names, and reduce the power consumption of TCAM which is proportional to the length of TCAM entry. Besides, we can apply Binary Content Addressable Memory (BCAM) to implement CCT lookup to improve the overall name lookup performance.

B. Conclusion

In this paper, we have proposed an effective Name Components Encoding approach named NCE to reduce memory overhead and accelerate lookup speed for longest name prefix lookup in NDN. The technique involves a Code Allocation Mechanism and an evolutionary State Transition Arrays. Code Allocation Mechanism reuses the codes as much as possible. The evolutionary State Transition Arrays for Encoded Name Prefix Trie and Component Character Trie reduces the memory cost further while accelerating lookup speed. Both theoretical analysis and experiments on real domain sets demonstrate that NCE could effectively reduce the memory cost while guaranteeing high-speed of longest name prefix lookup.

IX. Acknowledgement

This paper is supported by NSFC (61073171, 60873250), Tsinghua University Initiative Scientific Research Program, the Specialized Research Fund for the Doctoral Program of Higher Education of China(20100002110051), China Postdoctoral Science Foundation (20110490387), and Ningbo Natural Science Foundation (2010A610121).

References

[1] L. Zhang, D. Estrin, V. Jacobson, and B. Zhang, "Named data networking (ndn) project," in *Technical Report, NDN-0001*, 2010. [Online]. Available: http://www.named-data.net/

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, ser. CoNEXT '09. New York, NY, USA: ACM, 2009, pp. 1–12.

[3] B. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang, "Url forwarding and compression in adaptive web caching," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, 2000, pp. 670 –678 vol.2.

[4] Z. Zhou, T. Song, and Y. Jia, "A high-performance url lookup engine for url filtering systems," in *Communications (ICC), 2010 IEEE International Conference on*, may 2010, pp. 1 –5.

[5] ODP - Open Directory Project. [Online]. Available: http://www.dmoz.org/

[6] Blacklist. [Online]. Available: http://urlblacklist.com/

[7] Alexa the Web Information Company. [Online]. Available: http://www.alexa.com/

[8] ripe. [Online]. Available: http://rrc00.ripe.net/

[9] G. Z. and C. K., "Managing routing tables for url routers in content distribution networks," *International Journal of Network Management*, vol. 14, pp. 177–192, 2004.

[10] X. Li and W. Feng, "Two effective functions on hashing url," *Journal of Software*, vol. 15, pp. 179–184, 2004.

[11] Y. Yu and D. Gu, "The resource efficient forwarding in the content centric network," in *NETWORKING 2011*, ser. Lecture Notes in Computer Science, J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio, Eds. Springer Berlin / Heidelberg, 2011, vol. 6640, pp. 66–77.

[12] A. Singla, P. B. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy, "Scalable routing on flat names," in *Proceedings of the 6th International Conference*, ser. Co-NEXT '10. New York, NY, USA: ACM, 2010, pp. 20:1–20:12.

[13] S. Jain, Y. Chen, Z.-L. zhang, and S. Jain, "Viro: A scalable, robust and namespace independent virtual id routing for future networks," in *INFOCOM 2011. 30th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2011.

[14] C. Park and S. Hwang, "Fast url lookup using url prefix hash tree," in *Available: http://dbpia.co.kr/*, 2008.